Ingres® 2006 Release 3

Embedded SQL Companion Guide

INGRES®

October 2007

Contents

Chapter 1: About This Guide

Chapter 2: Embedded SQL for C

Chapter 3: Embedded SQL for COBOL

Chapter 4: Embedded SQL for Fortran

Chapter 5: Embedded SQL for Ada

Chapter 6: Embedded SQL for BASIC

Chapter 7: Embedded SQL for Pascal

Chapter 1: About This Guide

This chapter briefly describes the *Embedded SQL Companion Guide* and discusses how to use this manual most effectively. The chapter also describes conventions used in Ingres documentation, and lists other manuals that are relevant to this manual.

Purpose of This Manual

This guide describes how to use Ingres Embedded SQL with the following programming languages:

- C and C++
- COBOL
- Fortran
- Ada
- BASIC
- Pascal

For the most part, embedded SQL is identical in syntax and functionality across all supported host programming languages. Therefore, the documentation describes it independently of any one host language in the *SQL Reference Guide*, which covers database statements, and in the *Forms-based Application Development Tools User Guide*, which covers forms statements. The host language-dependent details of its use are described in this Companion Guide.

Audience

This manual is designed for programmers who have a working knowledge of SQL and C, COBOL, and Fortran. It must be read in conjunction with the *SQL Reference Guide* and the *Forms-based Application Development Tools User Guide*, as it discusses only those issues on which the various host languages diverge.

Contents

Each chapter in this guide discusses embedded SQL for a particular language. Each chapter contains the following sections:

Section	Description
Embedded SQL Statement Syntax	Language-specific issues of embedded SQL statement syntax
Variables and Data Types	Declaration and use of language-specific program variables in embedded SQL
The SQL Communications Area	The SQL Communications Area as implemented in the language
Dynamic Programming	Dynamic SQL as implemented in the language
Advanced Processing	User-defined handlers
Preprocessor Operation	The operation of the embedded SQL preprocessor for the language and the steps required to create, compile, and link an embedded SQL program
Preprocessor Error Messages	A list of embedded SQL preprocessor error messages specific to the language
Remaining sections	Sample programs that illustrate many embedded SQL features

Enterprise Access Compatibility

This document assumes that your installation does not include an Enterprise Access product. If your installation does include one or more Enterprise Access products, check your OpenSQL documentation for information about syntax that may differ from that described in this manual.

Areas that may differ include:

- Varchar data type length
- Legal row size
- Command usage
- Name length
- Table size

Conventions

This section describes the conventions that Ingres documentation uses for consistency and clarity.

Statements and Commands

Ingres documentation handles statements and commands as follows.

Terminology

The documentation observes the following distinction in terminology:

- A *command* is an operation that you execute at the operating system levelA *statement* is an operation that you embed in a program or execute interactively from an Ingres terminal monitor

  A statement can be written in Ingres/4GL, a host programming language (such as Fortran), or a database query language (SQL or QUEL).

Syntax

This manual uses the following conventions to describe statement and command syntax specifications:

Convention	Usage
Boldface	Indicates keywords, symbols or punctuation that you must type as shown
Italics	Represent a variable name for which you must supply an actual value
[ ] (brackets)	Indicate an optional item
{ } (braces)	Indicate an optional item that you can repeat as many times as appropriate
| (vertical bar)	Used between items in a list to indicate that you should choose one of the items

The following example illustrates the syntax conventions:

create table *tablename* **(***columnname format*
*{,columnname format}***)**
[**with**_*clause*]

System-Specific Text

Although Ingres generally operates the same way on all systems, you need to know about a few system-specific differences. Where information differs by system, read the information that follows the name of your system, as follows:

UNIX

This text is specific to the UNIX environment.

VMS

This text is specific to the VMS environment.

Windows

This text is specific to the Windows environment.

The symbol indicates the end of the system-specific text.

In some instances, system-specific differences are indicated by using parenthesis ( ). For example: Filename specifies a filename or a system environment variable (UNIX) or a logical name extension (VMS) that points to the file name.

Related Manuals

This guide is part of a series of manuals that describe the full range of Ingres products.

To learn more about Ingres concepts and functions related to embedded SQL, see the following manuals:

- SQL Reference Guide
- Character-based Querying and Reporting Tools User Guide
- Forms-based Application Development Tools User Guide

Chapter 2: Embedded SQL for C

This chapter describes the use of Ingres Embedded SQL with the C and C++ programming languages.

Embedded SQL Statement Syntax for C

This section describes the language-specific issues inherent in embedding SQL database and forms statements in a C or C++ program. An embedded SQL database statements has the following general syntax:

[margin] **exec sql** *SQL_statement terminator*

The syntax of an embedded SQL/FORMS statement is almost identical:

[margin] **exec frs** *SQL/FORMS_statement terminator*

For information on SQL statements, see the *SQL Reference Guide*. For information on SQL/FORMS statements, see the *Forms-based Application Development Tools User Guide*.

The following sections describe the various syntactical elements of these statements as implemented in C.

Margin

There are no specified margins for embedded SQL statements in C. The **exec** keyword can begin anywhere on the source line.

Terminator

The terminator for C is the semicolon (;). The following example shows a **select** statement embedded in a C program:

```
exec sql select ename
    into :namevar
    from employee
    where eno = :numvar;
```

Do not follow an embedded statement on the same line with a C statement or another embedded statement. This causes preprocessor syntax errors on the second statement. Use only comments and white space (blanks and tabs) after the C terminator to the end of the line.

Labels

Like C statements, embedded SQL statements can have a label prefix. The label must begin with an alphabetic character or an underscore. The label must be the first word on the line (optionally preceded by white space), and must be terminated with a colon (:). For example:

```
close_cursor: exec sql close cursor1;
```

The label can appear anywhere a C label can appear. However, although the preprocessor accepts a label before any exec sql or exec frs prefix, you cannot label some lines. For example, although the preprocessor accepts the following, the compiler does not because labels are not allowed before declarations:

```
include_sqlca: exec sql include sqlca;
```

As a general rule, use labels only with executable statements.

Line Continuation

There are no line continuation rules for embedded SQL statements in C. Statements extend to the C terminator. Blank lines can also be included.

Comments

You can include C comments, delimited by /* and */ anywhere in an embedded SQL statement that a blank is allowed, with the following exceptions:

- Between the margin and the word exec (whether or not you have a C label prefix).
- Between the word exec and the word sql or frs. In the following example, comments cause both statements to be interpreted as C host code:

  ```
  /* Initial comment */ exec sql include sqlca;
  exec /* Between */ sql commit;
  ```

- Between words that are reserved when they appear together. For the list of double reserved words contained in the list of keywords, see the *SQL Reference Guide*.
- In string constants.
- In parts of statements that are dynamically defined. For example, a comment in a string variable specifying a form name is interpreted as part of the form name.

- Between component lines of embedded SQL/FORMS block-type statements. All block-type statements (such as activate and unloadtable) are compound statements that include a statement section delimited by begin and end. Comment lines must not appear between the statement and its section. The preprocessor interprets such comments as C host code, which causes preprocessor syntax errors. (However, comments can appear on the same line as the statement.)

  For example, the following statement causes a syntax error on the C comment:

  ```
  exec frs unloadtable empform
       employee (:namevar = ename);
  /* Illegal comment before statement body */
  exec frs begin; /* Comment legal here */
       strcat(msgbuf, namevar);
  exec frs end;
  ```

- Between any components in a statement composed of more than one compound statement. An example of such a statement is the display statement, which typically consists of the display clause, an initialize section, activate sections, and a finalize section. C comments are translated as host code and cause syntax errors on subsequent statement components.

You can also use the SQL comment delimiter (--) to indicate that the rest of the line is a comment. For example:

```
exec sql delete            --Delete all employees
    from employee;
```

String Literals

Use single quotes to delimit embedded SQL string literals. To embed a single quote in a string literal, you must double it. For example:

```
exec sql insert
into comments (anecdotes)
 values ('single'' quote followed by double " quote');
```

This insert writes the string:

```
single' quote followed by double " quote
```

Into the anecdotes column of the comments table.

In embedded SQL statements, the double quote and backslash need not be escaped because they have no special meaning.

To continue a string literal to additional lines, use the backslash (\) character. Any leading spaces on the next line are considered part of the string. This follows the C convention. For example, the following message statement is legal:

```
exec frs message 'Please correct errors found in\
      updating the  database tables.'
```

Use C conventions in the declaration section. You must use double quotes to delimit most C strings. For example:

```
char *dbname = "personnel";
```

String Literals and Statement Strings

The Dynamic SQL statements prepare and execute immediately, both use statement strings that specify an SQL statement. To specify the statement string, use a string literal or character string variable, as follows:

```
exec sql execute immediate 'drop employee';
 str = "drop employee";
exec sql execute immediate :str;
```

As with regular embedded SQL string literals, the statement string delimiter is the single quote. However, quotes embedded in statement strings must conform to SQL runtime rules when the statement executes. For example, the following dynamic insert statement:

```
exec sql prepare s1 from
  'insert into t1 values (''single''''double"slash\ '')';
```

is equivalent to the statement:

```
str = "insert into t1 values
    ('single''double\"slash\\ ')";
 exec sql prepare s1 from :str;
```

In fact, the string literal that the embedded SQL/C preprocessor generates for the first example matches the string literal assigned to the variable str in the second example. The runtime evaluation of the above statement string is:

```
insert into t1 values ('single''double"slash\ ');
```

Avoid using a string literal for a statement string whenever it contains quotes or the backslash character. Instead, build the statement string using the C language's rules for string literals together with the SQL rules for the runtime evaluation of the string.

The Create Procedure Statement

The create procedure statement has language-specific syntax rules for line continuation, string literal continuation, comments, and the final terminator. These syntax rules follow the rules discussed in this section. For example, the final terminator is a semicolon. Although the preprocessor treats the create procedure statement as a single statement which is terminated with a semicolon, you must terminate all statements in the body of the procedure with a semicolon.

The following example shows a create procedure statement that follows the embedded SQL for C (ESQL/C) syntax rules:

```
exec sql
 create procedure proc (parm integer) as
 declare
     var integer;
 begin
    if parm > 10 then /* Use C comment delimiter*/
    message 'C strings can continue (use backslash)    \over lines';
    insert into tab values (:parm);
    endif;
 end;
```

Creating Sub-Processes in ESQL/C Programs

Since child processes created by fork(), vfork(), or exec() system calls do not share the parent processes' status information, processes created in this way may experience protocol problems. The recommended method for creating sub-processes is to use exec sql call system.

C Variables and Data Types

This section describes how to declare and use C program variables in embedded SQL.

Variable and Type Declarations

The following sections describe the various variable and type declarations.

Embedded SQL Variable Declaration Sections

Embedded SQL statements use C variables to transfer data from the database or form into the program. You must declare C variables to SQL before you can use them in any embedded SQL statement.

Declare C variables to SQL in a declaration section. For example:

```
exec sql begin declare section;
```

C variable and type declarations.

```
exec sql end declare section;
```

Do not place a label in front of the exec sql end declare section statement because it causes a preprocessor syntax error.

Embedded SQL variable declarations are global to the program file from the point of declaration onwards. You can incorporate multiple declaration sections into a single program, as is the case when a few different C procedures issue embedded statements using local variables. Each procedure can have its own declaration section. For more information on the declaration of variables and types that are local to C procedures, see The Scope of Variables in this chapter.

Reserved Words in Declarations

The following C keywords are reserved. Therefore, you cannot declare types or variables with the same name as these keywords:

auto	**extern**	**int**	**typedef**
char	**float**	**long**	**union**
const	**globalconstdef**	**register**	**unsigned**
define	**globaldef**	**short**	**varchar**
double	**globalconstref**	**static**	**volatile**
enum	**globalref**	**struct**	

Not all C compilers reserve every keyword listed. However, the embedded SQL/C preprocessor does reserve all these words.

The embedded SQL preprocessor does not distinguish between uppercase and lowercase in keywords. When it generates C code, it converts any uppercase letters in keywords to lowercase.

For example, although the following declarations are initially unacceptable to the C compiler, the preprocessor converts them into legitimate C code:

```
# defINE ARRSIZE 256; /*"defINE"converts to "define" */
INT numarr[ARRSIZE];  /*"INT" is equivalent to "int" */
```

The rule just described is true only for keywords. The preprocessor does distinguish between case in program-defined types and variables.

Variable and type names must be legal C identifiers beginning with an underscore or alphabetic character.

Data Types

The embedded SQL/C preprocessor accepts the C data types shown in the following table. This table maps these types to their corresponding Ingres types. For further information on exact type mapping between Ingres and C data, see Data Type Conversion in this chapter.

C Data Type	Ingres Data Type
long	integer
int	integer
short	integer
char (no indirection)	integer
double	float
float	float
char * (character pointer)	character
char [ ] (character buffer)	character
unsigned	integer
unsigned int	integer
unsigned long	integer
unsigned short	integer
unsigned char	integer
long int	integer
short int	integer
long float	float

Integer Data Type

The embedded SQL preprocessor accepts all C integer data types. Even though some integer types do have C restrictions (for example, a variable of type short must have a value that can fit into two bytes) the preprocessor does not check these restrictions. At runtime, data type conversion is effected according to standard C numeric conversion rules. For details on numeric type conversion, see Data Type Conversion in this chapter.

The type adjectives long, short, or unsigned can qualify the integer type.

In the type mappings table previously shown, the C data type char has three possible interpretations, one of which is the Ingres integer data type. The adjective unsigned can qualify the char data type when using it as a single-byte integer. If you declare a variable of the char data type without any C indirection, such as an array subscript or a pointer operator (the asterisk), it is considered a single-byte integer variable. For example:

```
char age;
```

The above example is a legal declaration and can be used as an integer variable. If the variable is declared with indirection, then it is considered an Ingres character string.

You can use an integer variable with any numeric-valued object to assign or receive numeric data. For example, you can use it to set a field in a form or to select a column from a database table. You can also specify simple numeric objects, such as table field row numbers as shown in the following example:

```
char  age;             /* Single-byte integer */
short empnums[MAXNUMS]; /* Array of 2-byte integers */
long  *global_index;    /* Pointer to 4-byte integer */
unsigned int overtime;
```

Floating-point Data Type

The preprocessor accepts float and double as legal floating-point data types. The internal format of double variables is the standard C runtime format.

VMS

If you declare long floating variables to interact with the Ingres runtime routines, you should not compile your program with the g_float command line qualifier (assuming that you are using the VAX C compiler). This qualifier changes the long float internal storage format, causing runtime numeric errors.

You can only use a floating-point variable to assign or receive floating-point numeric data. You cannot use it to specify numeric objects, such as table field row numbers. The preprocessor accepts long float as a synonym for double, for example:

```
float  salary;
 double sales;
```

is equivalent to:

```
float salary;
 long float sales;
```

Both are accepted by the preprocessor.

Character String Data Type

Any variables built up from the char data type, except simple variables declared without any C indirection, are compatible with any Ingres character string objects. As previously mentioned, a variable of type char declared without any C indirection is considered an integer variable.

The preprocessor treats an array of characters and a pointer to a character string in the same way. Always null terminate a character string if you are assigning it to an Ingres object. Ingres automatically null terminates any character string values that are retrieved into C character string variables. Consequently, any variable that you use to receive Ingres values should be declared as the maximum object length, plus one extra byte for the C null character. For more information, see Runtime Character Type Conversion in this chapter.

The following example declares three character variables—one integer and two strings:

```
char age      /* Single byte integer */
char *name;   /* Use as a pointer to a static string */
char buf[16]; /* Use to receive string data */
```

Character strings containing embedded single quotes are legal in SQL, for example:

```
mary's
```

User variables may contain embedded single quotes and need no special handling unless the variable represents the entire search condition of a where clause:

```
where :variable
```

In this case you must escape the single quote by reconstructing the *:variable* string so that any embedded single quotes are modified to double single quotes, as in:

```
mary''s
```

Otherwise, a runtime error will occur.

For more information on escaping single quotes, see String Literals in this chapter. For more information on character strings that contain embedded nulls, see The Varying Length String Type in this chapter.

Define Declaration

The preprocessor accepts the # define directive, which defines a name to be a *constant_value*. The preprocessor accepts the *constant_name* when it is in an embedded SQL statement and treats it as if a *constant_value* had been given.

The syntax for the # define statement is:

 # define constant_name constant_value

Syntax Notes:

- The *constant_value* must be an integer, floating-point, or character string literal. It cannot be an expression or another name. It cannot be left blank, as happens if you intend to use it later with the # ifdef statement. If the value is a character string constant, you must use double quotes to delimit it. Do not delimit it with single quotes to make the *constant_name* be interpreted as a single character constant, because the preprocessor translates the single quotes into double quotes. For example, the preprocessor interprets both of the following names as string constants, even though the first might be intended as a character constant:

```
# define quitflag 'Q'
    # define errormsg "Fatal error occurred."
```

- The preprocessor does not accept casts before *constant_value*. In general, the preprocessor does not accept casts, and it interprets data types from the literal value.
- Do not terminate the statement with a semicolon.

You can only use a defined constant to assign values to Ingres objects. Attempting to retrieve Ingres values into a constant causes a preprocessor error. For example:

```
exec sql begin declare section;
 # define MINEMPNUM 1
# define MAXSALARY 150000.00
# define DEFAULTNM "No-name"
exec sql end declare section;
```

Embedded SQL statements in the program can reference *:constant_name*. For example:

```
exec frs putform formname (salary= :MAXSALARY);
```

Variable Declarations Syntax

The syntax of a variable declaration is:

[storage_class] [class_modifier] type_specification
declarator {**,** declarator}**;**

where each *declarator* is:

variable_name [**=** initial_value]

Syntax Notes:

- *Storage_class* is optional but, if specified, can be any of the following:

  auto

  extern

  register

  static

  varchar

VMS

The following *storage_classe*s are VMS only:

globaldef
globalref

The storage class provides no data type information to the preprocessor. The varchar storage class is described in more detail later.

- *Class_modifier* is optional, and can be one of the following:

  const

  volatile

The class modifier provides no information to the preprocessor, and is merely passed through to the C compiler. Use of const and volatile keywords in ESQL/C data declarations is supported to the extent specified in the ANSI/ISO SQL-92 standard for embedded SQL C. That does not include all the possible uses of const and volatile that are accepted by the C compiler.

- Begin a variable or type name with an alphabetic character, but follow it with alphanumeric characters or underscores.
- Although register variables are supported, be careful when using them in embedded SQL statements. In input/output statements, such as the insert and select statements, you can pass a variable by reference with the ampersand operator (&). Some compilers do not allow you to use register variables this way.
- The *type_specification* must be an embedded SQL/C type, a type built up with a typedef declaration (and known to the preprocessor), or a structure or union specification. Typedef declarations and structures are discussed in detail later.
- Precede the *variable_name* by an asterisk (*), to denote a pointer variable, or follow it with a bracketed expression ([*expr*]), to denote an array variable. Pointers and arrays are discussed in more detail later.
- Begin the *variable_name,* which must be a legal C identifier name, with an underscore or alphabetic character.
- Variable names are case sensitive; that is, a variable named empid is different from one named Empid.
- Do not use a previously defined typedef name for a variable name. This also applies to any variable name that is the name of a field in a structure declaration.
- The preprocessor does not parse *initial_value*. Consequently, the preprocessor accepts any initial value, even if it can later cause a C compiler error. For example, the preprocessor accepts both of the following initializations, even though only the first is a legal C statement:

```
char    *msg = "Try again";
int      rowcount = {0, 123};
```

The following example illustrates typical variable declarations:

```
extern int   first_employee;
 auto long    update_mode = 1;
 static char  *names[3] = {"neil","mark","barbara"};
static char  *names[3] = {"john","bob","tom"};
char         **nameptr = names;
 short        name_counter;
 float        last_salary = 0.0, cur_salary = 0.0;
 double       stat_matrix[STAT_ROWS][STAT_COLS];
 const char   xyz[] = “xyz”;
```

Type Declarations Syntax

The syntax of a type declaration is:

typedef *type_specificationtypedef_name* {**,** *typedef_name*}**;**

Syntax Notes:

- The typedef keyword acts somewhat like a storage class specifier in a variable declaration, the only difference being that the resulting *typedef_name* is marked as a type name and not as a variable name.
- The *type_specification* must be an embedded SQL/C type known to the preprocessor, a type built up with a typedef declaration, or a structure or union specification. Structures are discussed in more detail later.
- Use an asterisk (*) before the *typedef_name* to denote a pointer type, or follow it with a bracketed expression ([*expr*]) to denote an array type. Pointers and arrays are discussed in more detail later.
- The preprocessor accepts an *initial_value* after *typedef_name*, although you should avoid putting one there because it does not signify anything. Most C compilers allow an *initial_value* that is ignored after the *typedef_name*.
- Once you declare a typedef name, it is reserved for all subsequent declarations in the current scope. Thus variable names (including variable names that are names of fields in structure declarations) cannot have the same name as a previously defined typedef name.

The following example illustrates the use of type declarations:

```
typedef   short INTEGER2;
 typedef   char  CHAR_BUF[2001], *CHAR_PTR;

INTEGER2   i2;
 CHAR_BUF   logbuf;
 CHAR_PTR   name_ptr = (char *)0;
```

Array Declarations Syntax

The syntax of a C array declaration is:

array_name[*dimension*] {[*dimension*]}

In the context of a simple variable declaration, the syntax is:

type_specification array_variable_name[*dimension*] {[*dimension*]};

In the context of a type declaration, the syntax is:

typedef type_specification array_type_name[dimension] {[dimension]};

Syntax Notes:

- The preprocessor does not parse the *dimension* specified in the brackets. Consequently, the preprocessor accepts any dimensions. However, it also accepts illegal dimensions, such as non-numeric expressions, although these later cause C compiler errors. For example, the preprocessor accepts both of the following declarations, even though only the second is legal C:

```
typedef int SQUARE["bad expression"];
                /* Non-constant expression */
int     cube_5[5][5][5];
```

- You can specify any number of dimensions. The preprocessor notes the number of dimensions when the variable or type is declared. When you later reference the variable, it must have the correct number of indices.
- You can initialize an array variable, but the preprocessor does not verify that the initial value is an array aggregate.
- Variables cannot have grouping parentheses in their references or declarations.
- An array of characters is considered to be the pseudo character string type.

The following example illustrates the use of array declarations:

```
# define COLS 5

typedef short SQUARE[COLS][COLS];
 SQUARE        sq;

static  int   matrix[3][3] =
                { {11, 12, 13},
                {21, 22, 23},
                {31, 32, 33} };

char    buf[50];
```

Pointer Declarations Syntax

The syntax of a C pointer declaration is:

***** {*****} *pointer_name*

In the context of a simple variable declaration, the syntax is:

type_specification *****{*****} *pointer_variable_name***;**

In the context of a type declaration, the syntax is:

typedef *type_specification* *****{*****} *pointer_type_name***;**

Syntax Notes:

- You can specify any number of asterisks. The preprocessor notes the number specified when the variable or type is declared. When the variable is later referenced, it must have the correct number of asterisks.
- You can initialize a pointer variable, but the preprocessor does not verify that the initial value is an address.
- A pointer to the char data type is considered to be the pseudo character string type.
- Do not put grouping parentheses in variable references or variable declarations.
- You can use arrays of pointers.

The following example illustrates the use of pointer declarations:

```
extern int    min_value;
 int           *valptr = &min_value;
 char          *tablename = "employee";
```

Structure Declarations Syntax

A C structure declaration has three variants, depending on whether it has a tag and/or a body. The following sections describe these variants.

A Structure with a Tag and a Body

The syntax of a C structure declaration with a tag and a body is:

> **struct** *tag_name* **{**
> *structure_declaration* {*structure_declaration*}
> **}**

where *structure_declaration* is:

> *type_specification member* {**,** *member*}**;**

In the context of a simple variable declaration, the syntax is:

> **struct** *tag_name* **{**
> *structure_declaration* {*structure_declaration*}
> **}** [*structure_variable_name*]**;**

In the context of a type declaration, the syntax is:

> **typedef struct** *tag_name* **{**
> *structure_declaration* {*structure_declaration*}
> **}** *structure_type_name***;**

Syntax Notes:

- Wherever the keyword struct appears, the keyword union can appear instead. The preprocessor treats them as equivalent.
- Each member in a *structure_declaration* has the same rules as a variable of its type. For example, as with variable declarations, the *type_specification* of each member must be a previously defined type or another structure. Also, you can precede the member name by asterisks or follow it with brackets. Because of the similarity between structure members and variables, the following discussion focuses only on those areas in which they differ.

```
struct person
{
   charname[40];
   struct
   {
      int day, month, year;
   } birth_date;
 } owner;
```

- The preprocessor permits an initial value after each member name. Do not, however, put one there, because it causes a compiler syntax error.

- If you do not specify *structure_variable_name*, the declaration is considered a declaration of a structure tag.
- You can initialize a structure variable, but the preprocessor does not verify that the initial value is a structure aggregate.

The following example illustrates the use of tags and body:

```
# define MAX_EMPLOYEES 1500

typedef struct employee
{
       char    name[21];
       short   age;
       double  salary;
 } employee_desc;
 employee_desc employees[MAX_EMPLOYEES];
 employee_desc *empdex = &employees[0];
```

A Structure with a Body and No Tag

The syntax of a C structure declaration with a body and no tag is:

> **struct {**
> *structure_declaration* {*structure_declaration*}
> **}**

where *structure_declaration* is the same as in the previous section.
In the context of a simple variable declaration, the structure's syntax is:

> **struct {**
>   *structure_declaration* {*structure_declaration*}
> **}** *structure_variable_name***;**

In the context of a type declaration, the structure's syntax is:

> **typedef struct {**
>   *structure_declaration* {*structure_declaration*}
> **}** *structure_type_name***;**

Syntax Notes:

- All common clauses have the same rules as in the previous section. For example, struct and union are treated as equivalent, and the same rules apply to each structure member as to variables of the same type.
- Specify the *structure_variable_name* when there is no tag.
  The actual structure definition applies only to the variable being declared.

The following example illustrates the use of a body with no tag:

```
# define MAX_EMPLOYEES 1500

struct
{
        char      name[21];
        short     age;
        double    salary;
 } employees[MAX_EMPLOYEES];
```

A Structure with a Tag and No Body

The syntax of a C structure declaration with a tag and no body is:

struct *tag_name*

In the context of a simple variable declaration, the syntax is:

struct *tag_name structure_variable_name***;**

In the context of a type declaration, the syntax is:

typedef struct *tag_name structure_type_name***;**

Syntax Notes:

- All common clauses have the same rules as in the previous section. For example, struct and union are treated as equivalent, and you can initialize the structure without the preprocessor checking for a structure aggregate.
- The *tag_name* must refer to a previously defined structure or union. The preprocessor does not support *forward* structure declarations. Therefore, when referencing a structure tag in this type of declaration, you must have already defined the tag. In the declaration below, the tag new_struct must have been previously declared:

  ```
  typedef struct new_struct *NEW_TYPE;
  ```

The following example illustrates the use of a tag and no body:

```
union a_name
{
   char    nm_full[30];
   struct
   {

          char nm_first[10];
          char nm_mid[2];
          char nm_last[18];
    } nm_parts;
 };

union a_name empnames[MAX_EMPLOYEES];
```

Enumerated Integer Types

An enumerated type declaration, enum, is treated as an integer declaration. The syntax of an enumerated type declaration is:

> **enum** [*enum_tag*]
> **{** *enumerator* [**=** *integer_literal*]
> {**,** *enumerator* [**=** *integer_literal*]} **}** [*enum_vars*];

The outermost braces ({ and }) represent braces that you have to type.

Syntax Notes:

- If you use the *enum_tag*, the list of enumerated literals (*enumerators*) and enum variables (*enum_vars*) is optional, like a structure tag without a body. The two declarations that follow are equivalent. The first declaration declares an *enum_tag*, while the second declaration uses that tag to declare a variable.

  First declaration:

  ```
  enum color {RED, WHITE, BLUE};/* Tag,
       no variable */
  enum color col;     /* Tag, no body,
       has variable */
  ```

  Second declaration:

  ```
  enum color {RED, WHITE, BLUE} col;/* Tag, body,

  has variable */
  ```

  If you do not use the *enum_tag*, the declaration must include a list of enumerators, in the same way as a structure declaration.

- You can use the enum declaration with any other variable declaration, type declaration, or storage class. For example, the following declarations are all legal:

  ```
  typedef enum {dbTABLE, dbCOLUMN, dbROW, dbVIEW,
       dbGRANT} dbOBJ;
   dbOBJ   obj, objs[10];
   extern  dbOBJ *obj_ptr;
  ```

- Enumerated variables are treated as integer variables and enumerated literals are treated as integer constants.

The Varying Length String Type

As mentioned in the section describing character strings, all C character strings are *null-terminated*. Ingres data of type char or varchar can contain random binary data including the zero-valued byte (the null byte or \0 in C terms). If a program uses a C char variable to retrieve or set binary data that includes nulls, the runtime system is not able to differentiate between embedded nulls and the null terminator.

In order to set and retrieve binary data that can include nulls, a new C storage class, varchar, has been provided for varying length string variables. varchar identifies the following variable declaration as a structure that describes a varying length string, namely, a 2-byte integer representing the count, and a fixed length character array. Like other storage classes, described in a previous section, the keyword varchar must appear before the variable declaration:

```
varchar struct {
    short       current_length;
    char        data_buffer[MAX_LENGTH];
 } varchar_structure;
```

Syntax Notes:

- The word varchar is reserved and can be in uppercase or lowercase.
- The varchar keyword is not generated to the output C file.
- The varchar storage class can only refer to a variable declaration, not to a type declaration. For example, the following declaration is legal because it declares the variable vch:

  ```
  varchar struct {
  short       buf_size;
  char        buf[100];
  } vch;
  ```

  But the varchar declaration of the structure tag vch (without a variable) is not legal and generates an error:

  ```
  varchar struct vch {
          short       buf_size;
          char        buf[100];
      };
  ```

- You can replace the structure definition of a varchar variable declaration by a structure tag or typedef reference. For example, the following typedef and varchar declarations are legal:

  ```
  typedef struct vch_ {
      short       vch_count;
      char        vch_data[VCH_MAX];
  } VCH;
  varchar VCH vch_1;         /* Typedef referenced */
  varchar struct vch_ vch_2; /* Structure tag */
                             /* referenced */
  ```

- You can use the varchar storage class for any type of variable declaration, including external and static variables, and to qualify nested structure members.

  For example, the following declarations are all legal:

```
static varchar struct _txt {
    short       tx_len;
    char        tx_data[TX_MAX];
} txt_var, *txt_ptr, txt_arr[10];

struct {
      char    ename[20];
      int     eage;
      varchar struct _txt ecomments;
} emp;

typedef short   buf_size;
typedef char    buf[512];

varchar struct {
    buf_size  len;
    buf       data;
} vchar;
```

The Varying Length Binary Type

The Ingres data type varbyte behaves just like varchar except that it bypasses character set translation when transmitted across Heterogeneous Ingres/Net.

A special varbyte structure type exists, which behaves exactly like the varchar structure type except that the associated internal data type is varbyte instead of varchar. Typedefs and struct tag declarations are supported in exactly the same way as for varchar.

Note that when a retrieved byte value does not fit into the embedded variable provided it will be truncated and a "Warning - string data, right truncation" condition is set via SQLSTATE and sqlca.sqlwarn1. This is identical to the handling of string truncation for character data.

Syntax Notes:

- The word varbyte is reserved and can be in uppercase or lowercase.
- The varbyte keyword is not generated to the output C file.
- The varbyte storage class can only refer to a variable declaration, not to a type declaration. For example, the following declaration is legal because it declares the variable vbyt:

```
varbyte struct {
    short       buf_size;
    char        buf[100];
} vbyt;
```

But the **varbyte** declaration of the structure tag vbyt (without a variable) is not legal and generates an error:

```
varbyte struct vbyt {
    short       buf_size;
    char        buf[100];
};
```

- You can replace the structure definition of a varbyte variable declaration by a structure tag or typedef reference. For example the following typedef and varbyte declarations are legal:

```
typedef struct vbyt_ {
   short       vbyt_count;
   char        vbyt_data[VCH_MAX];
} VBYT;

varbyte VBYT vbyt 1;        /* Typedef referenced */
varbyte struct vbyt_ vch_2; /* Structure tag */
                            /* referenced */
```

- You can use the varbyte storage class for any type of variable declaration, including external and static variables, and to qualify nested structure members. For example, the following declarations are legal:

```
static varbyte struct _txt {
       short       tx_len;
       char        tx_data[TX_MAX];
} txt_var, *txt_ptr, txt_arr[10];
struct v_ {
        short    length;
        char     data[MAXLEN];
};
VARBYTE struct v_ my_varbyte;

   typedef short   buf_size;
   typedef char    buf[512];

   varbyte struct {
         buf_size  len;
         buf       data;
} vbyte;
```

The DCLGEN Utility

DCLGEN (Declaration Generator) is a structure-generating utility that maps the columns of a database table into a structure that you can include in a declaration section.

The following command invokes DCLGEN from the operating system level:

dclgen *language dbname tablename filename structurename*

where:

- *language* is the embedded SQL host language, in this case, C.
- *dbname* is the name of the database containing the table.
- *tablename* is the name of the database table.

- *filename* is the output file into which the structure declaration is placed.
- *structurename* is the name of the host language structure that the command generates. The structure tag is the structure name followed by an underscore character (_).

This command creates the declaration file *filename*, containing a structure corresponding to the database table. The file also includes a declare table statement that serves as a comment and identifies the database table and columns from which the structure was generated.

When the file is generated, use an embedded SQL include statement to incorporate it into the variable declaration section. The following example demonstrates how to use DCLGEN in a C program.

Assume the Employee table was created in the Personnel database as:

```
exec sql create table employee
        (eno      smallint not null,
        ename     char(20) not null,
        age       integer1,
        job       smallint,
        sal       decimal(14,2) not null,
        dept      smallint)
        with  journaling;
```

and the DCLGEN system-level command is:

```
dclgen c personnel employee employee.dcl emprec
```

This command creates the employee.dcl file, which contains a comment and two statements. The first statement is the declare table description of employee, which serves as a comment. The second statement is a declaration of the C structure emprec. The contents of the employee.dcl file are:

```
/* Table employee description from database personnel */
exec sql declare employee table
            (eno        smallint not null,
            ename       char(20) not null,
            age         integer1,
            job         smallint,
            sal         decimal(14,2) not null,
            dept        smallint);

struct emprec_ {
            short     eno;
            char      ename[21];
            short     age;
            short     job;
            double    sal;
            short     dept;
 } emprec;
```

The length of the ename buffer is increased by one byte to accommodate the C null terminator. Also, the integer1 data type is mapped to short rather than char.

To include this file in an embedded SQL declaration section, use the embedded SQL include statement:

```
exec sql begin declare section;
       exec sql include 'employee.dcl';
 exec sql end declare section;
```

You can then use the emprec structure in a select, fetch, or insert statement.

The field names in the structure that DCLGEN generates are identical to the column names in the specified table. Therefore, if the column names in the table contain any characters that are illegal for host language variable names, you must modify the name of the field before attempting to use the variable in an application.

DCLGEN and Large Objects

When a table contains a large object column, DCLGEN will issue a warning message and map the column to a zero length character string variable. You must modify the length of the generated variable before attempting to use the variable in an application.

For example, assume that the job_description table was created in the personnel database as:

```
create table job_description (job smallint,
     description long varchar);
```

and the DCLGEN system-level command is:

```
dclgen c personnel job_description jobs.dcl jobs_rec
```

The contents of the jobs.dcl file would be:

```
/*Table job_description description from database
     personnel*/
    exec sql declare job_description table
        (job              smallint,
        description      long varchar);
    struct jobs_rec_ {
        short job;
        char  description[0];
   } jobs_rec;
```

Indicator Variables

An *indicator variable* is a 2-byte integer variable. You can use an indicator variable in an application in three ways:

- In a statement that retrieves data from the database, you can use an indicator variable to determine if its associated host variable was assigned a null.

- In a statement that writes data to the database, or to a form field, you can use an indicator variable to assign a null to the database column, form field, or table field column.
- In a statement that retrieves character (or byte) data, you can use the indicator variable as a check that the associated host variable was large enough to hold the full length of the returned string. However you can also use SQLSTATE to do this and it is the preferred method.

The base type for a null indicator variable must be the integer type short. Any type specification built up from short is legal. For example:

```
short   ind;        /* Indicator variable */
typedef short    IND;

IND   ind_arr[10]; /* Array of indicators */
IND   *ind_ptr;    /* Pointer to indicator */
```

The word indicator is reserved and cannot be used to define a type in a typedef statement.

When using an indicator array with a host structure, as described in Using Indicator Variables, you must declare the indicator array as an array of short integers (or a type built up from short). In the above example, you can use the variable ind_arr as an indicator array with a structure assignment.

Compiling and Declaring External Compiled Forms

You can precompile your forms in the Visual-Forms Editor (VIFRED). By doing so, you save the time otherwise required at runtime to extract the form's definition from the database forms catalogs. When you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in the C language. The following system specific section contains the remaining information you will need to declare your forms.

Windows Forms

Windows

VIFRED prompts you for the name of the file with the description. After creating the file, you can use the following cl command to compile it into linkable object code:

cl -c *filename*

The C compiler usually returns warning messages during this operation. You can suppress these, if you wish, with the -w flag on the cl command line. This command results in an object file that contains a global symbol with the same name as your form.

Before the embedded SQL/FORMS statement addform can refer to this global object, you must declare it in an embedded SQL declaration section, with the following syntax:

extern int **formname***;**

Syntax Notes:

- The *formname* is the actual name of the form. VIFRED gives this name to the address of the external object. The *formname* is also used as the title of the form in other embedded SQL/FORMS statements.
- The extern storage class associates the object with the external form definition.
- Although you *declareformname* as a pointer, you should *not* precede it with an asterisk when using it in the addform statement.

The example below shows a typical form declaration and illustrates the difference between using the form's object definition and the form's name:

```
exec sql begin declare section;
           extern int *empform;

           ...

exec sql end declare section;

         ...

exec frs addform :empform; /* the global object */
exec frs display empform; /* The name of the form */
         ...
```

UNIX Forms

UNIX

VIFRED prompts you for the name of the file with the description. After creating the file, you can use the following cc command to compile it into linkable object code:

cc -c *filename*

The C compiler usually returns warning messages during this operation. You can suppress these, if you wish, with the -w flag on the cc command line. This command results in an object file that contains a global symbol with the same name as your form.

Before the embedded SQL/FORMS statement addform can refer to this global object, you must declare it in an embedded SQL declaration section, with the following syntax:

extern int **formname***;**

Syntax Notes:

- The *formname* is the actual name of the form. VIFRED gives this name to the address of the external object. The *formname* is also used as the title of the form in other embedded SQL/FORMS statements.
- The extern storage class associates the object with the external form definition.
- Although you *declare formname* as a pointer, you should *not* precede it with an asterisk when using it in the addform statement.

The example below shows a typical form declaration and illustrates the difference between using the form's object definition and the form's name:

```
exec sql begin declare section;
         extern int *empform;

         ...

exec sql end declare section;

         ...

exec frs addform :empform; /* the global object */
exec frs display empform; /* The name of the form */

         ...
```

VMS Forms

VMS

After the file is created, you can use the following command to assemble it into a linkable object module.

macro *filename*

This command produces an object file that contains a global symbol with the same name as your form. Before the embedded SQL/FORMS statement addform can refer to this global object, you must declare it in an embedded SQL declaration section, with the following syntax:

globalref int ****formname*;**

Syntax Notes:

- The *formname* is the actual name of the form. VIFRED gives this name to the address of the external object. The *formname* is also used as the title of the form in other embedded SQL/FORMS statements.
- The globalref storage class associates the object with the external form definition.
- Although you declare *formname* as a pointer, you should *not* precede it with an asterisk when using it in the addform statement.

The example below shows a typical form declaration and illustrates the difference between using the form's object definition and the form's name:

```
exec sql begin declare section;
           globalref int *empform;

           ...

exec sql end declare section;

           ...

exec frs addform :empform; /* The global object */
exec frs display empform;  /* The name of the form */

           ...
```

Concluding Example

The following example demonstrates some simple embedded SQL/C declarations:

```
exec sql include sqlca; /* include error handling */
exec sql begin declare section;
 # define max_persons 1000

  typedef struct datatypes_/* Structure of all types */
      {
            char    d_byte;
            short   d_word;
            long    d_long;
            float   d_single;
            double  d_double;
            char   *d_string;
      } datatypes;
  datatypes d_rec;

  char        *dbname = "personnel";
      char        *formname, *tablename, *columnname;

  varchar struct {
            short   len;
            char    binary_data[512];
      } binary_chars;

  enum color {RED, WHITE, BLUE} col;

  unsigned int  empid;
      short int     vac_balance;
  struct person_   /* Structure with a union */
      {
            char      age;
            long      flags;
            union
            {
                  char full_name[30];
                  struct {
                         char firstname[12], lastname[18];
                  } name_parts;
            } person_name;
      } person, *newperson, person_store[MAX_PERSONS];

exec sql include 'employee.dcl'; /* From DCLGEN */
```

Windows

```
extern int *empform, *deptform; /* Compiled forms */
exec sql end declare section;
```

UNIX

```
extern int *empform, *deptform; /* Compiled forms */
exec sql end declare section;
```

VMS

```
globalref int *empform, *deptform; /* Compiled forms */
exec sql end declare section;
```

The Scope of Variables

The preprocessor references all variables declared in an embedded SQL declaration section and accepts them from the point of declaration to the end of the file. This may not be true for the C compiler, which only allows variables to be referred to in the scope of the nearest enclosing program block in which they were declared. If you have two unrelated procedures in the same file, each of which contains a variable with the same name to be used by embedded SQL, you do not have to redeclare the variable in a declaration section. The preprocessor uses the data type information supplied by the first declaration.

If you *do* redeclare the variable, the preprocessor confirms that both declarations have compatible data types and the same *indirection level*. The indirection level is the sum of the number of pointer operators preceding the variable declaration name and the number of array dimensions following the name. This redeclaration can only occur for simple, non-structured variable or formal procedure parameter declarations. Do not redeclare structures, typedefs, enumerated types and arrays even if used in a different context.

If you declare a variable name in two incompatible instances, the preprocessor generates an error and continues to process any references to the variable using only its first declaration. You can solve the problem by renaming the variables declared in the second and any subsequent declarations.

In the following program fragment, the variable dbname is passed as a parameter between two procedures. In the first declaration section, the variable is a local variable. In the second declaration section, the variable is a formal parameter passed as a string to be used with the connect statement. In both cases, the data type attributes are compatible character strings.

For example:

```
exec sql include sqlca;
 Access_Db()
{
          exec sql begin declare section;
                  char dbname[20];
          exec sql end declare section;

          /* Prompt for and read database name */
          printf("Database: ");
          gets(dbname);
          Open_Db(dbname);
              ...
 }

Open_Db(dbname)
 exec sql begin declare section;
          char *dbname;
 exec sql end declare section;
 {
          exec sql whenever sqlerror stop;
          exec sql connect :dbname;
          ...
 }
```

The above example is the first to demonstrate a formal parameter to a procedure in a declaration section. In this particular example, you do not need to declare the parameter, in which case the preprocessor uses the character string data type of the initial declaration of dbname. For example:

```
Open_Db(dbname)
 char *dbname;
 {
       exec sql whenever sqlerror stop;
       exec sql connect :dbname;
       ...
 }
```

To enhance the readability of the examples in this document, formal parameters are not declared. Instead, local variables are declared that can be initialized to formal parameters.

For example, the Open_Db procedure above could also be written as:

```
Open_Db(dbname)
 char *dbname;
 {
       exec sql begin declare section;
                    char *dbnm = dbname;
       exec sql end declare section;

       exec sql whenever sqlerror stop;
       exec sql connect :dbnm;
       ...
 }
```

Take special care when using variables in a declare cursor statement. The variables used in such a statement must also be valid in the scope of the open statement for that same cursor. The preprocessor actually generates the code for the declare at the point that the open is issued and, at that time, evaluates any associated variables. For example, in the following program fragment, even though the variable number is valid to the preprocessor at the point of both the declare cursor and open statements, it is not a valid variable name for the C compiler at the point that the open is issued.

For example:

```
Init_Csr1() /* This example contains an error */
{
        exec sql begin declare section;
            int number; /* A local variable */
        exec sql end declare section;

        exec sql declare cursor1 cursor for
            select ename, age
            from employee
            where eno = :number;

        /* Initialize "number" to a particular value */
          ...
 }

Process_Csr1()
{
         exec sql begin declare section;
                 char ename[16];
                 int age;
         exec sql end declare section;

         exec sql open cursor1; /* Illegal evaluation of
                                 "number" */
          exec sql fetch cursor1 into :ename, :age;
          ...
 }
```

Variable Usage

C variables that you declare in an embedded SQL declaration section can substitute for most elements of embedded SQL statements that are not keywords. Of course, the variable and its data type must make sense in the context of the element. When you use a C variable in an embedded SQL statement, precede it with a colon. You must further verify that the statement using the variable is in the scope of the variable's declaration. As an example, the following select statement uses the variables namevar and numvar to receive data, and the variable idno as an expression in the where clause:

```
exec sql select ename, eno
       into :namevar, :numvar
       from employee
       where eno = :idno;
```

Various rules and restrictions apply to the use of C variables in embedded SQL statements. The following sections describe the usage syntax of different categories of variables and provide examples of such use.

Simple Variables

The following syntax refers to a simple scalar-valued variable (integer, floating-point or character string):

:*simplename*

Syntax Notes:

- If you use the variable to send values to the database, or a field on a form, it can be any scalar-valued variable or **#** define constant, enumerated variable or enumerated literal.
- If you use the variable to receive values from the database or a field on a form, it can only be a scalar-valued variable or enumerated variable. Character strings that you declare as:

  char *character_string_pointer**;**

  or:

  char character_string_buffer**[];**

  are considered scalar-valued variables and must not include any indirection when referenced. External compiled forms that are declared as:

  extern int **compiled_formname*; (UNIX)

  globalref int **compiled_formname***;** (VMS)

  should not include any indirection when referenced in the addform statement:

  exec frs addform :*compiled_formname*;

The following program fragment demonstrates a typical message handling routine. It passes two scalar-valued variables as parameters: "buffer", a character string, and "seconds", an integer variable.

```
Print_Message(buffer, seconds)
 exec sql begin declare section;
       char *buffer;
       short seconds;
 exec sql end declare section;
 {
       exec frs message :buffer;
       exec frs sleep :seconds;
       ...
 }
```

Note: Ingres supports Unicode using Unicode Transformation Format 16 (UTF-16), representing Unicode code points in 16 bits (two octets). Embedded C for Ingres allows for variables of the C data type wchar_t to contain Ingres Unicode data. The C Standard does not specify a size for the wchar_t data type, however, if the compilation platform uses at least 16 bits for the data type wchar_t, it can be used for Ingres embedded C programs. When Ingres updates variables of the type wchar_t, only the low 16 bits are used; any extra high bits are set to zero. When Ingres reads values from wchar_t variables, only the low 16 bits are used and any extra high bits are ignored.

Array Variables

The following syntax refers to an array variable:

:*arrayname* **[***subscript***]** {**[***subscript***]**}

Syntax Notes:

- You must subscript the variable, because only scalar-valued elements (integers, floating-point and character strings) are legal SQL values.
- When you reference the array, the number of indices is noted but the embedded SQL preprocessor does not parse the subscript values. Consequently, even though the preprocessor confirms that you used the correct number of array indirections, the preprocessor accepts illegal subscript values. You must make sure that the subscript is legal. For example, the preprocessor accepts both of the following references, even though only the first is correct:

  ```
  float salary_array[5];
  :salary_array[0]
  :salary_array[+-1-+]A character string, declared as an array of characters,
  is not considered an array and cannot be subscripted in order to reference a
  single character. In fact, single characters are illegal string values, as
  all character string values must be null-terminated.
  ```

  For example, if the following variable were declared:

  ```
  static char abc[3] = {'a', 'b', 'c'};
  ```

you could not access the character "a" with the reference:

```
:abc[0]
```

To perform such a task, declare the variable as an array of three single character strings:

```
static char *abc[3] = {"a","b","c"};
```

- As with standard C, any variable that can be denoted with array subscripting can also be denoted with pointers. This is because the preprocessor only records the number of indirection levels used when referencing a variable. The indirection level is the sum of the number of pointer operators preceding the variable reference name and the number of array subscripts following the name. For example, if a variable is declared as an array:

  ```
  int age_set[2];
  ```

  it can be referenced as either an array:

  ```
  :age_set[0]
  ```

  or a pointer:

  ```
  :*age_set
  ```

- Do not precede references to elements of an array with the ampersand operator (&) to denote the address of the element.
- Any arrays of indicator variables that you use with structure assignments must not include subscripts.

The following example uses the variable "i" as a subscript. This variable does not need to be declared in the declaration section, as it is not parsed.

```
exec sql begin declare section;
  char *formnames[3={"empform","deptform","helpform"};
exec sql end declare section;
 int I;
 for (i = 0; i < 3; i++)
         exec frs forminit :formnames[i];
```

Pointer Variables

The following syntax refers to a pointer variable:

:*{*****}*pointername*

Syntax Notes:

- Refer to the variable indirectly, because only scalar-valued elements (integers, floating-point, and character strings) are legal SQL values.

- When you declare the variable, the preprocessor notes the number of preceding asterisks. Later references to the variable must have the same indirection level. The indirection level is the sum of the number of pointer operators (asterisks) preceding the variable declaration name and the number of array subscripts following the name.
- A character string, declared as a pointer to a character, is not considered a pointer and cannot be subscripted in order to reference a single character. As with arrays, single characters are illegal string values because any character string value *must* be null-terminated. For example, assuming the following declaration:

  ```
  char *abc = "abc";
  ```

  you could not access the character "a" with the reference:

  ```
  :*abcExternal compiled forms that you declare as:
  ```

  UNIX

  extern int ***compiled_formname;**

  VMS

  globalref ***compiled_formname;

  These external compiled forms must not include any indirection when referenced in the addform statement.
- As with standard C, any variable that you can denote with pointer indirection can also be denoted with array subscripting. This is true because the preprocessor only records the number of indirection levels used when referencing a variable. For example, if you declare a variable as a pointer:

  ```
  int *age_pointer;
  ```

  it can be referenced as either a pointer:

  ```
  :*age_pointer;
  ```

  or an array:

  ```
  :age_pointer[0];
  ```

The next section describes pointers to structures and members of structures.

The following example, uses a pointer to insert integer values into a database table:

```
exec sql begin declare section;
        int *numptr;
 exec sql end declare section;
 static int numarr[6] = {1, 2, 3, 4, 5, 0};

for (numptr = numarr; *numptr; numptr++)
    exec sql insert into items (number) values (:*numptr);
```

Structure Variables

You can use a structure variable in two different ways. First, you can use the structure as a simple variable, implying that all its members are used. This would be appropriate in the embedded SQL select, fetch, and insert statements. Second, you can use a member of a structure to refer to a single element. Of course, this member must be a scalar value (integer, floating-point or character string).

Using a Structure as a Collection of Variables

The syntax for referring to a complete structure is the same as referring to a simple variable:

:*structurename*

Syntax Notes:

- The *structurename* refers to a main or nested structure. It can be an element of an array of structures. Any variable reference that denotes a structure is acceptable. For example:

```
:emprec              /* A simple structure */
:struct_array[i]     /* An element of an array of structures */
:struct.minor2.minor3 /* A nested structure at level 3 */
```

- To use the final structure of the reference as a collection of variables, it must have no nested structures or arrays. The preprocessor enumerates all the members of the structure, which must have scalar values. The preprocessor generates code as though the program had listed each structure member in the order in which it was declared.

The following example uses the employee.dcl file generated by DCLGEN, to retrieve values into a structure:

```
exec sql begin declare section;
     exec sql include 'employee.dcl'; /* See above for
         description */
exec sql end declare section;

exec sql select *
        into :emprec
        from employee
        where eno = 123;
```

The example above generates code as though the following statement had been issued instead:

```
exec sql select *
      into :emprec.eno, :emprec.ename, :emprec.age,
                 :emprec.job, :emprec.sal, :emprec.dept
      from employee
      where eno = 123;
```

The following example fetches the values associated with all the columns of a cursor into a record:

```
exec sql begin declare section;
      exec sql include 'employee.dcl'; /* See above for
          description */
exec sql end declare section;

exec sql declare empcsr cursor for
      select *
      from employee
      order by ename;
      ...

exec sql fetch empcsr into :emprec;
```

The next example inserts values by looping through a locally declared array of structures whose elements have been initialized:

```
exec sql begin declare section;
      exec sql declare person table
            (pname    char(30),
                page     integer1,
                paddr    varchar(50));
      struct person_
      {
                char      name[31];
                short     age;
                char      addr[51];
      } person[10];
      int    i;
 exec sql end declare section;

...

for (i = 0; i < 10; i++)
{
    exec sql insert into person
        values (:person[i]);
 }
```

The insert statement in the example above generates code as though the following statement had been issued instead:

```
exec sql insert into person
      values (:person[i].name, :person[i].age,
      :person[i].addr);
```

Using a Structure Member

The syntax embedded SQL uses to refer to a structure member is the same as
in C:

:*structure***.***member*{**.***member*}

Syntax Notes:

- The *structure* member in the above statement must be a scalar value (integer, floating-point or character string). There can be any combination of arrays and structures, but the last object referenced must be a scalar value. Thus, the following references are all legal:

```
:employee.sal   /* Member of a structure */
:person[3].name /* Element member of an array */
:structure.mem2.mem3.age /* Deeply nested member */
```

- Any array elements referred to *within* the structure reference, and not at the very end of the reference, are not checked by the preprocessor. Consequently, both of the following references are accepted, even though one must be wrong, depending on whether person is an array:

```
:person[1].age
:person.ageStructure references can also include pointers to structures. The
arrow operator (->) denotes these structures. The preprocessor treats the arrow
operator exactly like the dot operator and does not check that the arrow is used
when referring to a structure pointer and that the dot is used when referring to
a structure variable.
```

For example, the preprocessor accepts both of the following references to a structure, even though only the second one is legal C:

```
Struct
{
    char *name;
    int   number;
} people[10], *one_person;

:people[i]->name  /* Should use the dot operator */
:one_person->name /* Correct use of pointer
                            qualifier */In general, the preprocessor supports
unambiguous and direct references to structure members, as in the following
example:
```

```
:ptr1->struct2.mem3[ind4]->arr5[ind6][ind7]
```

In this case, the last object denoted, arr5[ind6][ind7], must specify a scalar-valued object. References to structure variables cannot contain grouping parentheses. For example, assuming you declare struct1 correctly, the following reference causes a syntax error on the left parenthesis:

```
:(struct1.mem2)->num3
```

The only exception to this rule occurs when grouping a reference to the first and main member of a structure by starting the reference with a left parenthesis followed by an asterisk. Note that the two operators, "(" and "*" must be bound together without separating spaces, as in the following example:

```
:(*ptr1)->mem2
```

The following example uses the emprec structure that DCLGEN generates to put values into the empform form:

```
exec sql begin declare section;
      struct emprec_ {
            short   eno;
            char    ename[21];
            short   age;
            short   job;
            double  sal;
            short   dept;
      } emprec;
 exec sql end declare section;
        ...

exec frs putform empform
      (eno = :emprec.eno, ename = :emprec.ename,
                age = :emprec.age, job  = :emprec.job,
                sal = :emprec.sal, dept = :emprec.dept);
```

Using an Enumerated Variable (Enum)

The syntax for referring to an enumerated variable or enumerated literal is the same as referring to a simple variable:

*:enum_name***;**

Enumerated variables are treated as integer variables when referenced and you can use them to retrieve data from and assign data to Ingres. The enumerated literals are treated as declarations of integer constants and follow the same rules as integer constants declared with the # define statement. Use enumerated literals only to assign data to Ingres.

The following program fragment demonstrates a simple example of the enumerated type color:

```
exec sql begin declare section;

  exec sql declare clr table (num integer,color integer);
  typedef enum {RED, WHITE, BLUE} color;
  color col_var, *col_ptr;
  static COLOR col_arr[3] = {BLUE, WHITE, RED};
    int i;
 exec sql end declare section;
   /* Mapping from color to string */

static char *col_to_str_arr[3] = {"RED","WHITE", "BLUE"};
#   define ctos(c) col_to_str_arr[(int)c]

/* Fill rows with color array */
for (i = 0; i < 3; i++)
     exec sql insert into clr values (:i+1, :col_arr[i]);

/*
** Retrieve the rows - demonstrating a COLOR variable
** and pointer, and arithmetic on a stored COLOR value.
** Results are:
**    [1] BLUE, RED
**    [2] WHITE, BLUE
**    [3] RED, WHITE
*/
col_ptr = &col_arr[0];
 exec sql select num, color, color+1
        into :i, :col_var, :*col_ptr
        from clr;
 exec sql begin;
        printf("[%d] %s, %s\n", i, ctos(col_var),
              ctos(*col_ptr%3));
 exec sql end;
```

Using a Varying Length String Variable (Varchar or Varbyte)

The syntax for referring to a varchar (or varbyte) variable is the same as referring to a simple variable:

:varchar_name;

Syntax Notes:

- When using a variable declared with the varchar (or varbyte) storage class, you cannot reference the two members of the structure individually but only the structure as a whole. For example, the following declaration and select statement are legal:

```
varchar struct {
        short       buf_size;
        char        buf[100];

} vch;
   select data into :vch from objects;
```

But the following statement generates an error on the use of the member "buf_size":

```
select data, length(data)
       into :vch, :vch.buf_size
       from objects;
```

- When you use the variable to retrieve Ingres data, the 2-byte length field is assigned the length of the data, and the data is copied into the fixed length character array. The data is not null-terminated. You can use a varchar (or varbyte) variable to retrieve data in the select, fetch, inquire_sql, getform, finalize, unloadtable, getrow, and inquire_frs statements.
- When you use the variable to set Ingres data, the program must assign the length of the data (in the character array) to the 2-byte length field. You can use a varchar (or varbyte) variable to set data in the insert, update, putform, initialize, loadtable, putrow, and set_frs statements.

Using Indicator Variables

The syntax for referring to an indicator variable is the same as for a simple variable, except that an indicator variable is always associated with a host variable:

:host_variable:indicator_variable ***;***

or

:host_variable ***indicator*** *:indicator_variable;*

Syntax Notes:

- The indicator variable can be a simple variable, an array element or a structure member that yields a short integer. For example:

  ```
  short     ind_var, *ind_ptr, ind_arr[5];

                        :var_1:ind_var
                        :var_2:*ind_ptr
                        :var_3:ind_arr[2]
  ```

- If the host variable associated with the indicator variable is a structure, the indicator variable should be an array of short integers. In this case, the array should *not* be dereferenced with a subscript.
- When you use an indicator array, the first element of the array corresponds to the first member of the structure, the second element to the second member, and so on. Array elements begin at subscript 0, and not at 1 as in other languages.

The following example uses the employee.dcl file that DCLGEN generated to retrieve values into a structure and null indicators into the empind array:

```
exec sql begin declare section;
        exec sql include 'employee.dcl';
            /* See above for description */
        short    empind[10];
 exec sql end declare section;

exec sql select *
        into :emprec:empind
        from employee;
```

The above example generates code as though the following statement had been issued:

```
exec sql select *
  into:emprec.eno:empind[0], :emprec.ename:empind[1],
    :emprec.age:empind[2], :emprec.job:empind[3],
    :emprec.sal:empind[4], :emprec.dept:empind[5],
 from employee;
```

Using Varchar Variables for Logical Key Data Types

It is recommended that you use varchar variables to retrieve or insert Ingres logical key data types instead of char(8) or char(16) compatible variables. If logical key data contain embedded nulls, the Ingres runtime system may not be able to detect the end-of-string terminator on char variables; using varchar will eliminate this confusion between null-terminated strings and null data. System maintained logical keys are very likely to contain binary data including null bytes; therefore, you should always use a varchar variable when dealing with system maintained logical keys.

For example:

```
exec sql begin declare section;

exec sql declare keytab table
     (tkey table_key with system_maintained,
     okey object_key with sytem_maintained,
     row integer);

exec sql declare savetab table
     (tsave table_key not system_maintained,
     osave object_key not system_maintained);

#define tablen 8 /* Table_key length */
#define objlen 16 /* Object_key length */
varchar struct
{
    short   obj_len;
    char    obj_data[OBJLEN]];
 } objvar;

varchar struct
{
    short   tab_len;
    char    tab_data[TABLEN];
 } tabvar;

int     indx;
 short   tabind, objind;

exec sql end declare section;
. . .

exec sql insert into keytab (row) values (1);
/*
** Retrieve the table key and object key values
** that were just inserted by the system. Then
** INSERT the table key and object key values into
** another table with non-system maintained logical keys.
*/
exec sql inquire_sql (:tabvar:tabind = table_key,
                         :objvar:objind= object_key);
 if (tabind == -1 || objind == -1)
      printf ("No logical key values available.\n");
else
     exec sql insert into savetab (tsave, osave)
     values (table_key(:tabvar), object_key(:objvar));

/*
** Select data from a table that contains logical key
** data types.
*/
exec sql select tsave, osave into :tabvar, :objvar
      from savetab;
exec sql begin;
      /*Print out the table key value in Hex */
      printf (" Table key value = 0x");
           for (indx = 0; indx < tabvar.tab_len; indx++)

     {
                 printf ("%02x", (unsigned char)
    tabvar.tab_data[indx]);
         }
         printf ("\n");
exec sql end;
```

Declaring Function Arguments

If you intend to use function arguments in ESQL statements, you must declare the variable to the ESQL/C compiler. In non-ANSI style C functions, you can declare function arguments directly; for example:

```
void myfunct(arg1, arg2)

exec sql begin declare section;
     int arg1;

exec sql end declare section;
     int arg2;
```

In ANSI style functions, you cannot use the function argument variable directly. You must declare a local variable for use in ESQL statements, and copy the value from the function argument to the variable. For example:

```
void myANSIfunct(int arg1, int arg2)

exec sql begin declare section;
     int localarg1;

exec sql end declare section;
     int localarg2;
     localarg1 = arg1;

/* Now use localarg1 in your ESQL statements */
...
```

Data Type Conversion

A C variable declaration must be compatible with the Ingres value it represents. Numeric Ingres values can be set by and retrieved into numeric variables, and Ingres character values can be set by and retrieved into character variables.

Data type conversion occurs automatically for different numeric types such as from floating-point database column values into integer C variables, and for character strings, such as from varying-length Ingres character fields into fixed-length C character string buffers.

Ingres does *not* automatically convert between numeric and character types. You must use the Ingres type conversion operators, the Ingres ascii function, or a C conversion routine for this purpose.

The following table shows the default type compatibility for each Ingres data type:

Ingres Type	C Type
char(N)	char [*N*+1]
varchar(*N*)	char [*N*+1]
char(*N*)(*with embedded nulls*)	varchar
varchar(*N*)(*with embedded nulls*)	varchar
integer1	short
integer2	short
smallint	short
integer	int
integer	long
float4	float
bigint	long (64-bit); long long (32-bit)
float	double
date	char [26]
money	double
table_key	varchar
object_key	varchar
decimal	double
long varchar	char[ ]
long varchar (*with embedded nulls*)	varchar
byte	varbyte
varbyte	varbyte
long byte	varbyte

Runtime Numeric Type Conversion

The Ingres runtime system provides automatic data type conversion between numeric-type values in the database and the forms system and numeric C variables. It follows the standard type conversion rules (according to standard C numeric conversion rules). For example, if you assign a float variable to an integer-valued field, the digits after the decimal point of the variable's value are truncated. Runtime errors are generated for overflow on conversion.

Unsigned integers can be assigned to and retrieved from the database wherever plain integers are used. However, take care when using an unsigned integer whose positive value is large enough to cause the high order bit to be set. Integers such as these are treated as negative numbers in Ingres arithmetic expressions and are displayed as negative numbers by the Forms Runtime system.

The Ingres money type is represented as an 8-byte floating-point value compatible with a C double.

Runtime Character Type Conversion

Automatic conversion occurs between Ingres character string values and C character string variables. The string-valued Ingres objects that can interact with character string variables are:

- Ingres names, such as form and column names
- database columns of type character
- database columns of type varchar
- form fields of type character
- database columns of type long varchar

Several considerations apply when dealing with character string conversions, both to and from Ingres.

References in this section to *character string variables* do not refer to single byte integers declared with the char type, but to the character string pointer:

char ***_character_string_pointer_;**

or to the character string buffer:

char _character_string_buffer_**[**_length_**];**

Character string pointers are always assumed to be pointing at legal string values. Any pointer that has not been initialized to point at a string value causes a runtime error, resulting in program failure or the overwriting of space in memory.

The conversion of C character string variables used to represent Ingres names is simple: trailing blanks are truncated from the variables, because the blanks make no sense in that context. For example, the string literals empform   and empform refer to the same form.

The conversion of other Ingres objects is a bit more complicated. First, the storage of character data in Ingres differs according to whether the medium of storage is a database column of type character, a database column of type varchar, or a character form field. Ingres pads columns of type character with blanks to their declared length. Conversely, it does not add blanks to the data in columns of type varchar or long varchar, or in form fields.

Second, the C convention is to *null terminate* character strings, and the Ingres runtime system assumes that all strings *are* null-terminated. For example, the character string abc is stored as the string literal abc followed by the C null character, \0, requiring four bytes.

Fixed length character string variables cannot contain embedded nulls, because the runtime system cannot differentiate between embedded nulls and the trailing null terminator. For a complete description of variables that contain embedded nulls and the C varchar storage class, see The Varying Length String Type in this chapter.

When retrieving character data from an Ingres database column or form field into a C character string variable, be sure to always supply enough room in the variable to accommodate the maximum size of the particular object, plus one byte for the C null character. (Consider the maximum size to be the length of the database column or the form field.) If the character string buffer is too small to contain the complete string value together with the null character, the runtime system may overwrite other space in memory.

If the length of a character string variable is known to the preprocessor, as in the declaration:

```
char character_string_buffer[fixed_length];
```

then the runtime system copies at most the specified number of characters including the trailing null character. In cases where the fixed length of the variable (less one for the null) is smaller than the data to be copied, the data is truncated. The specified length must be *at least* 2, because one character and the terminating null are retrieved. If the length is exactly 1, the data is overwritten by the terminating null.

Furthermore, take note of the following conventions:

- Data stored in a database column of type character is padded with blanks to the length of the column. The variable receiving such data will contain those blanks, followed by the null character. If the receiving variable was declared with a fixed length known to the preprocessor, such as:

```
char myvar[25]
```

  and the data retrieved is longer than the buffer, the variable will receive only as many characters as will fit (including the terminating null). If the data received is shorter than the variable, the behavior is determined by the setting of the **-**blank_pad preprocessor flag. By default, the terminating null is placed at the end of the retrieved data, without padding out any space remaining in the variable. But if a module is preprocessed with the -blank_pad flag then receiving variables are blank padded to their full defined length (less one space reserved for the terminating null). The -blank_pad behavior is specified by the ANSI SQL92 standard.

- Data stored in a database column of type varchar is not padded with blanks. The character string variable receives only the actual characters in the column, plus the terminating null character.
- Data stored in a character form field contains no trailing blanks. The character string variable receives only the actual characters in the field, plus the terminating null character.

When inserting character data into an Ingres database column or form field from a C variable, the following conventions are in effect:

- When data is inserted from a C variable into a database column of type character and the column is longer than the variable, the column is padded with blanks. If the column is shorter than the variable, the data is truncated to the length of the column.
- When data is inserted from a C variable into a database column of type varchar or long varchar and the column is longer than the variable, no padding of the column takes place. However, all characters in the variable, including trailing blanks, are inserted. Therefore, you may want to truncate any trailing blanks in character string variables before storing them in varchar columns. If the column is shorter than the variable, the data is truncated to the length of the column.
- When data is inserted from a C variable into a character form field and the field is longer than the variable, no padding of the field takes place. In addition, all trailing blanks in the data are truncated before the data is inserted into the field. If the field is shorter than the data (even after all trailing blanks have been truncated), the data is truncated to the length of the field.
- When comparing data in character or varchar database columns with data in a character variable, all trailing blanks are ignored. Initial and embedded blanks are significant.

For a more complete discussion of the significance of blanks in string comparisons, see the *SQL Reference Guid*e.

Caution! As just described, the conversion of character string data between Ingres objects and C variables often involves the trimming or padding of trailing blanks, with resultant change to the data. If trailing blanks have significance in your application, give careful consideration to the effect of any data conversion. Take care not to use the standard strcmp function to test for a change in character data, since blanks are significant.

The Ingres date data type is represented as 25-byte character string. Your program should allow 26 characters to accommodate the C null.

Using Varchar to Receive and Set Character Data

You can also use the C varchar storage class to retrieve and set character data. Typically, varchar variables are used when simple C char variables are not sufficient, as when null bytes are embedded in the character data. In those cases the runtime system cannot differentiate between embedded nulls and the null terminator of the string.

When using varchar variables, the 2-byte length specifier indicates how many bytes are used in the fixed length character array. The runtime system sets this length after a data retrieval, or the program sets it before assigning data to Ingres. This length does *not* include a null terminator, as the null terminator is not copied or included in the data. The runtime system copies, at most, the size of the fixed length data buffer into the variable.

You can also use varchar variables to retrieve character data that does not contain embedded nulls. Here too, the null terminator is not included in the data.

Because varchar variables never include a null terminator, the program should avoid sending the data member of varchar variables to C functions that assume null-terminated strings (such as strlen and strcmp).

The following program fragment demonstrates the use of the varchar storage class for C variables:

```
exec sql begin declare section;
    exec sql declare vch table
        (row  integer,
        data  varchar(10));      /* Note the VARCHAR type */
        static varchar struct vch_ {
            short   vch_length;
            char    vch_data[10];
 } vch_store[3] = {
               /* Statically initialized data with nulls */
          {3, {'1', '2', '3'}},
          {6, {'1', '2', '3', '\0', '5', '6'}},
          {8, {'\0', '2', '3', '4', '\0', '6', '7', '8'}}
};
        varchar struct vch_ vch_res;
        int i, j;
 exec sql end declare section;

exec sql whenever sqlerror call sqlprint;

/*
** Add all three rows of data from table above (including mulls).
** Note that the members of the varchar structure are not mentioned.
*/
for (i = 0; i < 3; i++)
{
     exec sql insert into vch
                        values (:i+1, :vch_store[i]);
 }
/*
** Now SELECT the data back. Note that the runtime system implicitly
** assigns to the length field the size of the data.
*/
exec sql select *
        into :i, :vch_res
        from vch;
 exec sql begin;

     /*
     ** Print the values of each row. Before printing the values,
     ** convert all embedded nulls to the '?' character for printing.
     ** The results are:
     **         [1] '123'
     **         [2] '123?56'
     **         [3] '?234?678'
     */
     for (j = 0; j < vch_res.vch_length; j++)
     {
         if (vch_res.vch_data[j] == '\0')
               vch_res.vch_data[j] = '?';
     }
     printf("[%d] '%.*s'\n", i, vch_res.vch_length,
                                 vch_res.vch_data);
     /*
     ** Note the printf format used here is %.*s rather than %s
     ** because Ingres does not null terminate varchar data.
     */

exec sql end;
```

The SQL Communications Area

This section describes the SQL Communications Area (SQLCA) as implemented in C.

The Include SQLCA Statement

In order to handle SQL database errors, you can issue the include sqlca statement at the outermost scope of your C file. If the file is made up of one main procedure that issues embedded SQL statements, it must be the first embedded SQL statement in the procedure:

```
Emp_Update()
{
    exec sql include sqlca;
    /* Declarations and embedded statements */
}
```

If the file is made up of a few procedures that issue embedded SQL statements, the include sqlca must be issued outside any of the procedures:

```
exec sql include sqlca;
 Emp_Util_1()
{
    /*
    ** Declarations & embedded statements for Emp_Util_1
    */
}
Emp_Util_2()
{
    /*
    ** Declarations & embedded statements for Emp_Util_2
    */
}
```

The include sqlca statement instructs the preprocessor to generate code that includes references to the SQLCA structure for error handling on database statements. It generates a C include directive to a file that defines the SQLCA structure.

You only need to issue the include sqlca statement if you intend to use the SQLCA for error handling. Some error handling mechanism should be included before all executable embedded SQL database statements because the default action is to ignore errors, which is rarely desirable.

Contents of the SQLCA

One of the results of issuing the include sqlca statement is the declaration of the SQLCA structure, which you can use for error handling in the context of database statements. You need to issue the statement only once per source file because it generates an extern structure declaration. The structure declaration for the SQLCA is:

```
typedef struct {
char      sqlcaid[8];
 long      sqlcabc;
 long      sqlcode;
 struct {
   short     sqlerrml;
   char      sqlerrmc[70];
 } sqlerrm;
 char    sqlerrp[8];
 long    sqlerrd[6];
 struct {
   char      sqlwarn0;
   char      sqlwarn1;
   char      sqlwarn2;
   char      sqlwarn3;
   char      sqlwarn4;
   char      sqlwarn5;
   char      sqlwarn6;
   char      sqlwarn7;
 } sqlwarn;
 char     sqlext[8];
 } IISQLCA;

extern IISQLCA sqlca;
```

The nested structure sqlerrm is a varying length character string consisting of the two variables sqlerrml and sqlerrmc described in the *SQL Reference Guide.* For a full description of all the SQLCA structure members, see the *SQL Reference Guide.*

The SQLCA is initialized at load-time. The sqlcaid and sqlcabc fields are initialized to the string SQLCA and the constant 136, respectively.

Note: that the preprocessor is not aware of the structure declaration. Therefore, you cannot use members of the structure in an embedded SQL statement.

For example, the following statement, attempting to insert the string SQLCA into a table, generates an error:

```
exec sql insert into employee (ename)
 /* This statement is illegal */
    values (:sqlca.sqlcaid);
```

Also note that the string-valued fields in the SQLCA are not *null-terminated.* Consequently, if you copy their values into other C variables, you must add the C *null* character afterwards.

All modules linked together share the same SQLCA.

Using the SQLCA for Error Handling

User-defined error, message and dbevent handlers offer the most flexibility for handling errors, database procedure messages, and database events. For more information, see the Advanced Processing section in this chapter.

However, you can do error handling with the SQLCA implicitly by using **whenever** statements, or explicitly by checking the contents of the SQLCA fields sqlcode, sqlerrd, and sqlwarn0.

Error Handling with the Whenever Statement

The syntax of the whenever statement is:

exec sql whenever *condition action***;**

The *condition* is dbevent, sqlwarning, sqlerror, sqlmessage, or not found. The *action* is continue, stop, goto a label, or call a C procedure. For a detailed description of this statement, see the *SQL Reference Guide.*

In C, all labels and procedure names must be legal C identifiers, beginning with an alphabetic character or an underscore. If the label is an embedded SQL reserved word, specify it in quotes. The label targeted by the goto action must be in the scope of all subsequent embedded SQL statements until another whenever statement is encountered for the same action. This is necessary because the preprocessor may generate the C statement:

if (*condition***) goto** *label***;**

after an embedded SQL statement. If the scope of the label is invalid, the C compiler generates an error.

The same scope rules apply to procedure names used with the call action. The reserved procedure sqlprint, which prints errors or database procedure messages and then continues, is always in the scope of the program. When a whenever statement specifies a call as the action, the target procedure is called, and after its execution, control returns to the statement following the statement that caused the procedure to be called. Consequently, after handling the whenever condition in the called procedure, you may want to take some action, instead of merely issuing a C return statement. The C return statement causes the program to continue execution with the statement following the embedded SQL statement that generated the error.

You can also use user-defined handlers for error handling. For more information, see the *SQL Reference Guide*.

The following example demonstrates use of the **whenever** statements in the context of printing some values from the Employee table. The comments do not relate to the program but to the use of error handling:

```
exec sql include sqlca;

Db_Test()

{
    exec sql begin declare section;
         short eno;
         char  ename[21];
         char  age;
    exec sql end declare section;
       exec sql declare empcsr cursor for
         select eno, ename, age
         from employee;
    /*
    ** An error when opening the personnel database will
    ** cause the error to be printed and the program
    ** to abort.
    */
    exec sql whenever sqlerror stop;
    exec sql connect personnel;
    /* Errors from here on will cause the program to
    ** clean up
    */
    exec sql whenever sqlerror call Clean_Up;

    exec sql open empcsr;

    printf("Some values from the \"employee\" table.\n");

    /*
    ** When no more rows are fetched, close the cursor
    */
    exec sql whenever not found goto close_csr;
    /*
    ** The last executable embedded SQL statement was an
    ** OPEN, so we know that the value of "sqlcode"
    ** cannot be SQLERROR or NOT FOUND.
    */
    while (1) /* Loop is broken by NOT FOUND */
    {
        exec sql fetch empcsr
            into :eno, :ename, :age;

            /*
            ** This "printf" does not execute after the
            ** previous FETCH returns the NOT FOUND
            ** condition.
            */
            printf("%d, %s, %d\n", eno, ename, age);
    }

    /*
    ** From this point in the file onwards, ignore all
    ** errors. Also turn off the NOT FOUND condition,
    ** for consistency
    */
    exec sql whenever sqlerror continue;
    exec sql whenever not found continue;
 Close_Csr:
    exec sql close empcsr;
    exec sql disconnect;
 }
```

```
/*
** Clean_Up: Error handling procedure (print error and disconnect).
*/

Clean_Up()
{
    exec sql begin declare section;
        char errmsg[101];
    exec sql end declare section;

    exec sql inquire_sql (:errmsg = ERRORTEXT);
    printf("Aborting because of error:\n%s\n", errmsg);
    exec sql disconnect;

    exit(-1); /* Do not return to Db_Test */
}
```

The Whenever Goto Action In Embedded SQL Blocks

An embedded SQL block-structured statement is delimited by the words begin and end. For example, the select loop and unloadtable loops are all block-structured statements. You can only terminate these statements by the methods specified for the particular statement in the *SQL Reference Guide.* For example, the select loop is terminated either when all the rows in the database result table are processed or by an **endselect** statement. The unloadtable loop is terminated either when all the rows in the forms table field are processed or by an endloop statement.

Therefore, if you use a whenever statement with the goto action in an SQL block, you must avoid going to a label outside the block. Such a goto causes the block to be terminated without issuing the runtime calls necessary to clean up the information that controls the loop. (For the same reason, you must not issue a C return or goto statement that causes control to leave or enter the middle of an SQL block.) The target label of the whenever goto statement should be a label in the block. However, if it is a label for a block of code that cleanly exits the program, the above precaution need not be taken.

The above information does not apply to error handling for database statements issued outside an SQL block, or to explicit hard-coded error handling. See the example of hard-coded error handling in The Table Editor Table Field Application in this chapter.

Explicit Error Handling

The program can also handle errors by inspecting values in the SQLCA structure at various points. For further details, see the *SQL Reference Guide*.

The following example is functionally the same as the previous example, except that the error handling is hard-coded in C statements:

```
exec sql include sqlca;

# define NOT_FOUND 100

Db_Test()
{
    exec sql begin declare section;
        short   eno;
        char    ename[21];
        char    age;
    exec sql end declare section;

    exec sql declare empcsr cursor for
        select eno, ename, age
        from employee;

    /* Exit if database cannot be opened */
    exec sql connect personnel;
    if (sqlca.sqlcode < 0)
    {
        printf("Cannot access database.\n");
        exit(-1);
    }
/* Error if cannot open cursor */
exec sql open empcsr;
 if (sqlca.sqlcode < 0)
    Clean_Up("OPEN \"empcsr\"");

printf("Some values from the \"employee\"
    table.\n");

/*
** The last executable embedded SQL statement was an OPEN, so we know
** that the value of "sqlcode" cannot be SQLERROR or NOT FOUND.
*/
while (sqlca.sqlcode == 0)
/* Loop broken by NOT FOUND */
{

        exec sql fetch empcsr
            into :eno, :ename, :age;

        if (sqlca.sqlcode < 0)
             Clean_Up("FETCH <"empcsr\"");

        /* Do not print the last values twice */
        else if (sqlca.sqlcode != NOT_FOUND)
        printf("%d, %s, %d\n", eno, ename, age);
    }

    exec sql close empcsr;
    exec sql disconnect;

}

/*
** Clean_Up: Error handling procedure
*/

Clean_Up(stmt)
 char    *stmt;
 {
    exec sql begin declare section;
        char *err_stmt = stmt;
        char errmsg[101];
    exec sql end declare section;
```

```
    exec sql inquire_sql (:errmsg = ERRORTEXT);
    printf("Aborting because of error in %s:\n%s\n",
        err_stmt, errmsg);
    exec sql disconnect;

    exit(-1); /* Do not return to Db_Test */
}
```

Determining the Number of Affected Rows

The third element of the SQLCA array sqlerrd indicates how many rows were affected by the last row-affecting statement. This element is referenced by sqlerrd[2] rather than sqlerrd[3] as in other languages, because C subscripts begin at number 0.

The following program fragment, which deletes all employees whose employee numbers are greater than a given number, demonstrates how to use sqlerrd:

```
exec sql include sqlca;
 Delete_Rows(lower_bound)
 int lower_bound;
 {
    exec sql begin declare section;
        int lower_bound_num = lower_bound;
    exec sql end declare section;

    exec sql delete from employee
        where eno > :lower_bound_num;

    /* Print the number of employees deleted */
    printf("%d row(s) were deleted.\n",
     sqlca.sqlerrd[2]);
    }
```

Using the SQLSTATE Variable

You can use the SQLSTATE variable in an embedded SQL for C (ESQL/C) program to return status information about the last SQL statement that was executed. SQLSTATE must be declared in a declaration section and must be in uppercase. Also, it is valid across all sessions, so you only need to declare one SQLSTATE per application.

To declare this variable, use:

```
char SQLSTATE [6];
```

or:

```
char *SQLSTATE;
 /* Where SQLSTATE points to a buffer 6 bytes long. *
```

Dynamic Programming for C

Ingres provides Dynamic SQL and Dynamic FRS to allow you to write generic programs. Dynamic SQL allows a program to build and execute SQL statements at runtime. For example, an application can include an expert mode in which the runtime user can type in select queries and browse the results at the terminal. Dynamic FRS allows a program to interact with any form at runtime. For example, an application can load in any form, allowing the runtime user to retrieve new data from the form and insert it into the database.

The Dynamic SQL and Dynamic FRS statements are described in the *SQL Reference Guide* and the *Forms-based Application Development Tools User Guide,* respectively. This section discusses the C-dependent issues of dynamic programming. For a complete example of using Dynamic SQL to write an SQL Terminal Monitor application, see The SQL Terminal Monitor Application in this chapter. For an example of using both Dynamic SQL and Dynamic FRS to browse and update a database using any form, see A Dynamic SQL/Forms Database Browser in this chapter.

The SQLDA Structure

The SQL Descriptor Area (SQLDA) is used to pass type and size information about an SQL statement, an Ingres form, or a table field, between Ingres and your program.

In order to use the SQLDA, issue the include sqlda statement at the outermost scope of the source file. The include sqlda statement generates a C include directive to a file that defines the SQLDA type. The file does *not* declare an SQLDA variable; the program must declare a variable of the specified type. You can also code this structure directly instead of using the include sqlda statement. You can choose any name for the structure.

The definition of the SQLDA (as specified in the include file) is:

```
# define IISQ_MAX_COLS 1024 /*Maximum number of columns*/
typedef struct sqlvar
{
    /* Single SQLDA variable */
    short sqltype;
    short sqllen;
    char  *sqldata;
    short *sqlind;
    struct {
                short sqlnamel;
                char  sqlnamec[34];
            } sqlname;
 } IISQLVAR;

typedef struct sqda
{
    /* SQLDA structure */
    char      sqldaid[8];
    long      sqldabc;
    short     sqln;
    short     sqld;
    IISQLVAR  sqlvar[IISQ_MAX_COLS];
 } IISQLDA;

/* Structure for Data handlers */

typdef struct sq_datahdlr_ {
  char *sqlarg;             /*optional argument to pass*/
  int  (*sqlhdlr)();/*user-defined datahandler function*/
} IISQLHDR
/* Type codes */
# define IISQ_DTE_TYPE   3  /* Date:Output */
# define IISQ_MNY_TYPE   5  /* Money:Output */
# define IISQ_DEC_TYPE   10 /* Decimal:Output*/
# define IISQ_CHA_TYPE   20 /* Char:Input,Output */
# define IISQ_VCH_TYPE   21 /* Varchar:Input,Output */
# define IISQ_LVCH_TYPE  22 /* LongVarchar:Input,Output*/
# define IISQ_BYTE_TYPE  23 /* Byte:Input,Output*/
# define IISQ_VBYTE_TYPE 24 /* Varbyte:Input,Output*/
# define IISQ_LBYTE_TYPE 25 /* Long Byte:Input,Output*/
# define IISQ_INT_TYPE   30 /* Integer:Input,Output */
# define IISQ_FLT_TYPE   31 /* Float:Input,Output */
# define IISQ_CHR_TYPE   32 /* C - not seen.*/
# define IISQ_TXT_TYPE   37 /* Text - not seen */
# define IISQ_OBJ_TYPE   45 /* 4GL Object: Output */
# define IISQ_HDLR_TYPE  46 /* IISQLHDLR: Datahandler */
# define IISQ_TBL_TYPE   52 /* Table field: Output */
# define IISQ_DTE_LEN    25 /* Date length */
/* Allocation sizes */
# define IISQDA_HEAD_SIZE 16
# define IISQDA_VAR_SIZE  sizeof(IISQLVAR)
```

The actual definition in the included file is a C macro, which you can use to declare your own sized SQLDA. For more detail, see Declaring and Allocating an SQLDA Variable in this chapter.

Structure Definition and Usage Notes:

- The type definition of the SQLDA is called IISQLDA. This is done so that an SQLDA variable can be called SQLDA without causing a compile-time conflict.

- The sqlvar array is a varying length array, which has a default dimension of IISQ_MAX_COLS (1024) elements. The real dimension is determined when the structure is dynamically allocated. Dynamic allocation is described later. If a variable of type IISQLDA is statically declared, then by default the program has a variable of IISQ_MAX_COLS or 1024 sqlvar elements.
- The sqlvar array begins at subscript 0, not at 1.
- If your program defines its own SQLDA type, you must confirm that the structure layout is identical to that of the IISQLDA type, although you can declare a different number of sqlvar elements.
- The nested structure sqlname is a varying length character string consisting of a length and data area. The sqlnamec field contains the name of a result field or column after the describe (or prepare into) statement. The length of the name is specified by sqlnamel. Unlike regular C character data, the characters in the sqlnamec field are *not* null-terminated. You can also set the sqlname structure by a program using Dynamic FRS. (See Setting SQLNAME for Dynamic FRS in this chapter.)
- The list of type codes represent the types that are returned by the describe statement, and the types used by the program when retrieving or setting data using an SQLDA. The type code IISQ_TBL_TYPE indicates a table field, and is set by the FRS when describing a form which contains a table field.
- The allocation sizes are defined so that a program can allocate a sequential block of memory with one SQLDA *head* and any number of SQLDA variables. Dynamic allocation is described later.

Declaring and Allocating an SQLDA Variable

Once the SQLDA definition has been included (or hard coded), the program can declare an SQLDA variable. You must declare this variable outside a declare section, because the preprocessor does not understand the special meaning of the SQLDA. When you use the variable, the preprocessor accepts any object name and assumes that the variable *points* at a legal SQLDA. The actual SQLDA area must be either dynamically allocated or statically declared and pointed at by the variable.

Dynamic Allocation of an SQLDA

In order to dynamically allocate an SQLDA, you must call an allocation routine (such as the C calloc function) and cast the result as a pointer to an SQLDA. The allocation call must include one header (IISQDA_HEAD_SIZE) and any number of variables (N * IISQDA_VAR_SIZE). For example, the following program fragment dynamically allocates an SQLDA with number variables, and points the variable sqlda at the allocated memory. As soon as the SQLDA is allocated, the sqln field is set to record how many array elements were allocated:

```
exec sql include sqlda;

IISQLDA *sqlda;     /* Pointer to an SQLDA */
...

/*
** 'number' has been assigned a positive number.
** Note that the result of the allocation call, calloc,
** is cast to be a pointer to an SQLDA.
** The calloc routine is passed 2 parameters
** (number of objects, and size of a single object).
*/
sqlda = (IISQLDA *)calloc(1, IISQDA_HEAD_SIZE +
       (number * IISQDA_VAR_SIZE));

if (sqlda == (IISQLDA *)0)        /* Memory error */
{
    /* Print error and exit */
    err_exit("Failure allocating %d SQLDA elements\n",
           number);
 }

sqlda->sqln = number;              /* Set the size */
...

exec sql describe s1 into :sqlda;
```

If you change the above allocation call to:

```
sqlda = (IISQLDA *)calloc(1, sizeof(IISQLDA));
```

then IISQ_MAX_COLS elements is allocated. This number of elements is the current maximum for data retrievals. In this case, the sqln field should be set to IISQ_MAX_COLS.

Static Declaration of an SQLDA

As previously mentioned, you can statically declare an SQLDA as well as dynamically allocate one. The C file that is included when issuing the include sqlda statement specifies some C macros that help a program tailor the size of the statically declared SQLDA. In fact, the IISQLDA type definition is derived from that macro.

If a program requires a statically declared SQLDA with the same number of variables as the IISQLDA type, then it can use code like the following:

```
exec sql include sqlda;

IISQLDA    _sqlda;
 IISQLDA    *sqlda = &_sqlda;

sqlda->sqln = IISQ_MAX_COLS; /* Set the size */
...

exec sql describe s1 into :sqlda;
```

Even though a pointer to an SQLDA is required when describing or executing a statement, it is also acceptable to use the syntax:

```
exec sql describe s1 into :&_sqlda;
```

You must confirm that the SQLDA object being used is a pointer to a valid SQLDA.

If a program requires a statically declared SQLDA with a *different* number of variables (not IISQ_MAX_COLS), it can use the macro IISQLDA_TYPE. This macro is described in more detail in the eqsqlda.h include file that is generated by include sqlda. (If you are not familiar with C macros then skip the following discussion). The syntax of IISQLDA_TYPE is:

IISQLDA_TYPE(*tag_name***,** *sqlda_name***,** *number_of_sqlvars*);

IISQLDA_TYPE is a macro that declares object *sqlda_name* (a type definition or a variable) of an SQLDA-like structure with tag *tag_name*, and with *num_of_sqlvars* SQLDA variables. For example, the following declaration declares a local SQLDA, called sqlda10 with 10 variables. The variable sqlda10 is *not* a pointer.

```
IISQLDA_TYPE(da10_, sqlda10, 10);
```

The following example declares a static SQLDA with 32 variables, and a pointer to the SQLDA:

```
static IISQLDA_TYPE(da32_, sqlda32, 32);
 struct da32_ *da32_ptr = &sqlda32;
```

Using the SQLVAR

The *SQL Reference Guide* discusses the legal values of the sqlvar array. The describe and prepare into statements assign type, length, and name information into the SQLDA. This information refers to the result columns of a prepared select statement, the fields of a form, or the columns of a table field. When the program uses the SQLDA to retrieve or set Ingres data, it must assign the type and length information that now refers to the variables being pointed at by the SQLDA.

C Variable Type Codes

The type codes shown in The SQLDA Structure are the types that describe Ingres result fields and columns. For example, the SQL types long varchar, date, decimal and money do not describe a program variable, but rather data types that are compatible with the C types char and double. When these types are returned by the describe statement, the type code must be changed to a compatible C or SQL/C type.

The following table describes the type codes to use with C variables that will be pointed at by the sqldata pointers:

ESQL/C Type Codes (sqltype)	Length (sqllen)	C Variable Type
IISQ_INT_TYPE	1	char
IISQ_INT_TYPE	2	short
IISQ_INT_TYPE	4	int, long
IISQ_FLT_TYPE	4	float
IISQ_FLT_TYPE	8	double
IISQ_CHA_TYPE	LEN	char var[LEN +1]
IISQ_VCH_TYPE	LEN	varchar with data array [LEN]
IISQ_HDLR_TYPE	0	IISQHDLR

One-byte integer data types are specified as a char variable with no specified array dimension. Do not confuse this data type with string data types that are specified as a char variable with a fixed array dimension.

You can specify nullable data types (those variables that are associated with a null indicator) by assigning the negative of the type code to the sqltype field. If the type is negative, a null indicator must be pointed at by using the sqlind field.

Character Data and the SQLDA

As with regular embedded SQL statements, there are special rules for C character data. The describe statement returns IISQ_CHA_TYPE for fixed length character strings (char), IISQ_VCH_TYPE for varying length character strings (varchar),and IISQ_LVCH_TYPE for long strings (long varchar). For example, two columns of type char(5) and varchar(100) return types and lengths IISQ_CHA_TYPE:5 and IISQ_VCH_TYPE:100. The lengths specify the maximum lengths for both columns and do not include the C null terminator.

A column of type long varchar will return IISQ_LVCH_TYPE: 0. The length returned is zero because this character type may be of any size up to 2 gigabytes. Long varchar is an Ingres SQL datatype, so when using the SQLDA to retrieve or set data of a long varchar column into a host variable, IISQ_CHA_TYPE or IISQ_VCH_TYPE must be used. For information on how to specify user-defined data handlers for retrieving or setting large object data through the SQLDA, see Data Handlers and the SQLDA in this chapter.

When using the SQLDA to retrieve character data, the length you supply for fixed length C char variables must include the space for the null terminator. As with normal retrieval of character data, the data is copied (up to the specified length) and a null terminator is then added.

For example, the type specification:

```
/*
** Assume 'sqlda' is a pointer to a dynamically allocated SQLDA
*/

sqlda->sqlvar[0].sqltype = IISQ_CHA_TYPE;
 sqlda->sqlvar[0].sqllen  = 5;
```

assumes that 5 bytes of data can be copied, and that there is one extra byte for the null terminator, such as in the declaration:

```
char buf[6];
```

If there are more than five bytes to copy, the data is truncated at five bytes and the null terminator is put into the sixth byte. If there are less than five bytes to copy, fewer bytes are copied and a null terminator is added. This rule is identical to the normal rule of character retrieval. The specified length must be *at least* 2 because one character and the terminating null are retrieved. If the length is exactly 1, data is overwritten.

If you may be retrieving character data with embedded nulls (such as binary streams of data), then you must use the embedded SQL/C varchar storage class. You can also use varchar variables to retrieve any character data even if there are no embedded nulls. The Dynamic SQL rules for retrieving into varchar variables are the same as the normal retrieval rules: the runtime system sets the 2-byte length field of the varchar data to the amount of data that was copied. The length specified in the sqllen field must be the size of the fixed length data buffer in the varchar variable.

For example, the type specification:

```
sqlda->sqlvar[0].sqltype = IISQ_VCH_TYPE;
 sqlda->sqlvar[0].sqllen  = 100;
```

assumes that up to 100 bytes of data can be copied, such as in the declaration:

```
varchar struct {
     short     len;
     char      buf[100];
 } vch;
```

In the case of varchar, the data is not null-terminated.

You can also use the SQLDA to set Ingres data, as in the statements:

```
exec sql execute statement_name USING DESCRIPTOR

      descriptor_name;

exec frs putform form_name USING DESCRIPTOR
      descriptor_name;
```

When setting character data using pointers to fixed C char data, the data must be null-terminated, and the length specified in sqllen is ignored. It is good programming style to set the length to zero. For example, the type specification:

```
sqlda->sqlvar[0].sqltype = IISQ_CHA_TYPE;
 sqlda->sqlvar[0].sqllen  = 0;
```

can refer to the *any* C string value.

When setting character data using pointers to varchar variables, the sqllen must specify the size of the fixed size data array, and the 2-byte length field must specify the current length of data.

Binary Data and the SQLDA

The describe statement may return any of the three binary types: IISQ_BYTE_TYPE, IISQ_VBYTE_TYPE or IISQ_LBYTE_TYPE. However, only IISQ_BYTE_TYPE AND IISQ_VBYTE_TYPE can be used when actually sending and retrieving data. The long byte data type must be changed to byte or varbyte if it is less than 32K, or else replaced by a data handler reference type.

Pointing at C Variables

In order to fill an element of the sqlvar array, you must set the type information and assign a valid address to sqldata. The address can be that of a dynamically allocated data area or a legal variable address. The address should always be cast to a pointer to a character (char *), as that is the base type of the sqldata field.

For example, the following fragment sets the type information and points at a dynamically allocated 4-byte integer and an 8-byte nullable floating-point variable:

```
/* Assume sqlda is a pointer to a dynamically allocated SQLDA */

sqlda->sqlvar[0].sqltype  = IISQ_INT_TYPE;
sqlda->sqlvar[0].sqllen   = sizeof(long);
sqlda->sqlvar[0].sqldata  = (char *)calloc(1,
       sizeof(long));
sqlda->sqlvar[0].sqlind   = (short *)0;

sqlda->sqlvar[1].sqltype  = -IISQ_FLT_TYPE;
sqlda->sqlvar[1].sqllen   = sizeof(double);
sqlda->sqlvar[1].sqldata  = (char *)calloc(1,
       sizeof(double));
sqlda->sqlvar[1].sqlind   = (short *)calloc(1,
       sizeof(short));
```

You can replace the three calls to the calloc allocation routine by references to program variables, such as:

```
...
sqlda->sqlvar[0].sqldata = (char *)&long_var;
...
sqlda->sqlvar[1].sqldata = (char *)&double_var;
sqlda->sqlvar[1].sqlind  = (short *)&short_var;
```

Of course, in the latter case, it is appropriate to maintain a pool of available variables to use, such as arrays of differently typed variables.

When pointing at character data, you should allocate sqllen bytes plus one for the null, as in:

```
/* Assume 'sqltype' and 'sqllen' are set by DESCRIBE */
sqlda->sqlvar[0].sqltype  = IISQ_CHA_TYPE;
sqlda->sqlvar[0].sqllen   = some length;
sqlda->sqlvar[0].sqldata  =       (char*)calloc(1,sqlda-sqlvar[0].sqllen + 1);
```

When pointing at varchar data, you should allocate sqllen bytes plus two (or sizeof(short)) for the 2-byte length field. For example:

```
sqlda->sqlvar[0].sqltype  = IISQ_VCH_TYPE;
sqlda->sqlvar[0].sqllen   = 50;
sqlda->sqlvar[0].sqldata  = (char *)calloc(1,
    sizeof(short) + 50);
```

You may also set the SQLVAR to point to a data handler for large object columns. For details, see Advanced Processing in this chapter.

Setting SQLNAME for Dynamic FRS

Using the sqlvar with Dynamic FRS statements requires a few extra steps. These extra steps relate to the differences between Dynamic FRS and Dynamic SQL and are described in the *Forms-based Application Development Tools User Guide* and *SQL Reference Guide* respectively.

When using the SQLDA in a forms input or output using clause, the value of sqlname must be set to a valid field or column name. If a previous describe statement set the name, the program must retain or reset it. If the name refers to a hidden column in a table field, the program must set it directly. If your program sets sqlname directly, it must also set sqlnamel and sqlnamec. You do not need to pad the name portion with blanks or null-terminate it.

For example, a dynamically named table field has been described, and the application always initializes any table field with a hidden 6-byte character column called rowid.

The code used to retrieve a row from the table field including the hidden column and _state variable has to construct the two named columns:

```
...

char  rowid[6+1];
 int   rowstate;

...

exec frs describe table :formname :tablename
     into :sqlda;

...

/* C is zero-based so save before incrementing */
col_num = sqlda-sqld++;

/* Set up to retrieve rowid */
sqlda->sqlvar[col_num].sqltype = IISQ_CHA_TYPE;
sqlda->sqlvar[col_num].sqllen  = 6;
sqlda->sqlvar[col_num].sqldata = rowid;
sqlda->sqlvar[col_num].sqlind  = (short *)0;
sqlda->sqlvar[col_num].sqlname.sqlnamel = 5;
strcpy(sqlda->sqlvar[col_num].sqlname.sqlnamec,
    "rowid");

col_num = sqlda-sqld++;

/* Set up to retrieve _STATE */
sqlda->sqlvar[col_num].sqltype = IISQ_INT_TYPE;
sqlda->sqlvar[col_num].sqllen  = sizeof(int);
sqlda->sqlvar[col_num].sqldata = &rowstate;
sqlda->sqlvar[col_num].sqlind  = (short *)0;
sqlda->sqlvar[col_num].sqlname.sqlnamel = 6;
strcpy(sqlda-sqlvar[col_num].sqlname.sqlnamec,
    "_state");
...

exec frs getrow :formname :tablename using descriptor :sqlda;
```

Advanced Processing

This section describes user-defined handlers. It includes information about user-defined error, dbevent, and message handlers as well as data handlers for large objects.

User-Defined Error, DBevent, and Message Handlers

You can use user-defined handlers to capture errors, messages, or events during the processing of a database statement. Use these handlers instead of the sql **whenever** statements with the SQLCA when you want to do the following:

- Capture more than one error message on a single database statement.
- Capture more than one message from database procedures fired by rules.
- Trap errors, events, and messages as the DBMS raises them. If an event is raised when an error occurs during query execution, the WHENEVER mechanism detects only the error and defers acting on the event until the next database statement is executed.

User-defined handlers offer you flexibility. If, for example, you want to trap an error, you can code a user-defined handler to issue an inquire_sql to get the error number and error text of the current error. You can then switch sessions and log the error to a table in another session; however, you must switch back to the session from which the handler was called before returning from the handler. When the user handler returns, the original statement continues executing. User code in the handler cannot issue database statements for the session from which the handler was called.

The handler must be declared to return an integer. However, the preprocessor ignores the return value.

Syntax Notes:

Use the following syntax to specify the three types of handlers:

```
exec sql set_sql (errorhandler    = error_routine|0);
exec sql set_sql (dbeventhandler  = event_routine|0);
exec sql set_sql (messagehandler  = message_routine|0);
```

- The errorhandler, dbeventhandler, and messagehandler denote a user-defined handler to capture errors, events, and database messages respectively, as follows:
  - error_routine is the name of the function the Ingres runtime system calls when an error occurs.
  - event_routine is the name of the function the Ingres runtime system calls when an event is raised.
  - message_routine is the name of the function the Ingres runtime system calls whenever a database procedure generates a message.
- Errors that occur in the error handler itself do not cause the error handler to be re-invoked. You must use inquire_sql to handle or trap any errors that may occur in the handler.

- Unlike regular variables, you must not declare the handler in an ESQL declare section; therefore, do not use a colon before the handler argument. (However, you must declare the handler to the compiler.)
- If you specify a zero (0) instead of a name, the zero will unset the handler.

User-defined handlers are also described in the *SQL Reference Guide.*

Declaring and Defining User-Defined Handlers

The following example shows how to declare a handler for use in the set_sql errorhandler statement for ESQL/C:

```
exec sql include sqlca;

main()
{
      int error_func();
       exec sql connect dbname;
       exec sql set_sql (errorhandler = error_func);
      . .
}

int
error_func()
{
      exec sql begin declare section;
            int errnum;
      exec sql end declare section;

      exec sql inquire_sql (:errnum = ERRORNO);
      printf ("Error number is %d", errnum);
      return 0;
 }
```

If you are using ANSI C function prototypes, declare the handler function prototype as follows:

```
int error_funct(void);
```

where the handler is defined as follows:

```
int error_funct(void)
 {
        ...
 }
```

User-Defined Data Handlers for Large Objects

You can use user-defined data handlers to transmit large object column values to or from the database a segment at a time. For more details on large objects, the data handler clause, the get data statement and the put data statement, see the *SQL Reference Guide* and the *Forms-based Application Development Tools User Guide*.

ESQL/C Usage Notes

When using ESQL/C, the following notes apply:

- The data handler, and the data handler argument, should not be declared in an ESQL declare section. Therefore do not use a colon before the data handler or its argument.
- You must ensure that the data handler argument is a valid C pointer. ESQL will not do any syntax or datatype checking of the argument.
- The data handler must be declared to return an integer. However, the actual return value will be ignored.

Data Handlers and the SQLDA

You may specify a user-defined data handler as an SQLVAR element of the SQLDA, to transmit large objects to/from the database. The eqsqlda.h file included via the include sqlda statement defines an IISQLHDLR type that may be used to specify a data handler and its argument. It is defined:

```
typedef struct sq_datahdlr
   {
       char *sqlarg;    /* optional argument to pass */
       int  (*sqlhdlr)();  /* user-defined datahandler */
    } IISQLHDLR;
```

The file does not declare an IISQLHDLR variable; the program must declare a variable of the specified type and set the values:

```
IISQLHDLR    data_handler;
char         *arg;
int          Get_Handler();
data_handler.sqlarg = arg;
data_handler.sqlhdlr = Get_Handler;
```

The sqltype, sqllen and sqldata fields of the SQLVAR element of the SQLDA should then be set as follows:

```
/*
** assume sqlda is a pointer to a dynamically allocated ** SQLDA
*/
sqlda->sqlvar[i].sqltype = IISQ_HDLR_TYPE;
sqlda->sqlvar[i].sqllen  = 0;
sqlda->sqlvar[i].sqldata = (char*)&data_handler;
```

To indicate nullability for a column set sqltype to negative IISQ_HDLR_TYPE, as shown in the following code fragment:

```
sqlda->sqlvar[i].sqltype = -IISQ_HDLR_TYPE;
```

Sample Programs

The programs in this section are examples of how to declare and use user-defined data handlers in an ESQL/C program. There are examples of a handler program, a put handler program, a get handler program and a dynamic SQL handler program.

If you precompile these examples using the **-**prototypes flag (for ANSI C style function declarations), you must declare the functions using a generic pointer argument. For example:

```
int Put_Handler(void *hdlr_arg)
```

Handler Program

This example assumes that the book table was created with the statement:

```
exec sql create table book (chapter_num integer,
    chapter_name char(50), chapter_text long
         varchar);
```

This program inserts a row into the book table using the data handler Put_Handler to transmit the value of column chapter_text from a text file to the database in segments. Then it selects the column chapter_text from the table book using the data handler Get_Handler to retrieve the chapter_text column a segment at a time.

```
/*
** For this example the argument to the datahandlers
** will be a pointer to a HDLR_PARAM structure.
*/

typedef struct hdlr_arg_struct
{
     char *arg_str;
     int   arg_int;
 } HDLR_PARAM;

main()
{

/* Do not declare the datahandlers or the datahandler argument to the ESQL
** preprocessor. The argument passed to a datahandler must be a pointer.
*/
         int Put_Handler();
         int Get_Handler();

    HDLR_PARAM hdlr_arg;

    /*
    ** The indicator variable must be declared to ESQL.
    */
         exec sql begin declare section;
                 short indvar;
                 int   chapter_num;
         exec sql end declare section;

    /*
    ** Insert a long varchar value chapter_text into the table book
    ** using the datahandler Put_Handler. The argument passed to the
    ** datahandler is the address of structure hdlr_arg.
    */

    . . .

    . . .

    exec sql insert into book (chapter_num,

        chapter_name,
             chapter_text)
             values (5,'One dark and stormy night',
       datahandler(Put_Handler(&hdlr_arg)));

. . .

    /*
    ** Select the column chapter_num and the long
    ** varchar column chapter_text from the table
    ** book. The Datahandler (Get_Handler) will be
    ** invoked for each non null value of the column
    ** chapter_text retrieved. For null values the
    ** indicator variable will be set to "-1" and the
    ** datahandler will not be called
    */
. . .

. . .

exec sql select chapter_num, chapter_text into
   :chapter_num,
      datahandler(Get_Handler(&hdlr_arg)):indvar
           from book;
```

```
 exec sql begin;
      process row...
 exec sql end;

    . . .

}
```

Put Handler

This example shows how to read the long varchar chapter_text from a text file and insert it into the database a segment at a time:

```
Int
Put_Handler(hdlr_arg)
 HDLR_PARAM *hdlr_arg;
 {

    /*
    ** Host variables in the put data statement must
    ** be declared to the ESQL preprocessor
    */

    exec sql begin declare section;
        char seg_buf[1000];
        int  seg_len;
        int  data_end;
    exec sql declare section;

    int more_data;

    open file...

    data_end = 0;
    more_data = 1;

    while (more_data == 1)
    {
    read segment of less than 1000 chars from
    file into seg_buf...
    if (end_of_file)
    {
        data_end = 1;
        more_data = 0;

     }

    seg_len = number_of_bytes_read;

    exec sql put data (segment       = :seg_buf,
                       segmentlength = :seg_len,
                       dataend       = :data_end);
    };

    . .
    close file...
    set hdlr_arg fields to return appropriate
        values...
    . . .
 }
```

Get Handler

This example shows how to get the long varchar chapter_text from the database and write it to a text file:

```
Get_Handler(hdlr_arg)
 HDLR_PARAM *hdlr_arg;
 {

    /* Host variables in the get data statement must
    ** be declared to the ESQL preprocessor
    */

    exec sql begin declare section;
        char seg_buf[2000];
        int  seg_len;
        int  data_end;
        int  max_len;
    exec sql end declare section;
    . . .

    process information passed in via the
         hdlr_arg...
    open file...


    /* Get a maximum segment length of 2000 bytes. */

    max_len = 2000;
    data_end = 0;
    while (data_end == 0)
    {
    /*
    ** segmentlength: will contain the length of the
    ** segment retrieved
    ** seg_buf: will contain a segment of the column
    ** chapter_text
    ** data_end: will be set to '1' when the entire
    ** value in chapter_text has been retrieved
    */

    exec sql get data (:seg_buf  = segment,
                       :seg_len  = segmentlength,
                       :data_end = dataend)
                        with maxlength = :max_len;

    write segment to a file...
    }
    . . .
    set hdlr_arg fields to return appropriate
         values...
    . .

}
```

Dynamic SQL Handler Program

The following is an example of a dynamic SQL handler program. This program fragment shows the declaration and usage of a data handler in a dynamic SQL program, using the SQLDA. It uses the data handler Get_Handler() and the HDLR_PARAM structure described in the previous example.

```
main()
{

    exec sql include sqlda;

    /* Declare the SQLDA and IISQLHDLR structures */
    IISQLDA    _sqlda;
    IISQLDA    *sqlda = &_sqlda;
    IISQLHDLR datahdlr_struct;

    /*
    ** Do not declare the datahandlers or the
    ** datahandler argument to the ESQL preprocessor
    */

    int Get_Handler();
    HDLR_PARAM hdlr_arg;
    int base_type;
    int col_num;

    /* Declare null indicator to ESQL */

    exec sql begin declare section;
        short indvar;
        char stmt_buf[100];
    exec sql end declare section;

    . . .

    /*
    ** Set the IISQLHDLR structure with the appropriate
    ** datahandler and datahandler argument.
    */

    datahdlr_struct.sqlarg = &hdlr_arg;
    datahdlr_struct.sqlhdlr = Get_Handler;
    sqlda->sqln = IISQ_MAX_COLS;
    /* Describe the statement into the SQLDA */

    strcpy(stmt_buf, "select * from book");
    exec sql prepare stmt from :stmt_buf;
    exec sql describe stmt into sqlda;

    . . .

    /*
    ** Determine the base_type of the SQLDATA
    ** variables
    */

    for (col_num = 0; col_num < sqlda->sqln;
         col_num++)
    {
        base_type = abs(sqlda-
            >sqlvar[col_num].sqltype);

        /*
        ** Set the sqltype, sqldata and sqlind for
        ** each column. The long varchar column
        ** chapter text will be set to use a
        ** datahandler
        */

        if (base_type == IISQ_LVCH_TYPE)
        {
           sqlda->sqlvar[col_num].sqltype =-IISQ_HDLR_TYPE
```

```
        sqldata->sqlvar[col_num].sqldata =
                        (char *)&datahdlr_struct;
        sqlda->sqlvar[col_num].sqlind = &indvar;
    }
    else
        . . .

}
/*
** The datahandler (Get_Handler) will be
** invoked for each non null value of column
** chapter_text retrieved. For null values
** the indicator variable will be set to
** "-1" and the datahandler will not be called
*/

. .

exec sql execute immediate :stmt_buf using :sqlda;
exec sql begin;
    process rows...
exec sql end;

. . .
```

Preprocessor Operation

This section describes the embedded SQL preprocessor for C and the steps required to create, compile, and link an embedded SQL program.

Include File Processing

The embedded SQL include statement provides a means to include external files in your program's source code. Its syntax is:

exec sql include *filename***;**

where *filename* is a quoted string constant specifying a file name, or a system environment variable (UNIX) or a logical name (VMS) that points to the file name. If you do not give an extension to the filename (or to the file name pointed at by the environment variable), the default C input file extension .sc is assumed.

This statement is normally used to include variable declarations, although it is not restricted to such use. For more details on the include statement, see the *SQL Reference Guide.*

The included file is preprocessed and an output file with the same name but with the default output extension .c is generated. You can override this default output extension with the -o.*ext* flag on the command line. The reference in the original source file to the included file is translated in the output file to the specified include output file. If the -o flag is used (without an extension), then the output file is not generated for the include statement. This is useful for program libraries that are using make or VMS dependencies.

If you use both the -o.*ext* and the -o flags, then the preprocessor generates the specified extension for the translated include statements in the program but does not generate new output files for the statements.

For example, assuming that no overriding output extension is explicitly given on the command line. The embedded SQL statement:

```
exec sql include 'employee.dcl';
```

is preprocessed to the C statement:

```
# include "employee.c"
```

The employee.dcl file is translated into the C file employee.c.

As another example, assume that a source file called inputfile contains the following include statement:

```
exec sql include 'mydecls';
```

Windows

The name MYDECLS can be defined as a system environment variable pointing to the file c/dev/headers/myvars.sc by means of the following command at the system level:

```
setenv MYDECLS="c:\dev\headers\myvars"
```

Because the extension .sc is the default input extension for embedded SQL include files, you do not need to specify it when defining an environment variable for the file.

Assume now that inputfile is preprocessed with the command:

```
esqlc -o.h inputfile
```

The command line specifies .h as the output file extension for include files. As the file is preprocessed, the include statement shown earlier is translated into the C statement:

```
# include "c:\dev\headers\myvars.h"
```

The C file c:\dev\headers\myvars.h is generated as output for the original include file, c:\dev\headers\myvars.sc.

You can also specify include files with a relative path. For example, if you preprocess the file c:\dev\mysource\myfile.sc. the embedded SQL statement:

```
exec sql include '..\headers\myvars.sc';
```

is preprocessed to the C statement:

```
# include "..\headers\myvars.c"
```

The C file c:\dev\headers\myvars.c is generated as output for the original include file, c:\dev\headers\myvars.sc.

UNIX

The name MYDECLS can be defined as a system environment variable pointing to the file /dev/headers/myvars.sc by means of the following command at the system level:

```
setenv MYDECLS "/dev/headers/myvars"
```

Because the extension .sc is the default input extension for embedded SQL include files, you do not need to specify it when defining an environment variable for the file.

Assume now that inputfile is preprocessed with the command:

```
esqlc -o.h inputfile
```

The command line specifies .h as the output file extension for include files. As the file is preprocessed, the include statement shown earlier is translated into the C statement:

```
# include "/dev/headers/myvars.h"
```

The C file /dev/headers/myvars.h is generated as output for the original include file, /dev/headers/myvars.sc.

You can also specify include files with a relative path. For example, if you preprocess the file /dev/mysource/myfile.sc. the embedded SQL statement:

```
exec sql include '../headers/myvars.sc';
```

is preprocessed to the C statement:

```
# include "../headers/myvars.c"
```

The C file /dev/headers/myvars.c is generated as output for the original include file, /dev/headers/myvars.sc.

VMS

The name mydecls can be defined as a system logical name pointing to the file dra1:[headers]myvars.sc by means of the following command at the system level:

```
define mydecls dra1:[headers]myvars
```

Because the extension .sc is the default input extension for embedded SQL include files, it need not be specified when defining a logical name for the file.

Assume now that inputfile is preprocessed with the command:

```
esqlc -o.h inputfile
```

The command line specifies .h as the output file extension for include files. As the file is preprocessed, the include statement shown earlier is translated into the C statement:

```
# include "dra1:[headers]myvars.h"
```

The C file dra1:[headers]myvars.h is generated as output for the original include file, dra1:[headers]myvars.sc.

You can also specify include files with a relative path. For example, if you preprocess the file dra1:[mysource]myfile.sc, the embedded SQL statement:

```
exec sql include '[-.headers]myvars.sc'
```

is preprocessed to the C statement:

```
# include "[-.headers]myvars.c"
```

The C file dra1:[headers]myvars.c is generated as output for the original include file, dra1:[headers]myvars.sc.

Including Source Code with Labels

Some embedded SQL statements generate labels in the output code. If you include a file containing such statements, you must be careful to include the file only once in a given C scope. Otherwise, you may find that the compiler later complains that the generated labels are defined more than once in that scope.

The statements that generate labels are the select statement and all the embedded SQL/Forms block-type statements, such as display and unloadtable.

Coding Requirements for Writing Embedded SQL Programs

The following sections describe embedded SQL coding requirements.

Comments Embedded in C Output

Each embedded SQL statement generates one comment and few lines of C code. You may find that the preprocessor translates 50 lines of embedded SQL into 200 lines of C. This can confuse you if you are trying to debug the original source code. To facilitate debugging, a comment corresponding to the original embedded SQL source precedes each group of C statements associated with a particular statement. (Note that a comment precedes only executable embedded SQL statements.) Each comment is one line long and informs the reader of the file name line number and the type of statement in the original sources file. The **-#** flag to the esqlc command makes the C comment a C compiler directive, causing any error messages generated by the C compiler to refer to the original file and line number; this can be useful in some cases.

One consequence of the generated comment is that you cannot comment out embedded statements by putting the opening comment delimiter on an earlier line. You have to put the delimiter on the same line, before the exec word, to cause the preprocessor to treat the complete statement as a C comment.

Embedding Statements Inside C If Blocks

As mentioned above, the preprocessor can produce several C statements for a single embedded SQL statement. However, all of the statements that the preprocessor generates are delimited by left and right braces, composing a C block. Thus the statement:

```
if (!dba)
    exec sql select passwd
        into :passwd
        from security
        where usrname = :userid;
```

produces legal C code, even though the SQL select statement produces more than one C statement. However, two or more embedded SQL statements generate multiple C blocks, so you must delimit them yourself, just as you delimit two C statements in a single if block.

For example:

```
if (!dba)
 {
    exec frs message 'Confirming your user id';
    exec sql select passwd
        into :passwd
        from security
        where usrname = :userid;
 }
```

VMS

Because the preprocessor generates a C block for every embedded SQL statement, the VAX C compiler may generate the Internal Table Overflow error when a single procedure has a very large number of embedded SQL statements and local variables. You can correct this problem by splitting the file or procedure into smaller components.

Embedded SQL Statements that Do Not Generate Code

The following embedded SQL declarative statements do not generate any C code:

declare cursor

declare table

declare statement

whenever

These statements must not contain labels and must not be coded as the only statements in C constructs that do not allow *null* statements. For example, coding a declare cursor statement as the only statement in a C if statement not bounded by left and right braces causes compiler errors:

```
if (using_database)
      exec sql declare empcsr cursor for
                   select ename from employee;
 else
        printf("You have not accessed the database.\n");
```

The preprocessor generates the code:

```
if (using_database)
 else
      printf("You have not accessed the database.\n");
```

This is an illegal use of the C **else** clause.

Command Line Operations

The following sections describe commands that you can use to turn your embedded SQL/C source program into an executable program. These commands preprocess, compile, and link your program.

The Embedded SQL Preprocessor Command

The following command line invokes the C preprocessor:

esqlc {*flags*} {*filename*}

where *flags* are those shown in the following table:

Flag	Description
-blank_pad	Informs the preprocessor to generate code that complies with ANSI and ISO Entry SQL92 data retrieval rules for fixed length char variables. At runtime, data selected into fixed length char host variables will be padded with blanks up to the declared length of the variable less one byte for the C null terminator.
-noblank_pad	Informs the preprocessor to generate code that complies with current Ingres data retrieval rules. At runtime, data selected into fixed length char host variables will not be blank-padded, it will be null terminated to the length of the data retrieved. The default is -noblank_pad
-check_eo	Causes ESQL/C applications to check fixed length host string variables for an end of string null terminator. If one is not found an error condition is raised. This feature is provided for ISO Entry SQL92 conformity.
-nocheckeos	Turns off the above checking. This option is the default.
-d	Adds debugging information to the runtime database error messages generated by embedded SQL. The source file name, line number and the erroneous statement itself are printed along with the error message.
-f[*filename*]	Writes preprocessor output to the named file. If you do not specify *filename*, the output is sent to standard output, one screen at a time.
-i*N*	Sets integer size to *N* bytes. *N* is 1, 2, or 4. The default is 4.

Flag	Description
-l	Writes preprocessor error messages to the preprocessor's listing file, as well as to the terminal. The listing file includes preprocessor error messages and your source text in a file named *filename*.lis, where *filename* is the name of the input file.
-lo	Like -l, but the generated C code also appears in the listing file.
-o	Directs the preprocessor not to generate output files for include files. This flag does not affect the translated include statements in the main program. The preprocessor generates a default extension for the translated include file statements unless you use the **-**o.*ext* flag.
-o. *ext*	Specifies the extension the preprocessor gives to both the translated include statements in the main program and the generated output files. If you do not specify this flag, the default extension is .c. If you use this flag in combination with the **-**o flag, then the preprocessor generates the specified extension for the translated include statements but does not generate output files for the include statements.
-prototypes	Directs the preprocessor to include a header file containing ANSI style function prototypes for the Ingres runtime routines. The default is **-**noprototypes (the prototypes in the header file are not ANSI style)
-s	Reads input from standard input and generates C code to standard output. This is useful for unfamiliar testing statements. If you specify the **-**l option with this flag, the listing file is called stdin.lis. To terminate the interactive session, type Ctrl + D (UNIX) or Ctrl + Z (VMS).
-sqlcode	Indicates the file declares an integer variable named SQLCODE to receive status information from SQL statements. That declaration need not be in an exec sql begin/end declare section. This feature is provided for ISO Entry SQL92 conformity. However, the ISO 92 specification describes SQLCODE as a deprecated feature and recommends using the SQLSTATE variable.
-nosqlcode	Tells the preprocessor not to assume the existence of a status variable named SQLCODE. The default is **-**nosqlcode.
-w	Prints warning messages.

Flag	Description
-wopen	This flag is identical to -wsql=open. However, -wopen is supported only for backwards capability. For more information, see -wsql=open.
-#|-p	Generates # line directive to the C compiler (by default, they are in comments). This flag can prove helpful when debugging the error messages from the C compiler.
-wsql=entry_SQL92	Causes the preprocessor to flag any usage of syntax or features that do not conform to the ISO Entry SQL92 entry level standard. (This is also known as the FIPS flagger option.)
-wsql=open	Use open only with OpenSQL syntax. -wsql = open generates a warning if the preprocessor encounters an embedded SQL statement that does not conform to OpenSQL syntax. (For OpenSQL syntax, see the *OpenSQL Reference Guide*.) This flag is useful if you intend to port an application across different Enterprise Access products. The warnings do not affect the generated code and the output file may be compiled. This flag does not validate the statement syntax for any Enterprise Access product whose syntax is more restrictive than that of OpenSQL.
Windows -?	Shows the command line options for esqlc.
UNIX --	Shows the command line options for esqlc.
VMS -?	Shows the command line options for esqlc.

The embedded SQL/C preprocessor assumes that input files are named with the extension.sc. You can override this default by specifying the file extension of the input file(s) on the command line. The output of the preprocessor is a file of generated C statements with the same name and the extension.c.

If you enter the command without specifying any flags or a filename, a list of flags available for the command are displayed.

The following table presents the options available with esqlc:

Command	Comment
esqlc file1	Preprocesses file1.sc to file1.c
esqlc -l file2.xc	Preprocesses file2.xc to file2.c and creates listing file2.lis
esqlc -s	Accepts input from standard input
esqlc -ffile3.out file3	Preprocesses file3.sc to file3.out
esqlc	Displays a list of flags available for this command

The C Compiler

The preprocessor generates C code. You can then use the cc command to compile this code.

Windows

The preprocessor generates C code. You can then use the cl command to compile this code. You can use all of your compiler options.

For example, to pre-process and compile the file "test1" with the cl compiler, issue the following command:

```
esqlc test1.sc
cl -c test1.c
```

UNIX

You can use all of the cc command line options.

The following example preprocesses and compiles the file test1:

```
esqlc test1.sc
cc -c test1.c
```

VMS

Most of the cc command line options can be used. You should not use the g_float qualifier (to the VAX C compiler) if floating-point values in the file are interacting with Ingres floating-point objects.

As of Ingres II 2.0/0011 (axm.vms/00) Ingres uses member alignment and IEEE floating-point formats. Embedded programs must be compiled with member alignment turned on. In addition, embedded programs accessing floating-point data (including the MONEY data type) must be compiled to recognize IEEE floating-point formats.

The following example preprocesses and compiles the file test1. Note that both the embedded SQL preprocessor and the C compiler assume the default extensions.

```
esqlc test1
cc/list test1
```

Note: For any operating system specific information on compiling and linking ESQL/C programs, see the Readme file.

Linking Embedded SQL Programs—Windows

Windows

Embedded SQL programs require procedures from an Ingres library. The required library is listed below and must be included in your compile or link command after all user modules. You must specify the library in the order shown in the following examples.

Programs Without Embedded Forms

The following example demonstrates the link command of an embedded SQL program called dbentry that was preprocessed and compiled.

```
link -out:dbentry.exe dbentry.obj ^
%II_SYSTEM%\ingres\lib\libingres.lib
```

Compiling and Linking Precompiled Forms

In order to use such a precompiled form in your program, you must follow the steps described here.

In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a file in your directory describing the form in C. VIFRED lets you select the name for the file. After creating the C file this way, you can compile it into linkable object code.

For example, if you were to use the cl compiler:

cl -c *filename*

The output of this command is a file with the extension .obj. You then link this object file with your program by listing it in the link command, as in the following example, which includes the compiled form empform.obj:

```
link -out:formentry.exe formentry.obj empform.obj ^
     %II_SYSTEM%\ingres\lib\libingres.lib
```

Linking Embedded SQL Programs—UNIX

UNIX

Embedded SQL programs require procedures from an Ingres library. The required library is listed below and must be included in your compile or link command after all user modules. You must specify the library in the order shown in the following examples.

Programs Without Embedded Forms

The following example demonstrates the link command of an embedded SQL program called dbentry that was preprocessed and compiled.

```
cc -o dbentry dbentry.o\
 $II_SYSTEM/ingres/lib/libingres.a \
-lm -lc
```

You must include both the math library and the C runtime library.

Ingres shared libraries are available on some Unix platforms**.** To link with these shared libraries replace libingres.a in your link command with:

```
-L $II_SYSTEM/ingres/lib -linterp.1 -lframe.1 -lq.1 \
     -lcompat.1
```

To verify if your release supports shared libraries check for the existence of any of these four shared libraries in the $II_SYSTEM/ingres/lib directory**.** For example:

```
ls -l $II_SYSTEM/ingres/lib/libq.1.*
```

Compiling and Linking Precompiled Forms

The technique of declaring a precompiled form to the FRS is discussed in the *Forms-based Application Development Tools User Guide*. To use such a form in your program, you must also follow the steps described here.

In VIFRED, you can select a menu item to compile a form*.* When you do this, VIFRED creates a file in your directory describing the form in C*.* VIFRED lets you select the name for the file*.* After creating the C file this way, you can compile it into linkable object code with the cc command:

cc -c *filename*

The output of this command is a file with the extension .o*.*

You then link this object file with your program by listing it in the link command, as in the following example, which includes the compiled form empform.o:

```
cc -o formentry formentry.o \
    empform.o \
     $II_SYSTEM/ingres/lib/libingres.a \
     -lm -lc
```

Linking Embedded SQL Programs—VMS

VMS

Embedded SQL programs require procedures from several VMS shared libraries in order to run properly. Once you have preprocessed and compiled an embedded SQL program, you can link it. Assuming the object file for your program is called dbentry, use the following link command:

```
$link dbentry.obj,-
 ii_system:[ingres.files]esql.opt/opt,-
 sys$library:vaxcrtl.olb/library
```

The last line in the link command shown above serves to link the C runtime library for certain basic C functions, such as printf. You need to include this line only if you use those functions in your program.

Assembling and Linking Precompiled Forms

The technique of declaring a precompiled form to the FRS is discussed in the *Forms-based Application Development Tools User Guide*. To use such a form in your program, you must also follow the steps described here.

In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED lets you select the name for the file. Once you have created the MACRO file this way, you can assemble it into linkable object code with the VMS command:

macro *filename*

The output of this command is a file with the extension .obj. You then link this object file with your program by listing it in the link command, as in the following example:

```
$link formentry,-
 empform.obj,-
 ii_system:[ingres.files]esql.opt/opt,-
 sys$library:vaxcrtl.olb/library
```

Linking an Embedded SQL Program Without Shared Libraries

While the use of shared libraries in linking embedded SQL programs is recommended for optimal performance and ease of maintenance, non-shared versions of the libraries have been included in case you require them. Non-shared libraries required by embedded SQL are listed in the esql.noshare options file. The options file must be included in your link command *after* all user modules. Libraries must be specified in the order given in the options file.

The following example demonstrates the link command of an embedded SQL program called dbentry that has been preprocessed and compiled:

```
$link dbentry,-
 ii_system:[ingres.files]esql.noshare/opt
```

Placing User-Written Embedded SQL Routines in Shareable Images

When you plan to place your code in a shareable image, note the following about the psect attributes of your global or external variables:

- As a default, some compilers mark global variables as shared (SHR: every user who runs a program linked to the shareable image sees the same variable) and others mark them as not shared (NOSHR: every user who runs a program linked to the shareable image gets a private copy of the variable).
- Some compilers support modifiers you can place in your source code variable declaration statements to explicitly state which attributes to assign a variable.
- The attributes that a compiler assigns to a variable can be overridden at link time with the psect_attr link option. This option overrides attributes of all variables in the psect.

Consult your compiler reference manual for further details.

Embedded SQL/C Preprocessor Errors

To correct most errors, you may wish to run the embedded SQL preprocessor with the listing (-l) option on. The listing is sufficient for locating the source and reason for the error.

For preprocessor error messages specific to C and C++, see Preprocessor Error Messages in this chapter.

C++ Programming

This section tells you how to embed ESQL statements in C++ programs, how to build ESQL/C++ programs, and what restrictions to observe in ESQL/C++ programs. The ESQL/C++ preprocessor is available only on the UNIX platform.

Creating ESQL/C++ Programs

The ESQL/C++ precompiler supports the same features as the ESQL/C precompiler, plus the additional features described in this section.

Program Comments

You can use either C-style comment delimiters (**/* */**) or C++ comment delimiters (**//**) in ESQL/C++ programs.

For example:

```
/* Declare data */
exec sql begin declare section;
    int idno;
        // identification number
    exec sql end declare section;
```

Declaring Data

ESQL/C++ supports C data types, including pointers and structures. To declare data in ESQL/C++ applications, use the data types and techniques described in C Variables and Data Types in this chapter.

You cannot declare an entire class to ESQL/C++; however, you can declare the class members. For example:

Wrong:

```
exec sql begin declare section;
  class Employee {
  char *     name;     // Name
  char *  address;     // Address
  char *  title;       // Title
  int     age;
  public:
       Employee();      // Constructor
       ~Employee();     // Destructor
       void operator=(const Employee&);     // Assignment
       void print();    // Print
       void select(char *);    // Select
};
exec sql end declare section;
```

Right:

```
class Employee {
exec sql begin declare section;
  char *     name;     // Name
  char *  address;     // Address
  char *  title;       // Title
  int     age;
 exec sql end declare section;
 public:
     Employee();       // Constructor
     ~Employee();      // Destructor
     void operator=(const Employee&);     // Assignment
     void print();     // Print
     void select(char *);     // Select
};
```

Transferring Data Between Programs and the Database

To transfer data between your application and the database, you can use either of the following techniques:

- Declare class members to ESQL, and use them in ESQL DML statements (select, update, insert, and delete) in class member functions.
- Copy data between class members and local variables. Use the local variables in ESQL statements to transfer data between your application and the database.

Names of variables that are declared to the ESQL/C++ precompiler must be unique in the scope of the source file. If you declare class members, avoid using the same name for members in different classes.

Declaring Function Parameters

To declare function parameters to ESQL/C++, use local variables. In the following example, the local variables ptrsqlvar1 and locsqlvar2 are declared to the ESQL precompiler. The function parameters sqlvar1 and sqlvar2 are copied to ptrsqlvar1 and locsqlvar2 when their values are required for use in ESQL statements.

```
int myfunc (int sqlvar1, int sqlvar2)
 {
  exec sql begin declare section
    int *ptrsqlvar1;   /* Use local pointer */
    int locsqlvar2;   /*Use local variable */
  exec sql end declare section
  ptrsqlvar1 = &sqlvar1;
  locsqlvar2 = sqlvar2;
 // Use local variables in SQL statement:
  exec sql insert into mytable
    values (ptrsqlvar1, locsqlvar2);
...
 }
```

DCLGEN and ESQL/C++

DCLGEN does not generate classes for C++. DCLGEN generates structures as it does for C. However, you can specify C++ as the language parameter (by specifying cc).

For example:

dclgen cc mydatabase mytable myfile**.dcl** mystructure

Ingres Runtime Library Prototypes

In each ESQL/C++ file you precompile, the precompiler automatically includes header files containing function prototypes for the Ingres runtime library routines.

4GL Restriction

You cannot call an ESQL/C++ routine from 4GL, Vision, or OpenROAD.

Creating User-Defined Handlers

(For basic information about user-defined handlers, see User-Defined Error, DBevent, and Message Handlers and User-Defined Data Handlers for Large Objects in this chapter.)

To declare user-defined handlers in ESQL/C++ programs, you must declare them to the C++ compiler as C functions. For example:

```
// Function prototype for event handler
extern "C" int event_func(void);
```

To direct the DBMS to call the handler routine when a database event is raised, your application must issue the following SQL statement:

```
exec sql set_sql(dbeventhandler=event_func);
```

You cannot overload a function that you intend to use as a handler routine.

User-defined handlers (*data handlers*) for long varchar and long byte I/O require an argument to be passed to the data handler. The argument must be defined as a generic pointer (void *) in the function prototype, and must be cast to the correct data type in the data handler routine.

The following example illustrates the construction of a data handler in C++:

```
// Handler prototype
// ESQL/C++ requires extern "C"
extern "C" int Put_Handler(void *hdlr_arg);
 typedef struct hdlr_param_
{
       char * arg_str;
       int arg_int;
 } HDLR_PARAM;
 void
main()
{
       HDLR_PARAM hdlr_arg;
       // Connect to the database
       exec sql connect testdatabase;
       exec sql insert into book(idno, long_text) values (1,
          datahandler(Put_Handler(&hdlr_arg)));
       exec sql disconnect;
 }

// Argument is declared as a generic pointer
int Put_Handler(void *hdlr_arg)
 {
       exec sql begin declare section;
          char seg_buf[50];
          int seg_length;
          int data_end;
       exec sql end declare section;
 // Cast argument for ESQL/C++
 ((HDLR_PARAM *)hdlr_arg)->arg_int = 0;
 rloop:
   read data from a file
    fill seg_buf and set seg_length
   at end of loop sent data_end to 1
   exec sql put data (segment = :seg_buf,
             segmentlength = :seg_length,
             dataend = :data_end);
 end rloop
       return 0;
 }
```

Building ESQL/C++ Programs

To build an ESQL/C++ program, you must precompile the ESQL/C++ source into a C++ program, compile the resulting C++ program, and link it with the Ingres runtime library.

To precompile ESQL/C++ programs, use the esqlcc command. The default extension for ESQL/C++ source files is .scc. The default extension for the C++ file generated by the precompiler is .cc. The syntax of the C++ precompiler command is as follows:

esqlcc *flags filename*

where *filename* is the name of the file containing the ESQL/C++ source for your application, and *flags* are one or more of the flags described in Command Line Operations in the Preprocessor Operation section, or the following ESQL/C++ flag:

Flag	Description
-extension = *ext*	Specifies the extension for the C++ file created by the precompiler.

To display a list of valid ESQL/C++ precompiler flags, issue the esqlcc command with no arguments.

To compile and link the resulting C++ program, invoke your C++ compiler. You must link the program with libingres.a (the Ingres runtime library). The following example illustrates the commands you must issue to build an ESQL/C++ application named inventory:

```
esqlcc -extension=cpp inventory.scc

CC -c inventory inventory.cpp

CC -o inventory inventory.o \

$II_SYSTEM/ingres/lib/libingres.a -lC
```

Sample Application

The following code is a sample ESQL/C++ application that illustrates the requirements described in this section:

```
**************************** Main Routine **************************
# include <stream.h>
// Simple ESQL/C++ program that uses the class Employee
// declared in employee.h.
// This program asks for a employee id, and then retrieves and prints
// that employee's information.
 #include "employee.h"
main()
{
// Connect to the database
exec sql connect testdatabase;
char     buf[20];     // Input buffer
// Prompt for Employee id
while (1)
   {
       Employee      a;     // Declare Employee object
       cout << "\nPlease enter employee id (or 'e' to exit): " << flush;
       cin >> buf;
       if (buf[0] == 'e')
        break;
       a.select(buf);     // Select employee info from database
       a.print();     // Print employee info
   }
exec sql disconnect;
 }

*********************** Member functions *****************************
# include <string.h>
# include <stream.h>
exec sql include 'employee.sh';
// Employee member routines

// Constructor - declare storage for all the character fields and
// Initialize to empty.
//
Employee::Employee()
{
 name = new char[MAXDATA];
 name[0] = '\0';
 address = new char[MAXDATA];
 address[0] = '\0';
 title = new char[MAXDATA];
 title[0] = '\0';
 age = 0;

}
// Destructor
Employee::~Employee()
{
 delete name;
 delete address;
 delete title;
 }
// Assignment Operator
void Employee::operator=(const Employee& a)
 {
int n;
 n = strlen(a.name);
 for ( int i = 0; i <n ; i++)
     name[i] = a.name[i];
```

```
 }
//Member functions
void Employee::print()
{
cout << "Employee Info \n";
cout << "------------- \n";
if (name[0] == '\0')
   cout << "** Employee Not found **\n";
else
{
   cout << "Name    = " << name << '\n';
   cout << "Address = " << address << '\n';
   cout << "Title = " << title << '\n';
   cout << "Age = " << age << '\n';
}
}
void Employee::select(char *empid)
{
// Use a local variable to store function argument so it can
// be declared to ESQL
exec sql begin declare section;
     char     *sqlempid;
exec sql end declare section;
sqlempid = empid;
exec sql select name, address, title, age into
     :name, :address, :title, :age
     from employee where empid = :sqlempid;
 }

********************** Class header file employee.h ********************
// Declare an employee class for C++
class Employee {
exec sql begin declare section;
char *     name;     // Name
char *  address;     // Address
char *  title;     // Title
int     age;
exec sql end declare section;
public:
     Employee();     // Constructor
     ~Employee();     // Destructor
     void operator=(const Employee&);     // Assignment
     void print();     // Print
     void select(char *);     // Select
};
const int     MAXDATA = 60
```

Preprocessor Error Messages

The following is a list of error messages specific to the C language:

E_DC000A

"Table 'employee' contains column(s) of unlimited length."

Explanation: Character strings(s) of zero length have been generated. This causes a compile-time error. You must modify the output file to specify an appropriate length.

E_E00001

"The #define statement may be used only with values, not names. Use typedef if you wish to make '%0c' a synonym for a type."

Explanation: The #define directive accepts only integer, floating-point or string literals as the replacement token. You may not use arbitrary text as the replacement token. To define type names you should use typedef. The embedded preprocessor #define is not as versatile as the C #define.

E_E00002

"Cast of #define value is ignored."

Explanation: The preprocessor ignores a cast of the replacement value in a #define statement. Casts, in general, are not supported by the embedded C preprocessor. Remove the cast from the #define statement.

E_E00003

"Incorrect indirection on variable'%0c'. Variable is subscripted, [], or dereferenced, *,%1c time(s) but declared with indirection of%2c."

Explanation: This error occurs when the address or value of a variable is incorrectly expressed because of faulty indirection. For example, the name of an integer array has been given instead of a single array element, or, in the case of character string variables, a single element of the string (that is, a character) has been given instead of a pointer to the string or the name of the array.

Either redeclare the variable with the intended indirection or change its use in the current statement.

E_E00004

"Last component of structure reference'%0c' is illegal."

Explanation: This error occurs when the preprocessor encounters an unrecognized name in a structure reference. The user may have incorrectly typed the name of structure element or may have failed to declare it to the preprocessor.

Check for misspellings in component names and that all of the structure components have been declared to the preprocessor.

E_E00008

"Incorrect declaration of C varchar variable is ignored. The members of a varchar structure variable may consist only of a short integer and a fixed length character array."

Explanation: Varchar variables (variables declared with the varchar storage class) must conform to an exact varying length string template so that Ingres can map to and from them at runtime. The length field must be exactly two bytes (derived from a short), and the character string field must be a single-dimensioned C character array. The varchar clause must be associated with a variable declaration and not with a type definition or structure tag declaration.

Check the varchar structure declaration. Make sure that both structure members are declared properly.

E_E00009

"Missing'=' in the initialization part of a C declaration."

Explanation: The preprocessor allows automatic initialization of variables and expects the regular C syntax. Insert an equals sign between the variable and the initializing value.

Sample Applications

This section contains sample applications.

The Department-Employee Master/Detail Application

This application uses two database tables joined on a specific column. This typical example of a department and its employees demonstrates how to process two tables as a master and a detail.

The program scans through all the departments in a database table, in order to reduce expenses. Based on certain criteria, the program updates department and employee records. The conditions for updating the data are the following:

Departments

- If a department has made less than $50,000 in sales, the department is dissolved.

Employees

- If an employee was hired since the start of 1998, the employee is terminated.

- If the employee's yearly salary is more than the minimum company wage of $14,000 and the employee is not nearing retirement (over 58 years of age), the employee takes a 5% pay cut.
- If the employee's department is dissolved and the employee is not terminated, the employee is moved into a state of limbo to be resolved by a supervisor.

This program uses two cursors in a master/detail fashion. The first cursor is for the Department table, and the second cursor is for the Employee table. The declare table statements at the beginning of the program describe both tables. The cursors retrieve all the information in the tables, some of which is updated. The cursor for the Employee table also retrieves an integer date interval whose value is positive if the employee was hired after January 1, 1998. The tables contain no null values.

Each row that is scanned, from both the Department table and the Employee table, is recorded into the system output file. This file serves both as a log of the session and as a simplified report of the updates that were made.

Each section of code is commented for the purpose of the application and also to clarify some of the uses of the embedded SQL statements. The program illustrates table creation, multi-statement transactions, all cursor statements, direct updates, and error handling.

Note: The application uses function prototypes and ifdef statements to enable you to build it using either the ESQL/C or ESQL/C++ precompiler.

`Sample Program`

```
# include <stdio.h>
EXEC SQL INCLUDE SQLCA;
/* The department table */
EXEC SQL DECLARE dept TABLE
  (name           char(12) NOT NULL,    /* Department name */
   totsales       money NOT NULL,       /* Total sales */
   employees      smallint NOT NULL);   /* Number of employees */

/* The employee table */
EXEC SQL DECLARE employee TABLE
  (name           char(20) NOT NULL,    /* Employee name */
   age            integer1 NOT NULL,    /* Employee age */
   idno           integer NOT NULL,     /* Unique employee id */
   hired          date NOT NULL,        /* Date of hire */
   dept           char(12) NOT NULL,    /* Department of work */
   salary         money NOT NULL);      /* Yearly salary */

/* "State-of-Limbo" for employees who lose their department */
EXEC SQL DECLARE toberesolved TABLE
  (name           char(20) NOT NULL,
   age            integer1 NOT NULL,
   idno           integer NOT NULL,
   hired          date NOT NULL,
   dept           char(12) NOT NULL,
   salary         money NOT NULL);
EXEC SQL BEGIN DECLARE SECTION;
# define MIN_DEPT_SALES    50000.00   /* Minimum sales of department */
# define MIN_EMP_SALARY    14000.00   /* Minimum employee salary */
# define NEARLY_RETIRED    58
# define SALARY_REDUC      0.95
EXEC SQL END DECLARE SECTION;
/*
** Function prototypes for C++ only so that this is compatible
** with old-style C compilers
*/
# ifdef __cplusplus
void Init_Db(void);
void End_Db(void);
void Process_Depts(void);
void Process_Employees( char *dept_name, short deleted_dept, short *emps_term );
void Close_Down(void);
# endif /* __cplusplus */
/*

** Procedure: MAIN
** Purpose:   Main body of the application. Initialize the database,
**            process each department and terminate the session.
** Parameters:
**            None
*/main()
{
  printf( "Entering application to process expenses.\n" );
  Init_Db();
  Process_Depts();
  End_Db();
  printf( "Successful completion of application.\n" );
 }

/*
** Procedure: Init_Db
** Purpose:   Initialize the database.
**            Connect to the database, and abort if an error. Before
**            processing employees, create the table for employees
**            who lose their department, "toberesolved".
** Parameters:
**            None
```

```
*/

# ifdef __cplusplus
void
Init_Db(void)
# else
Init_Db()

# endif /* __cplusplus */
{

  EXEC SQL WHENEVER SQLERROR STOP;
  EXEC SQL CONNECT personnel;


printf( "Creating \"To_Be_Resolved\" table.\n" );
  EXEC SQL CREATE TABLE toberesolved
  (name               char(20) NOT NULL,
  age                 integer1 NOT NULL,
  idno                integer NOT NULL,
  hired               date NOT NULL,
  dept                char(12) NOT NULL,
  salary              money NOT NULL);

}

/*
** Procedure: End_Db
** Purpose:   Commit the multi-statement transaction and access
**            to the database.
** Parameters:
**            None
*/

# ifdef __cplusplus
void
End_Db(void)
# else
End_Db()
# endif /* __cplusplus */
{
  EXEC SQL COMMIT;
  EXEC SQL DISCONNECT;
 }

/*
** Procedure: Process_Depts
** Purpose:   Scan through all the departments, processing each one.
**            If the department has made less than $50,000 in sales,
**            the department is dissolved. For each department, process
**            all  employees (they may even be moved to another table).
**            If an employee was terminated, then update the department's
**            employee counter.
** Parameters:
**            None
*/

# ifdef __cplusplus
void
Process_Depts(void)
# else
Process_Depts()
# endif /* __cplusplus */
{
  EXEC SQL BEGIN DECLARE SECTION;
    struct dept_ {           /* Corresponds to the "dept" table */
```

```
    char    name[13];
    double  totsales;
    short   employees;
  } dept;
  short   emps_term = 0;               /* Employees terminated */
EXEC SQL END DECLARE SECTION;
short       deleted_dept;              /* Was the dept deleted? */
char        *dept_format;              /* Formatting value */

EXEC SQL DECLARE deptcsr CURSOR FOR
  SELECT name, totsales, employees
  FROM dept
  FOR DIRECT UPDATE OF name, employees;

/* All errors from this point on close down the application */
EXEC SQL WHENEVER SQLERROR CALL Close_Down;
EXEC SQL WHENEVER NOT FOUND GOTO Close_Dept_Csr;

EXEC SQL OPEN deptcsr;

while (sqlca.sqlcode == 0)
{
  EXEC SQL FETCH deptcsr INTO :dept;
  /* Did the department reach minimum sales? */
  if (dept.totsales < MIN_DEPT_SALES)
  {
    EXEC SQL DELETE FROM dept
      WHERE CURRENT OF deptcsr;
    deleted_dept = 1;
    dept_format = "  --  DISSOLVED  --";
  }
  else
  {
    deleted_dept = 0;
    dept_format = "";
  }

  /* Log what we have just done */
  printf( "Department: %14s, Total Sales: %12.3f %s\n",
      dept.name, dept.totsales, dept_format );

  /* Now process each employee in the department */
  Process_Employees( dept.name, deleted_dept, &emps_term );

  /* If  employees were terminated, record this fact */
  if (emps_term > 0 && !deleted_dept)
  {
    EXEC SQL UPDATE dept
      SET employees = :dept.employees - :emps_term
      WHERE CURRENT OF deptcsr;
  }

}
Close_Dept_Csr:
  EXEC SQL WHENEVER NOT FOUND CONTINUE;
  EXEC SQL CLOSE deptcsr;
}

/*
** Procedure: Process_Employees
** Purpose:   Scan through all the employees for a particular department.
**            Based on given conditions the employee may be terminated or
**            given a salary reduction:
**            1. If an employee was hired since 1998, the employee is
**               terminated.
**            2. If the employee's yearly salary is more than minimum
```

```
**               company wage of $14,000 and the employee is not close to
**               retirement (over 58 years of age), the employee
**               takes a 5% salary reduction.
**            3. If the employee's department is dissolved and the employee
**               is not terminated, then the employee is moved into the
**               "toberesolved" table.
** Parameters:
**             dept_name     - Name of current department.
**             deleted_dept  - Is current department being dissolved?
**             emps_term     - Set locally to record how many employees
**                            were terminated for the current department.
*/

# ifdef __cplusplus
void
Process_Employees( char *dept_name, short deleted_dept, short *emps_term )
# else
Process_Employees( dept_name, deleted_dept, emps_term
)
char    *dept_name;
short   deleted_dept;
short   *emps_term;
# endif /* __cplusplus */
{
  EXEC SQL BEGIN DECLARE SECTION;
    struct emp_ {       /* Corresponds to "employee" table */
      char    name[21];
      short   age;
      int     idno;
      char    hired[26];
      float   salary;
      int     hired_since_98;
    } emp;
    char    *dname = dept_name;

  EXEC SQL END DECLARE SECTION;
  char         *title;            /* Formatting values */
  char         *description;

  /*
  ** Note the use of the INGRES function to find out who has been
  ** hired since 1998.
  */
  EXEC SQL DECLARE empcsr CURSOR FOR
    SELECT name, age, idno, hired, salary,
      int4(interval('days', hired-date('01-jan-1998')))
    FROM employee
    WHERE dept = :dname
    FOR DIRECT UPDATE OF name, salary;


  /* All errors from this point on close down the application */
  EXEC SQL WHENEVER SQLERROR CALL Close_Down;
  EXEC SQL WHENEVER NOT FOUND GOTO Close_Emp_Csr;

  EXEC SQL OPEN empcsr;

  *emps_term = 0;                 /* Record how many */
  while (sqlca.sqlcode == 0)
  {
    EXEC SQL FETCH empcsr INTO :emp;

    if (emp.hired_since_98 > 0)
     {
      EXEC SQL DELETE FROM employee
        WHERE CURRENT OF empcsr;
```

```
      title = "Terminated:";
      description = "Reason: Hired since 1998.";
      (*emps_term)++;

    }
    else
    {
      /* Reduce salary if not nearly retired */
      if (emp.salary > MIN_EMP_SALARY)
        {
        if (emp.age < NEARLY_RETIRED)
        {
          EXEC SQL UPDATE employee
            SET salary = salary * :SALARY_REDUC
            WHERE CURRENT OF empcsr;
          title = "Reduction: ";
          description = "Reason: Salary.";
        }
        else
        {
          /* Do not reduce salary */
          title = "No Changes:";
          description = "Reason: Retiring.";
        }

      }

      else    /* Leave employee alone */
      {
        title = "No Changes:";
        description = "Reason: Salary.";
      }

      /* Was employee's department dissolved? */
      if (deleted_dept)
      {
        EXEC SQL INSERT INTO toberesolved
          SELECT *
          FROM employee
          WHERE idno = :emp.idno;

        EXEC SQL DELETE FROM employee
          WHERE CURRENT OF empcsr;
      }
    }


    /* Log the employee's information */
    printf( "  %s %6d, %20s, %2d, %8.2f; %s\n",
      title, emp.idno, emp.name, emp.age, emp.salary,
      description );
  }

  Close_Emp_Csr:
    EXEC SQL WHENEVER NOT FOUND CONTINUE;
    EXEC SQL CLOSE empcsr;
 }

/*
** Procedure: Close_Down
** Purpose:   Error handler called any time after Init_Db has been
**            successfully completed. In all cases, print the cause of
**            the error and abort the transaction, backing out changes.
**            Note that disconnecting from the database will implicitly
**            close any open cursors.
** Parameters:
```

```
**           None
*/

# ifdef __cplusplus
void
Close_Down(void)
# else
Close_Down()
# endif /* __cplusplus */
{
  EXEC SQL BEGIN DECLARE SECTION;
    char       errbuf[101];
  EXEC SQL END DECLARE SECTION;

  EXEC SQL WHENEVER SQLERROR CONTINUE;  /* Turn off error handling */

  EXEC SQL INQUIRE_INGRES (:errbuf = ERRORTEXT);
  printf( "Closing Down because of database error:\n" );
  printf( "%s\n", errbuf );

  EXEC SQL ROLLBACK;
  EXEC SQL DISCONNECT;
  exit( -1 );
 }
```

The Table Editor Table Field Application

This application edits the Person table in the Personnel database. It is a forms application that allows the user to update a person's values, remove the person, or add new persons. Various table field utilities are provided with the application to demonstrate how they work.

The objects used in this application are shown in the following table:

Object	Description
personnel	The program's database environment.
person	A database table with three columns: Name (char(20)) Age (smallint) Number (integer) Number is unique.
personfrm	The VIFRED form with a single table field.

Object	Description
persontbl	A table field in the form, with two columns:name (char(20)) age (integer) When initialized the table file includes the hidden column number (integer).

At the start of the application, a database cursor is opened to load the table field with data from the Person table. After loading the table field, you can browse and edit the displayed values. You can add, update, or delete entries. When finished, the values are unloaded from the table field, and your updates are transferred back into the Person table.

Note: The application uses function prototypes and ifdef statements to enable you to build it using either the ESQL/C or ESQL/C++ precompiler.

Sample Program

```
# include <stdio.h>
```

```
# include <string.h>

EXEC SQL INCLUDE SQLCA;

EXEC SQL DECLARE person TABLE
  (name     char(20),       /* Person name */
   age      smallint,       /* Age */
   number   integer);       /* Unique id number */

/*
** Function prototypes for C++ only so that this is compatible
** with old-style C compilers
*/
# ifdef __cplusplus
int Load_Table(void);
# endif /* __cplusplus */

/*
** Procedure: MAIN
** Purpose:   Entry point into Table Editor program.
*/

main()
{
/* Table field row states */
# define stUNDEF      0   /* Empty or undefined row */
# define stNEW        1   /* Appended by user */
# define stUNCHANGED  2   /* Loaded by program - not updated */
# define stCHANGE     3   /* Loaded by program - since changed */
# define stDELETE     4   /* Deleted by program */

# define NOT_FOUND  100   /* SQLCA value for no rows */

  EXEC SQL BEGIN DECLARE SECTION;

    /* Person information */
    char pname[21];      /* Full name (with C null) */
    int  page,           /* Age of person */
         pnumber;        /* Unique person number */
    int  maxid;          /* Max person id number */

    /* Table field entry information */
    int  state,          /* State of data set entry */
         record,         /* Record number */
         lastrow;        /* Last row in table field */


    /* Utility buffers */
    char msgbuf[100],    /* Message buffer */
         respbuf[256];   /* Response buffer */
  EXEC SQL END DECLARE SECTION;

  int update_error;            /* Update error from database */
  int xact_aborted;            /* Transaction aborted */

  /* Set up error handling for main program */
  EXEC SQL WHENEVER SQLWARNING CONTINUE;
  EXEC SQL WHENEVER NOT FOUND CONTINUE;
  EXEC SQL WHENEVER SQLERROR STOP;

  /* Start up INGRES and the INGRES/FORMS system */
  EXEC SQL CONNECT 'personnel';

  EXEC FRS FORMS;
```

```
/* Verify that the user can edit the "person" table */
EXEC FRS PROMPT NOECHO ('Password for table editor: ', :respbuf);

if (strcmp(respbuf, "MASTER_OF_ALL") != 0)
{
  EXEC FRS MESSAGE 'No permission for task. Exiting . . .';
  EXEC FRS ENDFORMS;
  EXEC SQL DISCONNECT;
  exit( 1 );
}

/* We assume no SQL errors can happen during screen updating */
EXEC SQL WHENEVER SQLERROR CONTINUE;

EXEC FRS MESSAGE 'Initializing Person Form . . .';
EXEC FRS FORMINIT personfrm;

/*
**  Initialize "persontbl" table field with a data set in FILL mode,
**  so that the runtime user can append rows. To keep track of
**  events occurring to original rows loaded into the table field,
**  hide the unique person number.
*/
EXEC FRS INITTABLE personfrm persontbl FILL (number = integer);

maxid = Load_Table();

EXEC FRS DISPLAY personfrm UPDATE;
EXEC FRS INITIALIZE;

EXEC FRS ACTIVATE MENUITEM 'Top';
EXEC FRS BEGIN;
  /*
  ** Provide menu items, as well as the system FRS key,
  ** to scroll to both extremes of the table field.
  */
  EXEC FRS SCROLL personfrm persontbl TO 1;
EXEC FRS END;

EXEC FRS ACTIVATE MENUITEM 'Bottom';
EXEC FRS BEGIN;
  EXEC FRS SCROLL personfrm persontbl TO END;  /* Forward */
EXEC FRS END;


EXEC FRS ACTIVATE MENUITEM 'Remove';
EXEC FRS BEGIN;
  /*
  ** Remove the person in the row the user's cursor is on.
  ** If there are no persons, exit operation with message.
  ** Note that this check cannot really happen, as there is
  ** always an UNDEFINED row in FILL mode.
  */
  EXEC FRS INQUIRE_FRS table personfrm
      (:lastrow = lastrow(persontbl));
  if (lastrow == 0)
  {
    EXEC FRS MESSAGE 'Nobody to Remove';
    EXEC FRS SLEEP 2;
    EXEC FRS RESUME FIELD persontbl;
  }
  EXEC FRS DELETEROW personfrm persontbl; /* Record later */
EXEC FRS END;

EXEC FRS ACTIVATE MENUITEM 'Find';
```

```
EXEC FRS BEGIN;
  /*
  ** Scroll user to the requested table field entry.
  ** Prompt the user for a name, and if one is typed in,
  ** loop through the data set searching for it.
  */
  EXEC FRS PROMPT ('Person''s name : ', :respbuf);
  if (respbuf[0] == '\0')
    EXEC FRS RESUME FIELD persontbl;

  EXEC FRS UNLOADTABLE personfrm persontbl
    (:pname = name,
     :record = _record,
     :state = _state);
  EXEC FRS BEGIN;

    /* Do not compare with deleted rows */
    if ((strcmp(pname, respbuf) == 0) &&
        (state != stDELETE))
    {
      EXEC FRS SCROLL personfrm persontbl
        TO :record;
      EXEC FRS RESUME FIELD persontbl;
    }

  EXEC FRS END;

  /* Fell out of loop without finding name */
  sprintf(msgbuf,
    "Person \"%s\" not found in table [HIT RETURN] ",
    respbuf);
  EXEC FRS PROMPT NOECHO (:msgbuf, :respbuf);
EXEC FRS END;

EXEC FRS ACTIVATE MENUITEM 'Exit';
EXEC FRS BEGIN;
  EXEC FRS VALIDATE FIELD persontbl;
  EXEC FRS BREAKDISPLAY;
EXEC FRS END;
EXEC FRS FINALIZE;


/*
** Exit person table editor and unload the table field. If any
** updates, deletions or additions were made, duplicate these
** changes in the source table. If the user added new people,
** assign a unique id to each person before adding the person to
** the table. To do this, increment the previously-saved maximum
** id number with each insert.
*/

/* Do all the updates in a transaction */
EXEC SQL SAVEPOINT savept;

/*
** Hard code the error handling in the UNLOADTABLE loop, as
** we want to cleanly exit the loop.
*/
EXEC SQL WHENEVER SQLERROR CONTINUE;

update_error = 0;
xact_aborted = 0;

EXEC FRS MESSAGE 'Exiting Person Application . . .';
EXEC FRS UNLOADTABLE personfrm persontbl
  (:pname = name, :page = age,
```

```
   :pnumber = number, :state = _state);
EXEC FRS BEGIN;

  /* Appended by user. Insert with new unique id. */
  if (state == stNEW)
  {
    maxid = maxid + 1;
    EXEC SQL INSERT INTO person (name, age, number)
      VALUES (:pname, :page, :maxid);
  }
  /* Updated by user. Reflect in table. */
  else if (state == stCHANGE)
  {
    EXEC SQL UPDATE person SET
      name = :pname, age = :page
      WHERE number = :pnumber;
  }
  /*
  ** Deleted by user, so delete from table. Note that only
  ** original rows, not rows appended at runtime, are
  ** saved by the program.
  */
  else if (state == stDELETE)
  {
    EXEC SQL DELETE FROM person
      WHERE number = :pnumber;
  }
  /* Else UNDEFINED or UNCHANGED - No updates */


  /*
  ** Handle error conditions -
  ** If an error occurred, abort the transaction.
  ** If no rows were updated, inform user and prompt
  ** for continuation.
  */
  if (sqlca.sqlcode < 0)     /* Error */
  {
    EXEC SQL INQUIRE_INGRES (:msgbuf = ERRORTEXT);
    EXEC SQL ROLLBACK TO savept;
    update_error = 1;
    xact_aborted = 1;
    EXEC FRS ENDLOOP;
  }
  else if (sqlca.sqlcode == NOT_FOUND)
  {
    sprintf(msgbuf,
      "Person \"%s\" not updated. Abort all updates? ",
      pname);
    EXEC FRS PROMPT (:msgbuf, :respbuf);
    if (respbuf[0] == 'Y' || respbuf[0] == 'y')
    {
      EXEC SQL ROLLBACK TO savept;
      xact_aborted = 1;
      EXEC FRS ENDLOOP;
    }
  }

EXEC FRS END;

if (!xact_aborted)
  EXEC SQL COMMIT;       /* Commit the updates */

EXEC FRS ENDFORMS;         /* Terminate the FORMS and INGRES */
EXEC SQL DISCONNECT;
```

```
  if (update_error)
  {
    printf( "Your updates were aborted because of error:\n" );
    printf( msgbuf );
    printf( "\n" );
  }

} /* Main Program */

/*
** Procedure: Load_Table
** Purpose:   Load the table field from the "person" table. The
**            columns "name" and "age" will be displayed, and
**            "number" will be hidden.
** Parameters:
**            None
** Returns:
**            Maximum employee number
*/
# ifdef __cplusplus
int
Load_Table(void)
# else
int
Load_Table()
# endif /* __cplusplus */


{
  EXEC SQL BEGIN DECLARE SECTION;
    /* Person information */
    char pname[21];      /* Full name */
    int  page,           /* Age of person */
         pnumber;        /* Unique person number */
    int  maxid;          /* Max person id number to return */
  EXEC SQL END DECLARE SECTION;

  EXEC SQL DECLARE loadtab CURSOR FOR
    SELECT name, age, number
    FROM person;

  /* Set up error handling for loading procedure */
  EXEC SQL WHENEVER SQLERROR GOTO Load_End;
  EXEC SQL WHENEVER NOT FOUND GOTO Load_End;

  EXEC FRS MESSAGE 'Loading Person Information . . .';

  /* Fetch the maximum person id number for later use */
  EXEC SQL SELECT max(number)
    INTO :maxid
    FROM person;

  EXEC SQL OPEN loadtab;

  while (sqlca.sqlcode == 0)
  {
    /* Fetch data into record and load table field */
    EXEC SQL FETCH loadtab INTO :pname, :page, :pnumber;

    EXEC FRS LOADTABLE personfrm persontbl
      (name = :pname, age = :page, number = :pnumber);
  }

  Load_End:
       EXEC SQL WHENEVER SQLERROR CONTINUE;
       EXEC SQL CLOSE loadtab;
```

```
  return maxid;

} /* Load_Table */
```

The Professor-Student Mixed Form Application

This application lets the user browse and update information about graduate students who report to a specific professor. The program is structured in a master/detail fashion, with the professor being the master entry, and the students the detail entries. The application uses two forms—one to contain general professor information and another for detailed student information.

The objects used in this application are shown in the following table:

Object	Description
personnel	The program's database environment.
professor	A database table with two columns: pname (char(25)) pdept (char(10)) See its declare table statement in the program for a full description.

Object	Description
student	A database table with seven columns: sname (char(25)) sage (integer1) sbdate (char(25)) sgpa (float4) sidno (integer) scomment (varchar(200)) sadvisor (char(25)) See its declare table statement for a full description. The sadvisor column is the join field with the pname column in the Professor table.
masterfrm	The main form has fields pname and pdept, which correspond to the information in the Professor table and the studenttbl table field. The pdept field is display-only. This form is a compiled form.
studenttbl	A table field in masterfrm with the sname and sage columns. When initialized, it also has five hidden columns corresponding to information in the Student table.
studentfrm	The detail form, with seven fields, which corresponds to information in the Student table. Only the sgpa, scomment, and sadvisor fields are updatable. All other fields are display-only. This form is a compiled form.
grad	A global structure, whose members correspond in name and type to the columns of the Student database table, the studentfrm form, and the studenttbl table field.

The program uses the masterfrm as the general-level master entry, in which you can only retrieve and browse data, and the studentfrm as the detailed screen, in which you can update specific student information.

Enter a name in the pname field and then select the Students menu operation. The operation fills the studenttbl table field with detailed information of the students reporting to the named professor. The studentcsr database cursor in the Load_Students procedure does this. The program assumes that each professor is associated with exactly one department. You can then browse the table field (in read mode), which displays only the names and ages of the students. You can request more information about a specific student by selecting the Zoom menu operation. This operation displays the studentfrm form (in update mode). The fields of studentfrm are filled with values stored in the hidden columns of studenttbl. You can make changes to the sgpa, scomment, and sadvisor fields. If validated, these changes are written back to the database table (based on the unique student id), and to the table field's data set. You can repeat this process for different professor names.

Note: The application uses function prototypes and ifdef statements to enable you to build it using either the ESQL/C or ESQL/C++ precompiler.

Sample Program

```
# include <stdio.h>

# include <string.h>

EXEC SQL INCLUDE SQLCA;
 EXEC SQL DECLARE student TABLE     /* Graduate student table */
  (sname         char(25),         /* Name */
   sage          integer1,         /* Age */
   sbdate        char(25),         /* Birth date */
   sgpa          float4,           /* Grade point average */
   sidno         integer,          /* Unique student number */
   scomment      varchar(200),     /* General comments */
   sadvisor      char(25));        /* Advisor's name */
EXEC SQL DECLARE professor TABLE   /* Professor table
*/
  (pname         char(25),         /* Professor's name */
   pdept         char(10));        /* Department */

EXEC SQL BEGIN DECLARE SECTION;
  /* Global grad student record maps to database table */
  struct {
    char   sname[26];
    short  sage;
    char   sbdate[26];
    float  sgpa;
    int    sidno;
    char   scomment[201];
    char   sadvisor[26];
  } grad;
```

```
EXEC SQL END DECLARE SECTION;
/*
** Function prototypes for C++ only so that this is compatible
** with old-style C compilers
*/
# ifdef __cplusplus
void Master(void);
 void Load_Students(char *adv);
 int Student_Info_Changed(void);
 # endif /* __cplusplus */

/*
** Procedure: MAIN
** Purpose:   Start up program and call Master driver.
*/
main()
{
  /* Start up INGRES and the FORMS system */
  EXEC FRS FORMS;

  EXEC SQL WHENEVER SQLERROR STOP;
  EXEC FRS MESSAGE 'Initializing Student Administrator . . .';

  EXEC SQL CONNECT personnel;

  Master();

  EXEC FRS CLEAR SCREEN;
  EXEC FRS ENDFORMS;
  EXEC SQL DISCONNECT;
 }


/*
** Procedure: Master
** Purpose:   Drive the application, by running "masterfrm" and
**            allowing the user to "zoom" into a selected student.
** Parameters:
**            None - Uses the global student "grad" record.
*/

# ifdef __cplusplus
void
Master(void)
# else
Master()
# endif /* __cplusplus */
{
  EXEC SQL BEGIN DECLARE SECTION;
    /* Professor info maps to database table */
    struct {
      char  pname[26];
      char  pdept[11];
    } prof;

    /* Useful forms system information */
    int  lastrow,  /* Lastrow in table field */
         istable;  /* Is a table field? */

    /* Local utility buffers */
    char  msgbuf[100];      /* Message buffer */
    char  respbuf[256];     /* Response buffer */
    char  old_advisor[26];  /* Old advisor before ZOOM */

    /* Externally compiled master form */
```

```
  extern int *masterfrm;
EXEC SQL END DECLARE SECTION;

EXEC FRS ADDFORM :masterfrm;

/*
** Initialize "studenttbl" with a data set in READ mode.
** Declare hidden columns for all the extra fields that
** the program will display when more information is
** requested about a student. Columns "sname" and "sage"
** are displayed. All other columns are hidden, to be
** used in the student information form.
*/
EXEC FRS INITTABLE masterfrm studenttbl READ
  (sbdate = char(25),
   sgpa = float4,
   sidno = integer4,
   scomment = char(200),
   sadvisor = char(20));

EXEC FRS DISPLAY masterfrm UPDATE;

EXEC FRS INITIALIZE;
EXEC FRS BEGIN;
  EXEC FRS MESSAGE 'Enter an Advisor name . . .';
  EXEC FRS SLEEP 2;
EXEC FRS END;

EXEC FRS ACTIVATE MENUITEM 'Students', FIELD 'pname';

EXEC FRS BEGIN;


  /* Load the students of the specified professor */
  EXEC FRS GETFORM (:prof.pname = pname);

  /* If no professor name is given, resume */
  if (prof.pname[0] == '\0')
    EXEC FRS RESUME FIELD pname;

  /*
  ** Verify that the professor exists. Local error handling
  ** just prints the message and continues. Assume that each
  ** professor has exactly one department.
  */
  EXEC SQL WHENEVER SQLERROR CALL SQLPRINT;
  EXEC SQL WHENEVER NOT FOUND CONTINUE;
  prof.pdept[0] = '\0';
  EXEC SQL SELECT pdept
    INTO :prof.pdept
    FROM professor
    WHERE pname = :prof.pname;

  if (prof.pdept[0] == '\0')
  {
    sprintf(msgbuf,
      "No professor with name \"%s\" [RETURN]",
      prof.pname);
    EXEC FRS PROMPT NOECHO (:msgbuf, :respbuf);
    EXEC FRS CLEAR FIELD ALL;
    EXEC FRS RESUME FIELD pname;
  }

  /* Fill the department field and load students */
  EXEC FRS PUTFORM (pdept = :prof.pdept);
  EXEC FRS REDISPLAY;       /* Refresh for query */
```

```
    Load_Students(prof.pname);

    EXEC FRS RESUME FIELD studenttbl;

  EXEC FRS END;          /* "Students" */

  EXEC FRS ACTIVATE MENUITEM 'Zoom';
  EXEC FRS BEGIN;


    /*
    ** Confirm that user is in "studenttbl" and that
    ** the table field is not empty. Collect data from
    ** the row and zoom for browsing and updating.
    */
    EXEC FRS INQUIRE_FRS field masterfrm
      (:istable = table);

    if (istable == 0)
    {
      EXEC FRS PROMPT NOECHO
        ('Select from the student table [RETURN]',
         :respbuf);
      EXEC FRS RESUME FIELD studenttbl;
    }

    EXEC FRS INQUIRE_FRS table masterfrm
      (:lastrow = lastrow);

    if (lastrow == 0)
    {
      EXEC FRS PROMPT NOECHO
        ('There are no students [RETURN]',
         :respbuf);
      EXEC FRS RESUME FIELD pname;
    }

    /* Collect all data on student into global record
*/
    EXEC FRS GETROW masterfrm studenttbl
        (:grad.sname = sname,
         :grad.sage = sage,
         :grad.sbdate = sbdate,
         :grad.sgpa = sgpa,
         :grad.sidno = sidno,
         :grad.scomment = scomment,
         :grad.sadvisor = sadvisor);

    /*
    ** Display "studentfrm," and if any changes were made,
    ** make the updates to the local table field row.
    ** Only make updates to the columns corresponding to
    ** writable fields in "studentfrm". If the student
    ** changed advisors, then delete the row from the display.
    */
    strcpy(old_advisor, grad.sadvisor);
    if (Student_Info_Changed())
    {
      if (strcmp(old_advisor, grad.sadvisor) != 0)
        EXEC FRS DELETEROW masterfrm studenttbl;
      Else
        EXEC FRS PUTROW masterfrm studenttbl
          (sgpa = :grad.sgpa,
           scomment = :grad.scomment,
           sadvisor = :grad.sadvisor);
```

```
    }

  EXEC FRS END;          /* "Zoom" */

  EXEC FRS ACTIVATE MENUITEM 'Exit';
  EXEC FRS BEGIN;
    EXEC FRS BREAKDISPLAY;
  EXEC FRS END;          /* "Exit" */
  EXEC FRS FINALIZE;

} /* Master */

/*
** Procedure: Load_Students
** Purpose:   Given an advisor name, load into the "studenttbl"
**            table field all the students who report to the
**            professor with that name.
** Parameters:
**            advisor - User-specified professor name.
**            Uses the global student record.
*/

# ifdef __cplusplus
void
Load_Students(char *adv)
# else
Load_Students(adv)
char    *adv;
# endif /* __cplusplus */
{
  EXEC SQL BEGIN DECLARE SECTION;
    char   *advisor = adv;
  EXEC SQL END DECLARE SECTION;

  EXEC SQL DECLARE studentcsr CURSOR FOR
    SELECT sname, sage, sbdate, sgpa, sidno, scomment, sadvisor
    FROM student
    WHERE sadvisor = :advisor;

  /*
  ** Clear previous contents of table field. Load the table
  ** field from the database table based on the advisor name.
  ** Columns "sname" and "sage" will be displayed, and all
  ** others will be hidden.
  */
  EXEC FRS MESSAGE 'Retrieving Student Information . . .';

  EXEC FRS CLEAR FIELD studenttbl;

  EXEC SQL WHENEVER SQLERROR GOTO Load_End;
  EXEC SQL WHENEVER NOT FOUND GOTO Load_End;

  EXEC SQL OPEN studentcsr;


  /*
  ** Before we start the loop, we know that the OPEN was
  ** successful and that NOT FOUND was not set.
  */
  while (sqlca.sqlcode == 0)
  {
    EXEC SQL FETCH studentcsr INTO :grad;

    EXEC FRS LOADTABLE masterfrm studenttbl
        (sname = :grad.sname,
```

```
         sage = :grad.sage,
         sbdate = :grad.sbdate,
         sgpa = :grad.sgpa,
         sidno = :grad.sidno,
         scomment = :grad.scomment,
         sadvisor = :grad.sadvisor);
  }

Load_End:          /* Clean up on an error, and close
cursors */
  EXEC SQL WHENEVER NOT FOUND CONTINUE;
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL CLOSE studentcsr;

} /* Load_Students */

/*
** Procedure: Student_Info_Changed
** Purpose:   Allow the user to zoom in on the details of a selected
**            student. Some of the data can be updated by the user.
**            If any updates were made, then reflect these back into
**            the database table. The procedure returns TRUE if any
**            changes were made.
** Parameters:
**            None - Uses data in the global "grad" record.
** Returns:
**            TRUE/FALSE - Changes were made to the database.
**            Sets the global "grad" record with the new data. */

# ifdef __cplusplus
int
Student_Info_Changed(void)
 # else
int
Student_Info_Changed()
# endif /* __cplusplus */

{
  EXEC SQL BEGIN DECLARE SECTION;
    int changed;                 /* Changes made to data in form */
    int valid_advisor;           /* Valid advisor name? */
    extern int *studentfrm;      /* Compiled form */

  EXEC SQL END DECLARE SECTION;

  /* Control ADDFORM to only initialize once */
  static int loadform = 0;

  if (!loadform)
  {
    EXEC FRS MESSAGE 'Loading Student form . . .';
    EXEC FRS ADDFORM :studentfrm;
    loadform = 1;   }


  /*  Local error handler just prints error and continues */
  EXEC SQL WHENEVER SQLERROR CALL SQLPRINT;
  EXEC SQL WHENEVER NOT FOUND CONTINUE;

  EXEC FRS DISPLAY studentfrm FILL;
  EXEC FRS INITIALIZE
    (sname = :grad.sname,
     sage = :grad.sage,
     sbdate = :grad.sbdate,
     sgpa = :grad.sgpa,
     sidno = :grad.sidno,
```

```
     scomment = :grad.scomment,
     sadvisor = :grad.sadvisor);

  EXEC FRS ACTIVATE MENUITEM 'Write';
  EXEC FRS BEGIN;

    /*
    ** If changes were made, then update the database
    ** table. Only bother with the fields that are not
    ** read-only.
    */
    EXEC FRS INQUIRE_FRS form (:changed = change);

    if (changed == 1)
    {
      EXEC FRS VALIDATE;
      EXEC FRS MESSAGE 'Writing changes to database. . .';

      EXEC FRS GETFORM
        (:grad.sgpa = sgpa,
         :grad.scomment = scomment,
         :grad.sadvisor = sadvisor);

      /* Enforce integrity of professor name */
      valid_advisor = 0;
      EXEC SQL SELECT 1 INTO :valid_advisor
        FROM professor
        WHERE pname = :grad.sadvisor;

      if (valid_advisor == 0)
      {
        EXEC FRS MESSAGE 'Not a valid advisor name';
        EXEC FRS SLEEP 2;
        EXEC FRS RESUME FIELD sadvisor;
      }

      else
      {
        EXEC SQL UPDATE student SET
          sgpa = :grad.sgpa,
          scomment = :grad.scomment,
          sadvisor = :grad.sadvisor
          WHERE sidno = :grad.sidno;
        EXEC FRS BREAKDISPLAY;
      }
    }
  EXEC FRS END;                 /* "Write" */


  EXEC FRS ACTIVATE MENUITEM 'Quit';
  EXEC FRS BEGIN;
    /* Quit without submitting changes */
    changed = 0;
    EXEC FRS BREAKDISPLAY;
  EXEC FRS END;                 /* "Quit" */

  EXEC FRS FINALIZE;
  return (changed == 1);
}  /* Student_Info_Changed *
```

The SQL Terminal Monitor Application

This application executes SQL statements that are read in from the terminal. The application reads statements from input and writes results to output. Dynamic SQL is used to process and execute the statements.

When the application starts, it prompts the user for the database name. The program then prompts for an SQL statement. Each SQL statement can continue over multiple lines and must end with a semicolon. No SQL comments are accepted. The SQL statement is processed using Dynamic SQL, and results and SQL errors are written to output. At the end of the results, an indicator of the number of rows affected is displayed. The loop is then continued and the program prompts for another SQL statement. When the user types in end-of-file, the application rolls back any pending updates and disconnects from the database.

The user's SQL statement is prepared using prepare and describe. If the SQL statement is not a select statement, then it is run using execute and the number of rows affected is printed. If the SQL statement is a select statement, a Dynamic SQL cursor is opened, and all the rows are fetched and printed. The routines that print the results do not try to tabulate the results. A row of column names is printed, followed by each row of the results.

Keyboard interrupts are not handled. Fatal errors such as allocation errors and boundary condition violations are handled by means of rolling back pending updates and disconnecting from the database session.

Note: The application uses function prototypes and ifdef statements to enable you to build it using either the ESQL/C or ESQL/C++ precompiler.

Sample Program

```
# include <stdio.h>
# include <malloc.h>

/* Declare the SQLCA structure and the SQLDA typedef */
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;
EXEC SQL DECLARE stmt STATEMENT;
/* Dynamic SQL statement */
EXEC SQL DECLARE csr CURSOR FOR stmt;    /* Cursor for dynamic SQL statement */
/*
** Default number of result columns for a dynamic SELECT. If a SELECT
** statement returns more than DEF_ELEMS,  a new SQLDA will be allocated
*/
# define  DEF_ELEMS  5
```

```
/* Size of a DATE string variable */
# define  DATE_SIZE  25

/* The SQL code for the NOT FOUND condition */
# define  SQL_NOTFOUND  100
/* Buffer lengths */
# define  DBNAME_MAX  50    /* Max database name */
# define  INPUT_SIZE  256    /* Max input line length */
# define  STMT_MAX  1000    /* Max SQL statement length */

/* Global SQL variables */

IISQLDA    *sqlda = (IISQLDA *)0;  /* Pointer to the
SQL dynamic area */
/* Result storage buffer for dynamic SELECT statements */
struct {
  int    res_length;  /* Size of mem_data */
  char    *res_data;  /* Pointer to allocated result buffer */
} res_buf = {0, NULL};

/*
** Function prototypes for C++ only so that this is compatible
** with old-style C compilers
*/
# ifdef __cplusplus
void  Run_Monitor(void);        /* Run SQL Monitor */
void  Init_Sqlda(int num_elems);/* Initialize SQLDA */
int  Execute_Select(void);      /* Execute dynamic SELECT */
void  Print_Header(void);       /* Print SELECT column headers */
void  Print_Row(void);          /* Print SELECT row values */
void  Print_Error(void);        /* Print a user error */
char   *Read_Stmt(int stmt_num, char *stmt_buf, int stmt_max);
        /* Read statement from terminal */
char   *Alloc_Mem(int mem_size, char *error_string);
        /* Allocate memory */
char   *calloc(unsigned nelem, unsigned elsize);
        /* C allocation routine */
# else
void  Run_Monitor();    /* Run SQL Monitor */
void  Init_Sqlda();     /* Initialize SQLDA */
int  Execute_Select();  /* Execute dynamic SELECT */
void  Print_Header();   /* Print SELECT column headers */
void  Print_Row();      /* Print SELECT row values */
void  Print_Error();    /* Print a user error */
char   *Read_Stmt();    /* Read statement from terminal */
char   *Alloc_Mem();    /* Allocate memory */
char   *calloc();    /* C allocation routine */
# endif /* __cplusplus */
```

```
/*
** Procedure:  main
** Purpose:  Main body of SQL Monitor application. Prompt for database
**    name and connect to the database. Run the monitor and
**    disconnect from the database. Before disconnecting roll
**    back any pending updates.
** Parameters:
**      None
*/

main()
{
  EXEC SQL BEGIN DECLARE SECTION;
    char  dbname[DBNAME_MAX +1];    /* Database name */
  EXEC SQL END DECLARE SECTION;

  /* Prompt for database name - could be command line parameter */
  printf("SQL Database: ");
  if (fgets(dbname, DBNAME_MAX, stdin) == NULL)
    exit(1);

  printf( "-- SQL Terminal Monitor --\n" );

  /* Treat connection errors as fatal */
  EXEC SQL WHENEVER SQLERROR STOP;
  EXEC SQL CONNECT :dbname;

  Run_Monitor();

  EXEC SQL WHENEVER SQLERROR CONTINUE;

  printf("SQL: Exiting monitor program.\n");
  EXEC SQL ROLLBACK;
  EXEC SQL DISCONNECT;
 } /* main */

/*
** Procedure:  Run_Monitor
** Purpose:  Run the SQL monitor. Initialize the first SQLDA with the
**    default size (DEF_ELEMS 'sqlvar' elements). Loop while
**    prompting the user for input, and processing the statement.
**    If it is not a SELECT statement then execute it, otherwise
**    open a cursor a process a dynamic SELECT statement.
** Parameters:
**    None
*/

# ifdef __cplusplus
void
Run_Monitor(void)
# else
void
Run_Monitor()
# endif /* __cplusplus */
{
      EXEC SQL BEGIN DECLARE SECTION;
      char  stmt_buf[STMT_MAX +1];  /* SQL statement input buffer */
  EXEC SQL END DECLARE SECTION;

  int    stmt_num;    /* SQL statement number */
  int    rows;        /* Rows affected */


  /* Allocate a new SQLDA */
  Init_Sqlda(DEF_ELEMS);
```

```
  /* Now we are set for input */
  for (stmt_num = 1;; stmt_num++)
  {
      /*
      ** Prompt and read the next statement. If Read_Stmt
      ** returns NULL then end-of-file was detected.
      */
      if (Read_Stmt(stmt_num, stmt_buf, STMT_MAX) == NULL)
      break;

      /* Errors are non-fatal from here on out */
      EXEC SQL WHENEVER SQLERROR GOTO Stmt_Err;

      /*
      ** Prepare and describe the statement. If we cannot fully describe
      ** the statement (our SQLDA is too small) then allocate a new one
      ** and redescribe the statement.
      */
      EXEC SQL PREPARE stmt FROM :stmt_buf;
      EXEC SQL DESCRIBE stmt INTO :sqlda;
      if (sqlda->sqld > sqlda->sqln)
      {
        Init_Sqlda(sqlda->sqld);
        EXEC SQL DESCRIBE stmt INTO :sqlda;
      }

      /* If 'sqld' = 0 then this is not a SELECT */
      if (sqlda->sqld == 0)
      {
      EXEC SQL EXECUTE stmt;
      rows = sqlca.sqlerrd[2];
      }
      else  /* SELECT */
      {
      rows = Execute_Select();
      }
      printf("[%d row(s)]\n", rows);
      continue;        /* Skip error handler */

    Stmt_Err:
      EXEC SQL WHENEVER SQLERROR CONTINUE;
      /* Print error messages here and continue */
      Print_Error();
  } /* for each statement */
} /* Run_Monitor */


/*
** Procedure:  Init_Sqlda
** Purpose:  Initialize SQLDA. Free any old SQLDA's and allocate a new
**    one. Set the number of 'sqlvar' elements.
** Parameters:
**    num_elems    - Number of elements.
*/

# ifdef __cplusplus
void
Init_Sqlda(int num_elems)
# else
void
Init_Sqlda(num_elems)
 int  num_elems;
# endif /* __cplusplus */
{
  /* Free the old SQLDA */
```

```
  if (sqlda)
      free((char *)sqlda);

  /* Allocate a new SQLDA */
  sqlda = (IISQLDA *)
    Alloc_Mem(IISQDA_HEAD_SIZE + (num_elems * IISQDA_VAR_SIZE),
        "new SQLDA");
  sqlda->sqln = num_elems;    /* Set the size */
} /* Init_Sqlda */

/*
** Procedure:  Execute_Select
** Purpose:    Run a dynamic SELECT statement. The SQLDA has already been
**             described, so print the column header (names), open a cursor,
**             and retrieve and print the results. Accumulate the number or
**             rows processed.
** Parameters: None
** Returns:    Number of rows processed.
*/

# ifdef __cplusplus
int
Execute_Select(void)
 # else
int
Execute_Select()
# endif /* __cplusplus */
{
  int  rows;      /* Counter for rows fetched */

  /*
  ** Print the result column names, allocate the result variables,
  ** and set up the types.
  */
  Print_Header();
  EXEC SQL WHENEVER SQLERROR GOTO Close_Csr;

  /* Open the dynamic cursor */
  EXEC SQL OPEN csr;


  /* Fetch and print each row */
  rows = 0;
  while (sqlca.sqlcode == 0)
  {
      EXEC SQL FETCH csr USING DESCRIPTOR :sqlda;
      if (sqlca.sqlcode == 0)
      {
    rows++;        /* Count the rows */
    Print_Row();
      }
  } /* While there are more rows */

Close_Csr:
  /* If we got here because of an error then print the error message */
  if (sqlca.sqlcode < 0)
      Print_Error();
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL CLOSE csr;

  return rows;
 } /* Execute_Select */

/*
** Procedure:  Print_Header
** Purpose:  A statement has just been described so set up the SQLDA for
```

```
**    result processing. Print all the column names and allocate
**    a result buffer for retrieving data. The result buffer is
**    one buffer (whose size is determined by adding up the results
**    column sizes). The 'sqldata' and 'sqlind' fields are pointed
**    at offsets into this buffer.
** Parameters:
**      None
*/

# ifdef __cplusplus
void
Print_Header(void)
# else
void
Print_Header()
# endif /* __cplusplus */
{
  int    i;      /* Index into 'sqlvar' */
  IISQLVAR  *sqv;      /* Pointer to 'sqlvar */
  int    base_type;    /* Base type w/o nullability */
  int    res_cur_size;    /* Result size required */
  int    round;      /* Alignment */

  /*
  ** For each column print its title (and number), and accumulate
  ** the size of the result data area.
  */
  for (res_cur_size = 0, i = 0; i < sqlda->sqld; i++)
  {
      /* Print each column name and its number */
      sqv = &sqlda->sqlvar[i];
      printf("[%d] %.*s ",
       i+1, sqv->sqlname.sqlnamel, sqv->sqlname.sqlnamec);

      /* Find the base-type of the result (non-nullable) */
      if ((base_type = sqv->sqltype) < 0)
      base_type = -base_type;


      /* Collapse different types into INT, FLOAT or CHAR */
      switch (base_type)
      {
        case IISQ_INT_TYPE:
      /* Always retrieve into a long integer */
      res_cur_size += sizeof(long);
      sqv->sqllen = sizeof(long);
      break;

        case IISQ_MNY_TYPE:
      /* Always retrieve into a double floating-point */
      if (sqv->sqltype < 0)
          sqv->sqltype = -IISQ_FLT_TYPE;
      else
          sqv->sqltype = IISQ_FLT_TYPE;
      res_cur_size += sizeof(double);
      sqv->sqllen = sizeof(double);
      break;

        case IISQ_FLT_TYPE:
      /* Always retrieve into a double floating-point */
      res_cur_size += sizeof(double);
      sqv->sqllen = sizeof(double);

      break;

        case IISQ_DTE_TYPE:
```

```
        sqv->sqllen = DATE_SIZE;
        /* Fall through to handle like CHAR */

          case IISQ_CHA_TYPE:
          case IISQ_VCH_TYPE:
        /*
        ** Assume no binary data is returned from the CHAR type.
        ** Also allocate one extra byte for the null terminator.
        */
        res_cur_size += sqv->sqllen + 1;
        /* Always round off to aligned data boundary */
        round = res_cur_size % 4;
        if (round)
            res_cur_size += 4 - round;
        if (sqv->sqltype < 0)
            sqv->sqltype = -IISQ_CHA_TYPE;
        else
            sqv->sqltype = IISQ_CHA_TYPE;
        break;
        } /* switch on base type */

        /* Save away space for the null indicator */
        if (sqv->sqltype < 0)
        res_cur_size += sizeof(int);
    } /* for each column */

    printf("\n\n");


    /*
    ** At this point we've printed all column headers and converted all
    ** types to one of INT, CHAR or FLOAT. Now we allocate a single
    ** result buffer, and point all the result column data areas into it.
    **
    ** If we have an old result data area that is not large enough then free
    ** it and allocate a new one. Otherwise we can reuse the last one.
    */

    if (res_buf.res_length > 0 > && res_buf.res_length < res_cur_size)
    {
        free(res_buf.res_data);
        res_buf.res_length = 0;
    }
    if (res_buf.res_length == 0)
    {
        res_buf.res_data = Alloc_Mem(res_cur_size,
             "result data storage area");
        res_buf.res_length = res_cur_size;
    }

    /*
    ** Now for each column now assign the result address (sqldata) and
    ** indicator address (sqlind) from the result data area.
    */
    for (res_cur_size = 0, i = 0; i < sqlda->sqld; i++)
    {
        sqv = &sqlda->sqlvar[i];

        /* Find the base-type of the result (non-nullable) */
        if ((base_type = sqv->sqltype) < 0)
        base_type = -base_type;

        /* Current data points at current offset */
        sqv->sqldata = (char *)&res_buf.res_data[res_cur_size];
        res_cur_size += sqv->sqllen;
```

```
        if (base_type == IISQ_CHA_TYPE)
        {
          res_cur_size++;      /* Add one for null */
      round = res_cur_size % 4;  /* Round to aligned boundary */
      if (round)
      res_cur_size += 4 - round;
        }

        /* Point at result indicator variable */
        if (sqv->sqltype < 0)
        {
      sqv->sqlind = (short *)&res_buf.res_data[res_cur_size];
      res_cur_size += sizeof(int);
        }
        else
        {
      sqv->sqlind = (short *)0;
        } /* if type is nullable */
    } /* for each column */
  } /* Print_Header */

  /*
  ** Procedure:  Print_Row
  ** Purpose:  For each element inside the SQLDA, print the value. Print
  **    its column number too in order to identify it with a column
  **    name printed earlier. If the value is NULL print "N/A".
  ** Parameters:
  **    None
  */

  # ifdef __cplusplus
  void
  Print_Row(void)
  # else
  void
  Print_Row()
  # endif /* __cplusplus */
  {
    int    i;      /* Index into 'sqlvar' */
    IISQLVAR  *sqv;      /* Pointer to 'sqlvar */
    int    base_type;    /* Base type w/o nullability */

    /*
    ** For each column, print the column number and the data.
    ** NULL columns print as "N/A".
    */
    for (i = 0; i < sqlda->sqld; i++)
    {
        /* Print each column value with its number */
        sqv = &sqlda->sqlvar[i];
        printf("[%d] ", i+1);

        if (sqv->sqlind && *sqv->sqlind < 0)
        {
      printf("N/A ");
        }
        else /* Either not nullable, or nullable but not null */
        {
      /* Find the base-type of the result (non-nullable) */
      if ((base_type = sqv->sqltype) < 0)
          base_type = -base_type;

      switch (base_type)
      {
        case IISQ_INT_TYPE:
          /* All integers were retrieved into long integers */
```

```
        printf("%d ", *(long *)sqv->sqldata);
        break;

      case IISQ_FLT_TYPE:
        /* All floats were retrieved into doubles */
        printf("%g ", *(double *)sqv->sqldata);
        break;

      case IISQ_CHA_TYPE:
        /* All characters were null terminated */
        printf("%s ", (char *)sqv->sqldata );
        break;
    } /* switch on base type */
      } /*if not null */
  } /* foreach column */
  printf("\n");
} /* Print_Row */


/*
** Procedure:  Print_Error
** Purpose:    SQLCA error detected. Retrieve the error message and print it.
** Parameters: None
*/

# ifdef __cplusplus
void
Print_Error(void)
 # else
void
Print_Error()
# endif /* __cplusplus */
{
  EXEC SQL BEGIN DECLARE SECTION;
    char  error_buf[150];    /* For error text retrieval */
  EXEC SQL END DECLARE SECTION;

  EXEC SQL INQUIRE_INGRES (:error_buf = ERRORTEXT);
  printf("\nSQL Error:\n    %s\n", error_buf );
 } /* Print_Error */

/*
** Procedure: Read_Stmt
** Purpose:   Reads a statement from standard input. This routine prompts
**            the user for input (using a statement number) and scans input
**            tokens for the statement delimiter (semicolon).
**            - Continues over new-lines.
**            - Uses SQL string literal rules.
** Parameters:
**             stmt_num - Statement number for prompt.
**             stmt_buf - Buffer to fill for input.
**             stmt_max - Max size of statement.
** Returns:
**             A pointer to the input buffer. If NULL then end-of-file was
**             typed in.
 */

# ifdef __cplusplus
char *
Read_Stmt(int stmt_num, char *stmt_buf, int stmt_max)
# else
char *
Read_Stmt(stmt_num, stmt_buf, stmt_max)
 int    stmt_num;
 char    *stmt_buf;
 int    stmt_max;
```

```
 # endif /* __cplusplus */
{
  char  input_buf[INPUT_SIZE +1];  /* Terminal input buffer */
  char  *icp;          /* Scans input buffer */
  char  *ocp;          /* To output (stmt_buf) */
  int  in_string;        /* For string handling */


  printf("%3d> ", stmt_num);    /* Prompt user */
  ocp = stmt_buf;
  in_string = 0;
  while (fgets(input_buf, INPUT_SIZE, stdin) != NULL)
  {
      for (icp = input_buf; *icp && (ocp - stmt_buf < stmt_max);
 icp++)
      {
    /* Not in string - check for delimiters and new lines */
    if (!in_string)
    {
        if (*icp == ';')    /* We're done */
        {
        *ocp = '\0';
        return stmt_buf;
        }
        else if (*icp == '\n')
        {
        /* New line outside of string is replaced with blank */
        *ocp++ = ' ';
        break;      /* Read next line */
        }
        else if (*icp == '\'')  /* Entering string */
        {
        in_string++;
        }
        *ocp++ = *icp;
    }
    else           /* Inside a string */
    {
        if (*icp == '\n')
        {
        break;      /* New-line in string is ignored */
        }
        else if (*icp == '\'')
        {
      if (*(icp+1) == '\'')    /* Escaped quote ? */
          *ocp++ = *icp++;
      else
          in_string--;
        }
        *ocp++ = *icp;
    } /* if in string */
      } /* for all characters in buffer */

      if (ocp - stmt_buf >= stmt_max)
      {
    /* Statement is too large; ignore it and try again */
    printf("SQL Error: Maximum statement length (%d) exceeded.\n",
           stmt_max);
    printf("%3d> ", stmt_num);      /* Re-prompt user */
    ocp = stmt_buf;
    in_string = 0;
      }
      else  /* Break on new line - print continue sign */
      {
    printf("---> ");
      }
```

```
  } /* while reading from standard input */
  return NULL;
 } /* Read_Stmt */


/*
** Procedure:  Alloc_Mem
** Purpose:    General purpose memory allocator. If it cannot allocate
**             enough space, it prints a fatal error and aborts any
**             pending updates.
** Parameters:
**      mem_size     - Size of space requested.
**      error_string - Error message to print if failure.
** Returns:
**      Pointer to newly allocated space.
*/

# ifdef __cplusplus
char *
Alloc_Mem(int mem_size, char *error_string)
# else
char *
Alloc_Mem(mem_size, error_string)
 int  mem_size;
 char  *error_string;
# endif /* __cplusplus */
{
  char  *mem;

  mem = calloc(1, mem_size);
  if (mem)
      return mem;

  /* Print an error and roll back any updates */
  printf("SQL Fatal Error: Cannot allocate %s (%d bytes).\n",
         error_string, mem_size);
  printf("Any pending updates are being rolled back.\n");
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL ROLLBACK;
  EXEC SQL DISCONNECT;
  exit(-1);
 } /* Alloc_Mem */
```

A Dynamic SQL/Forms Database Browser

This program lets the user browse data from and insert data into any table in any database, using a dynamically defined form. The program uses Dynamic SQL and Dynamic FRS statements to process the interactive data. You should already have used VIFRED to create a Default Form based on the database table that you want to browse. VIFRED will build a form with fields that have the same names and data types as the columns of the specified database table.

When run, the program prompts the user for the name of the database, the table and the form. The form is profiled using the describe form statement, and the field name, data type, and length information is processed. From this information, the program fills in the SQLDA data and null indicator areas, and builds two Dynamic SQL statement strings to select data from and insert data into the database.

The Browse menu item retrieves the data from the database using an SQL cursor associated with the dynamic select statement, and displays that data using the dynamic putform statement. A submenu allows the user to continue with the next row or return to the main menu. The Insert menu item retrieves the data from the form using the dynamic getform statement, and adds the data to the database table using a prepared insert statement. The Save menu item commits the changes and, because prepared statements are discarded, again prepares the select and insert statements. When the user selects Quit, all pending changes are rolled back and the program is terminated.

Note: The application uses function prototypes and ifdef statements to enable you to build it using either the ESQL/C or ESQL/C++ precompiler.

Sample Program

```
 include <stdio.h>
# include <string.h>
# include <malloc.h>

/*
** Declare the SQLCA structure and the SQLDA typedef.
*/
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;
EXEC SQL DECLARE sel_stmt STATEMENT;        /* Dynamic
SQL SELECT statement */
EXEC SQL DECLARE ins_stmt STATEMENT;        /* Dynamic SQL INSERT statement */
EXEC SQL DECLARE csr CURSOR FOR sel_stmt;   /* Cursor for SELECT statement
/
/*
** Buffer lengths.
*/
# define        NAME_MAX         50      /* Max name lengths */
# define        STMT_MAX        1000    /* Max SQL statement length
/
/*
** Global SQL variables.
*/
IISQLDA *sqlda = (IISQLDA *)0;          /* Pointer
o the SQL dynamic area */
/*
** Result storage buffer for dynamic SQL and FRS statements.
** This result buffer is dynamically allocated and filled.
** Each SQLDA SQLVAR sets its SQLDATA and SQLIND pointers to
** offsets in this buffer.
 /
struct {
    int         res_length;     /* Size of res_data */
    char        *res_data;      /* Pointer to allocated result buffer */
} res_buf = {0, (char *)0};
/*
* Procedures in this file.
** Function prototypes for C++ only so that this is compatible
** with old-style C compilers
*/
# ifdef __cplusplus
int Describe_Form(char *formname, char *tabname, char *sel_buf, char *ins_buf);
        /* DESCRIBE form and set up SQL statements */
oid Init_Sqlda(int num_elems);  /* Initialize SQLDA */
char *calloc(unsigned nelem, unsigned elsize);
 # else
int     Describe_Form();     /* DESCRIBE form and set up SQL statements */
void    Init_Sqlda();        /* Initialize SQLDA */
char    *calloc();           /* C allocation routine */
# endif /* __cplusplus */

/*
** Procedure:   main
** Purpose:     Main body of Dynamic SQL forms application. Prompt for
**              database, form and table name. Call Describe_Form
**              to obtain a profile of the form and set up the SQL
**              statements. Then allow the user to interactively browse
*              the database table and append new data.
*/
main()
{
    EXEC SQL BEGIN DECLARE SECTION;
        char    dbname[NAME_MAX +1];            /* Database name */
        char    formname[NAME_MAX +1];           /* Form name */
        char    tabname[NAME_MAX +1];           /* Table name */
        char    sel_buf[STMT_MAX +1];           /* Prepared SELECT */
```

```
    char    ins_buf[STMT_MAX +1];           /* Prepared INSERT */
    int     err;                            /* Error status */
    char    ret[10];                        /* Prompt error buffer
*/
   EXEC SQL END DECLARE SECTION;
    EXEC FRS FORMS;
    /* Prompt for database name - will abort on errors
*/
    EXEC SQL WHENEVER SQLERROR STOP;
    EXEC FRS PROMPT ('Database name: ', :dbname);
    EXEC SQL CONNECT :dbname;
    EXEC SQL WHENEVER SQLERROR CALL SQLPRINT;
    /*
    ** Prompt for table name - later a Dynamic SQL SELECT statement
    ** will be built from it.
    */
    EXEC FRS PROMPT ('Table name: ', :tabname);
    /*
    ** Prompt for form name. Check forms errors reported
    ** through INQUIRE_FRS.
    */
    EXEC FRS PROMPT ('Form name: ', :formname);
    EXEC FRS MESSAGE 'Loading form ...';
    EXEC FRS FORMINIT :formname;
    EXEC FRS INQUIRE_FRS FRS (:err = ERRORNO);
    if (err > 0)
    {
       EXEC FRS MESSAGE 'Could not load form. Exiting.';
        EXEC FRS ENDFORMS;
        EXEC SQL DISCONNECT;
        exit(1);
    }
    /* Commit any work done so far - access of forms catalogs */

   EXEC SQL COMMIT;

    /* Describe the form and build the SQL statement strings */
    if (!Describe_Form(formname, tabname, sel_buf, ins_buf))
    {
        EXEC FRS MESSAGE 'Could not describe form. Exiting.';
        EXEC FRS ENDFORMS;
       EXEC SQL DISCONNECT;
        exit(1);
    }
    /*
    ** PREPARE the SELECT and INSERT statements that correspond to the
    ** menu items Browse and Insert. If the Save menu item is chosen
    ** the statements are reprepared.
    */
    EXEC SQL PREPARE sel_stmt FROM :sel_buf;
    err = sqlca.sqlcode;
    EXEC SQL PREPARE ins_stmt FROM :ins_buf;
    if ((err < 0) || (sqlca.sqlcode < 0))
    {

       EXEC FRS MESSAGE 'Could not prepare SQL statements. Exiting.';
        EXEC FRS ENDFORMS;
        EXEC SQL DISCONNECT;
        exit(1);
    }

   /*
   ** Display the form and interact with user, allowing browsing
   ** and the inserting of new data.
   */
    EXEC FRS DISPLAY :formname FILL;
```

```
EXEC FRS INITIALIZE;
EXEC FRS ACTIVATE MENUITEM 'Browse';
EXEC FRS BEGIN;
    /*
    ** Retrieve data and display the first row on the form, allowing
    ** the user to browse through successive rows. If data types
    ** from the database table are not consistent with data
    ** descriptions obtained from the form, a retrieval error
    ** will occur. Inform the user of this or other errors.
    **
    ** Note that the data will return sorted by the first field that
    ** was described, as the SELECT statement, sel_stmt, included an
    ** ORDER BY clause.
    */
    EXEC SQL OPEN csr;
    /* Fetch and display each row */
   while (sqlca.sqlcode == 0)
    {
        EXEC SQL FETCH csr USING DESCRIPTOR :sqlda;
        if (sqlca.sqlcode != 0)
        {
            EXEC SQL CLOSE csr;
            EXEC FRS PROMPT NOECHO ('No more rows :', :ret);
           EXEC FRS CLEAR FIELD ALL;
            EXEC FRS RESUME;
        }
        EXEC FRS PUTFORM :formname USING DESCRIPTOR :sqlda;
        EXEC FRS INQUIRE_FRS FRS (:err = ERRORNO);
        if (err > 0)
       {
            EXEC SQL CLOSE csr;
            EXEC FRS RESUME;
        }

        /* Display data before prompting user with submenu */
       EXEC FRS REDISPLAY;

       EXEC FRS SUBMENU;
        EXEC FRS ACTIVATE MENUITEM 'Next', FRSKEY4;
        EXEC FRS BEGIN;
            /* Continue with cursor loop */
            EXEC FRS MESSAGE 'Next row ...';
            EXEC FRS CLEAR FIELD ALL;
        EXEC FRS END;

       EXEC FRS ACTIVATE MENUITEM 'End', FRSKEY3;
        EXEC FRS BEGIN;
            EXEC SQL CLOSE csr;
            EXEC FRS CLEAR FIELD ALL;
            EXEC FRS RESUME;
        EXEC FRS END;
    } /* While there are more rows */
EXEC FRS END;
EXEC FRS ACTIVATE MENUITEM 'Insert';
EXEC FRS BEGIN;
    EXEC FRS GETFORM :formname USING DESCRIPTOR :sqlda;
    EXEC FRS INQUIRE_FRS FRS (:err = ERRORNO);
   if (err > 0)
    {
        EXEC FRS CLEAR FIELD ALL;
        EXEC FRS RESUME;
    }
    EXEC SQL EXECUTE ins_stmt USING DESCRIPTOR :sqlda;
    if ((sqlca.sqlcode < 0) || (sqlca.sqlerrd[2] == 0))

   {
```

```
            EXEC FRS PROMPT NOECHO ('No rows inserted :', :ret);
        }
        else
        {
            EXEC FRS PROMPT NOECHO ('One row inserted :', :ret);
        }

   EXEC FRS END;

    EXEC FRS ACTIVATE MENUITEM 'Save';
    EXEC FRS BEGIN;
        /*
        ** COMMIT any changes and then re-PREPARE the SELECT and INSERT
        ** statements as the COMMIT statements discards them.

        */
        EXEC SQL COMMIT;
        EXEC SQL PREPARE sel_stmt FROM :sel_buf;
        err = sqlca.sqlcode;
        EXEC SQL PREPARE ins_stmt FROM :ins_buf;
        if ((err < 0) || (sqlca.sqlcode < 0))
        {
         EXEC FRS PROMPT NOECHO ('Could not reprepare SQL statements :',
                                 :ret);
          EXEC FRS BREAKDISPLAY;
        }
    EXEC FRS END;
    EXEC FRS ACTIVATE MENUITEM 'Clear';
    EXEC FRS BEGIN;
        EXEC FRS CLEAR FIELD ALL;
    EXEC FRS END;

    EXEC FRS ACTIVATE MENUITEM 'Quit', FRSKEY2;
    EXEC FRS BEGIN;
       EXEC SQL ROLLBACK;
        EXEC FRS BREAKDISPLAY;
    EXEC FRS END;
    EXEC FRS FINALIZE;
    EXEC FRS ENDFORMS;
    EXEC SQL DISCONNECT;
 /* main */

/*
** Procedure: Describe_Form
** Purpose:   Profile the specified form for name and data type
**            information. Using the DESCRIBE FORM statement, the
**            SQLDA is loaded with field information from the form.
**             This procedure processes this information to allocate
**            result storage, point at storage for dynamic FRS data
**            retrieval and assignment, and build SQL statements
**            strings for subsequent dynamic SELECT and INSERT
**            statements. For example, assume the form (and table)
**            'emp' has the following fields:
**
**                     Field Name      Type            Nullable?
**                      ----------      ----            ---------
**                      name            char(10)        No
**                      age             integer4        Yes
**                      salary          money           Yes
**
**            Based on 'emp', this procedure will construct the SQLDA.

**            A data storage buffer, whose size is determined by
**            accumulating the field data type lengths, is allocated.
**            The SQLDATA and SQLIND fields are pointed at offsets into
**            the result storage buffer. The following SQLDA is built:
```

```
**
**                        sqlvar[0]
**                            sqltype     = IISQ_CHA_TYPE

**                            sqllen       = 10
**                            sqldata     = offset #1 into storage
**                            sqlind      = null
**                            sqlname     = 'name'
**                        sqlvar[1]
**                            sqltype     = -IISQ_INT_TYPE
**                            sqllen      = 4
**                            sqldata      = offset #2 into storage
**                            sqlind      = offset #3 into storage
**                            sqlname     = 'age'
**                        sqlvar[2]
**                            sqltype     = -IISQ_FLT_TYPE
**                            sqllen      = 8
**                            sqldata     = offset #4 into storage
**                            sqlind       = offset #5 into storage
**                            sqlname     = 'salary'
**
**           The procedure does not verify that the allocation routine
**           that is called does not fail.
**           This procedure also builds two dynamic SQL statements strings.
**            Note that the procedure should be extended to verify that the
**           statement strings do fit into the statement buffers (this was
**           not done in this example). The above example would construct
**           the following statement strings:
**
**                  'SELECT name, age, salary FROM emp ORDER BY name'
**                  'INSERT INTO emp (name, age, salary) VALUES (?, ?, ?)'
**
** Parameters:
**                formname        - Name of form to profile.
**                tabname         - Name of database table.
**                sel_buf         - Buffer to hold SELECT statement string.
**                ins_buf         - Buffer to hold INSERT statement string.
** Returns:

**                TRUE/FALSE      - Success/failure - will fail on error
**                                  or upon finding a table field.
*/

# ifdef __cplusplus
int
Describe_Form(char *formname, char *tabname, char *sel_buf, char *ins_buf)

 else
int
Describe_Form(formname, tabname, sel_buf, ins_buf)
 char    *formname;
 char    *tabname;
 char    *sel_buf;
 char    *ins_buf;

 endif /* __cplusplus */
{
    char        names[STMT_MAX +1];        /* Names for SQL statements */
    char        *nm;
    char        marks[STMT_MAX +1];        /* Place holders for INSERT */
    char        *mk;
    int         err;                       /* Error status */

   char        ret[10];                   /* Prompt error buffer */
    int         i;                         /* Index into SQLVAR */
    IISQLVAR    *sqv;                      /* Pointer to SQLVAR */
```

```
 int         base_type;                /* Base type w/o nullability*/
 int         nullable;                 /* Is nullable (SQLTYPE < 0) */
 int         res_cur_size;             /* Result size required */

/*
** Allocate a new SQLDA and DESCRIBE the form. Start out with a
** default SQLDA for 10 fields. If we cannot fully describe the
** form (our SQLDA is too small) then allocate a new one and
** redescribe the form.
*/
 Init_Sqlda(10);

EXEC FRS DESCRIBE FORM :formname ALL INTO :sqlda;
 EXEC FRS INQUIRE_FRS FRS (:err = ERRORNO);
 if (err > 0)
     return 0;                      /* Error already displayed */

 if (sqlda->sqld > sqlda->sqln)     /* Redescribe  */
 {

    Init_Sqlda(sqlda->sqld);
     EXEC FRS DESCRIBE FORM :formname ALL INTO :sqlda;
 }
 else if (sqlda->sqld == 0)         /* No fields */
 {
     EXEC FRS PROMPT NOECHO ('There are no fields in the form :', :ret);
     return 0;

}

 /*
 ** For each field determine the size and type of the data
 ** area, which will be allocated out of the result data area.
 ** This will be allocated out of res_buf in the next loop.
 ** If a table field type is returned then issue an error.

 **
 ** Also, for each field add the field name to the 'names' buffer
 ** and the SQL place holders '?' to the 'marks' buffer, which
 ** will be used to build the final SELECT and INSERT statements.
 */
 for (res_cur_size = 0, i = 0; i < sqlda->sqld; i++)
 {
    sqv = &sqlda->sqlvar[i];        /* Point at current column */
     /* Find the base-type of the result (non-nullable) */
     if ((base_type = sqv->sqltype) < 0)
     {
         nullable = 1;
         base_type = -base_type;
    }
     else
     {
         nullable = 0;
     }

     /* Collapse different types into INT, FLOAT or CHAR */
    switch (base_type)
     {
       case IISQ_INT_TYPE:
         /* Always retrieve into a long integer */
         sqv->sqltype  = IISQ_INT_TYPE;
         sqv->sqllen   = sizeof(long);
         res_cur_size += sizeof(long);

        break;
```

```
        case IISQ_MNY_TYPE:
        case IISQ_FLT_TYPE:
          /* Always retrieve into a double floating-point */
          sqv->sqltype  = IISQ_FLT_TYPE;
          sqv->sqllen   = sizeof(double);
         res_cur_size += sizeof(double);
          break;
        case IISQ_DTE_TYPE:
          sqv->sqllen = IISQ_DTE_LEN;
          /* Fall through to handle like CHAR */

       case IISQ_CHA_TYPE:
        case IISQ_VCH_TYPE:
          /*
          ** Assume no binary data is returned from the CHAR type.
          ** Also allocate one extra byte for the null terminator.
          */
          sqv->sqltype  = IISQ_CHA_TYPE;
         res_cur_size += sqv->sqllen + 1;
          /* Always round off to even data boundary */
          if (res_cur_size % 2)
              res_cur_size++;
          break;

       case IISQ_TBL_TYPE:           /* Table field */
          EXEC FRS PROMPT NOECHO ('Table field found in form :', :ret);
          return 0;

        default:
          EXEC FRS PROMPT NOECHO ('Invalid field type :', :ret);
          return 0;

     } /* switch on base type */

      /*
      ** Save away space for the null indicator and set
      ** negative  type id
      */
      if (nullable)

     {
          res_cur_size += sizeof(short);
          sqv->sqltype = -sqv->sqltype;
      }

      /*
      ** Store field names and place holders (separated by commas)

      ** for the SQL statements.
      */
      if (i == 0)
      {
    names[0] = marks[0] = '\0';
          nm = names;
          mk = marks;
     }
      else
      {
          strcat(nm++, ",");
          strcat(mk++, ",");
     }
     sprintf(nm, "%.*s", sqv->sqlname.sqlnamel, sqv->sqlname.sqlnamec);
     nm += sqv->sqlname.sqlnamel;
      strcat(mk++, "?");
  } /* for each column */
  /*
```

```
    ** At this point we've saved all field names and converted all
    ** types to one of INT, CHAR or FLOAT. Now we allocate a single
    ** result buffer, and point all the result column data areas into it.
    **
    ** If we have an old result data area that is not large enough then
    ** free it and allocate a new one. Otherwise we can reuse the last one.
    */

    if (res_buf.res_length > 0 && res_buf.res_length < res_cur_size)
    {
        free(res_buf.res_data);
        res_buf.res_length = 0;
    }
    if (res_buf.res_length == 0)
    {

       res_buf.res_data = calloc(1, res_cur_size);
        res_buf.res_length = res_cur_size;
    }


    /*
    ** Now for each column now assign the result address (SQLDATA) and
    ** indicator address (SQLIND) from the result data area.
    ** As already calculated in the previous loop, the addresses will
    ** point at offsets into res_buf.
    */
    for (res_cur_size = 0, i = 0; i < sqlda->sqld; i++)
    {
        sqv = &sqlda->sqlvar[i];
        /* Find the base-type of the result (non-nullable) */
        if ((base_type = sqv->sqltype) < 0)
            base_type = -base_type;
        /* Current data points at current offset */
        sqv->sqldata = (char *)&res_buf.res_data[res_cur_size];
        res_cur_size += sqv->sqllen;
        if (base_type == IISQ_CHA_TYPE)
        {
            res_cur_size++;             /* Add one for null */
            if (res_cur_size % 2)       /* Round off to even boundary */
                res_cur_size++;
        }

       /* Point at result indicator variable */
        if (sqv->sqltype < 0)
        {
            sqv->sqlind = (short *)&res_buf.res_data[res_cur_size];
            res_cur_size += sizeof(short);
        }
        else
       {
            sqv->sqlind = (short *)0;
        } /* if type is nullable */
    } /* for each column */
    /*
    ** Create final SELECT and INSERT statements. For the SELECT

    ** statement ORDER BY the first field.
    */
    sqv = &sqlda->sqlvar[0];
    sprintf(sel_buf, "SELECT %s FROM %s ORDER BY %.*s", names, tabname,
            sqv->sqlname.sqlnamel, sqv->sqlname.sqlnamec);
    sprintf(ins_buf, "INSERT INTO %s (%s) VALUES (%s)", tabname, names,
            marks);
   return 1;
```

```
} /* Describe_Form */

/*
** Procedure:   Init_Sqlda
** Purpose:     Initialize SQLDA. Free any old SQLDA's and allocate a new
**              one. Set the number of SQLVAR elements.
**
** Parameters:
**              num_elems    - Number of elements.
*/

 ifdef __cplusplus
void
Init_Sqlda(int num_elems)
 # else
void
Init_Sqlda(num_elems)
 int     num_elems;
 endif /* __cplusplus */
{
    /* Free the old SQLDA */
    if (sqlda)
        free((char *)sqlda);
    /* Allocate a new SQLDA */

   sqlda = (IISQLDA *)calloc(1,
                        IISQDA_HEAD_SIZE + (num_elems * IISQDA_VAR_SIZE));
    sqlda->sqln = num_elems;            /* Set the size */
} /* Init_Sqlda */
```

Multi-Threaded Applications

In standard, single-threaded embedded SQL (ESQL) applications, ESQL statements are executed in the context of the current database session. In multi-threaded applications, each thread executes ESQL statements in the context of its own current session.

ESQL needs to initialize itself for multi-threaded operation, which should be done either in single-thread mode or while multi-thread protected. ESQL does not provide a single entry point for this initialization but will perform the needed initialization on the first ESQL request. Applications need to make an ESQL call, such as INQUIRE_SQL or IIsqlca(), prior to entering the multi-threaded state.

Current Session

Each thread must designate a current session by executing the connect statement or a session switching statement. The session remains current until disconnected or another session switching statement is executed. If a thread is terminated with a current session, the session will be inaccessible until a new thread with the same thread ID as the original thread is created.

A session may be current on only one thread at any given moment. Attempting to switch to a session that is current on some other thread produces an error and no change in session is made. A session is not limited to the thread that created it; a thread may switch to any non-current session.

A thread may switch away from a session without selecting another session to be made current. Using the identifier NONE in place of the connection name or session ID in a session switching statement makes the current session accessible to other threads while leaving the current thread with no current session. The thread will need to switch to a session prior to executing any subsequent ESQL statements.

A thread may disconnect its own current session, or any session not current on another thread. Attempting to disconnect a session current on another thread results in an error being issued. The disconnect all statement may not be issued when sessions are current on any other thread.

SQLCA Diagnostic Area

In multi-threaded applications, each thread is provided its own SQLCA diagnostic area. The global SQLCA data object should not be used due to contention between threads for the global resource. Two extensions are available in the ESQLC preprocessor for gaining access to a threads SQLCA diagnostic area.

The command line flag -multi may be used to prepare an ESQLC source file for multi-threaded execution without requiring any additional changes to the source file. The -multi flag changes the code generated by ESQLC for the include sqlca statement.

Normally, the following code is generated by ESQLC when the include sqlca statement is processed:

```
#include "eqsqlca.h"
extern IISQLCA sqlca;
```

When -multi is included on the command line, ESQLC generates the following when the INCLUDE SQLCA statement is processed:

```
#include "eqsqlca.h"
IISQLCA *IIsqlca();
#define sqlca (*(IIsqlca()))
```

Using the -multi flag defines a macro which translates all references to the global sqlca variable into a call to the ESQL function IIsqlca() which returns the address to the SQLCA diagnostic area for the current thread. No code changes are required to take advantage of multi-threaded features of the ESQLC pre-processor.

Minor changes may be made to ESQLC applications to reduce the number of calls to IIsqlca() generated as described above. The ESQLC preprocessor accepts declaration of hosts' variables whose type is IISQLCA. In addition, if the host variable is a pointer type, all subsequent SQLCA references generated by ESQLC will be using the host variable.

For example, the following variable declaration will declare a SQLCA pointer host variable and initialize it to the current threads SQLCA diagnostic area:

```
EXEC SQL BEGIN DECLARE SECTION;
 IISQLCA*sqlca_ptr = IIsqlca();
EXEC SQL END DECLARE SECTION;
```

Subsequent references to the SQLCA diagnostic area may then be replaced with references to the host variable. Access to the previous error code would be coded as sqlca_ptr->sqlcode rather than sqlca.sqlcode. All subsequent SQLCA references generated by the ESQLC preprocessor use the application-declared host variable.

SQLCA variable declarations should not be global. Declarations are required in all functions containing ESQL statements.

ESQLC source files preprocessed with the –multi flag may be safely linked with files preprocessed without the -multi flag for single-threaded ESQL applications. The global SQLCA is assigned to the first (or only) thread to issue an ESQL statement.

It is recommended that applications issue an ESQL statement, such as inquire_sql or call IIsqlca() prior to starting multi-threaded execution so as to permit the ESQL runtime code to initialize safely.

Note: In multi-threaded applications, the SQLSTATE variable (or deprecated SQLCODE variable) should not be declared as a global variable. If used, SQLSTATE should be declared at the start of each function containing ESQL statements.

Chapter 3: Embedded SQL for COBOL

This chapter describes the use of Embedded SQL with the COBOL programming language.

Embedded SQL Statement Syntax for COBOL

This section describes the language-specific issues inherent in embedding SQL database and forms statements in a COBOL program. An Embedded SQL database statement has the following general syntax:

> [*margin*] **exec sql** *SQL_statement terminator*

The syntax of an embedded SQL/FORMS statement is almost identical:

> [*margin*] **exec frs** *SQL/FORMS_statement terminator*

For information on SQL statements, see the *SQL Reference Guide*. For information on SQL/FORMS statements, see the *Forms-based Application Development Tools User Guide*.

The following sections describe the various syntactical elements of these statements as implemented in COBOL.

Margin

Windows

The exec keyword, which begins all embedded SQL statements, can begin anywhere on the source line. However, you must code comment indicators, represented by the asterisk (*), in column 1 or in the COBOL indicator area (column 7). All coded string continuation indicators also belong in the COBOL indicator area.

VMS

In general, embedded SQL statements in COBOL require no special margins. The exec keyword can begin anywhere on the source line. Host declarations can also begin on any column. In the case, however, of comment lines and continued string literals contained in embedded SQL statements, the indicator symbol (* or -) must be coded in the COBOL indicator area. For programs coded using VAX COBOL terminal format conventions (the default), the indicator area is column 1. For programs coded in ANSI format (which requires specifying the -a flag on the preprocessor command line), the indicator area is column 7. Also, the -a flag allows a sequence number in specific columns on the source line. For more information on the two styles of format and the -a flag, see Preprocessor Operation in this chapter.

Comment and string literals are discussed in detail later in this section.

For portability to other implementations of SQL, you should not code beyond column 72.

COBOL Sequence Numbers

A COBOL sequence number can be placed at the beginning of any embedded SQL statement. For example:

```
000100 EXEC SQL DROP TABLE emp END-EXEC.
```

In most instances, the preprocessor outputs any COBOL sequence number that precedes an embedded SQL statement. However, in a few cases the preprocessor ignores a COBOL sequence number and does not include it in the code it generates. For example, sequence numbers occurring on embedded SQL statements that produce no COBOL code are ignored by the preprocessor. A sequence number on a continuation line for an embedded SQL statement or a declaration will be ignored.

The preprocessor never generates sequence numbers of its own. Thus, if you prefix an embedded SQL statement with a sequence number and that statement is translated by the preprocessor into several COBOL statements, the sequence number will appear before the first COBOL statement only. Subsequent COBOL statements will contain blanks in the sequence area.

A sequence number may contain any valid character in the character set. Also, it must be placed in the sequence area of a line. The sequence area ranges from Columns 1 to 6.

Embedded SQL statements in include files may also contain COBOL sequence numbers. Include files will generate sequence numbers in the same manner as outlined above.

VMS

COBOL sequence numbers can only be used in programs coded in ANSI format, which requires the -a flag on the preprocessor command line.

Terminator

The terminator for COBOL embedded SQL statements is the keyword end-exec. This terminator delimits an embedded SQL statement from the statement that follows it in the file. The following is an example of a select statement embedded in a COBOL program:

```
EXEC SQL SELECT ename
        INTO :NAMEVAR
        FROM employee
    WHERE eno = :NUMVAR
        END-EXEC
```

You have the option of following the end-exec terminator with the COBOL separator period, as, for example:

```
EXEC SQL SELECT ename
    INTO :NAMEVAR
    FROM employee
    WHERE eno = :NUMVAR
    END-EXEC.
```

In general, be sure to include the separator period wherever COBOL requires it for a normal COBOL statement (for example, at the end of a COBOL IF statement).

Do not use spaces between end-exec and the separator period. Certain considerations can arise concerning the way in which the preprocessor interprets the period. For details, see Preprocessor Operation in this chapter.

Labels

Embedded SQL statements can have a label prefix. The embedded SQL label is equivalent to a COBOL paragraph name. The label must begin with an alphanumeric character, which can be followed by alphanumeric characters, hyphens, and underscores.

Windows **UNIX** The label must be the first word on the line. It must start in column 8 or beyond, or be preceded by a tab, and it must be terminated with a period.

VMS The label must be the first word on the line (optionally preceded by white space) and must be terminated with a period.

For example:

```
CLOSE-CURSOR1.  EXEC SQL CLOSE cursor1 END-EXEC.
```

The label can appear anywhere a COBOL paragraph name can appear. Even though the preprocessor accepts it in front of any exec sql or exec frs prefix, it may not be appropriate to code it on some lines. For example, although the preprocessor accepts the following code, the code will cause a compiler error later if it is in the Data Division:

```
INCL-SQLCA.  EXEC SQL INCLUDE SQLCA END-EXEC.
```

As a general rule, use labels only with executable statements in the Procedure Division.

Line Continuation

You can continue embedded SQL statements over multiple lines. There is no continuation symbol for continuing embedded SQL statements, except in the case of continued string literals (see String Literals in this chapter.). Statements extend from the exec sql or exec frs keyword to the end-exec terminator. You can continue an embedded SQL statement onto a new line only at a word boundary, with the exception of string literals, which you can continue in a word. However, you cannot split the keyword pairs, exec sql and exec frs, between lines. Similarly, the end-exec terminator must be on a single line. You can use blank lines between continued lines.

Comments

Windows **UNIX**

An asterisk (*) in column 1 or in the indicator area indicates a COBOL comment line.

VMS

COBOL comment lines are indicated by an asterisk (*) in the indicator area. As mentioned earlier, the indicator area is either column 1 or column 7, according to whether you choose VAX COBOL terminal format or ANSI format.

You can place these comments in embedded SQL statements anywhere that blank lines are allowed, with the following exceptions:

- Between an embedded SQL/FORMS block-type statement, such as activate and unloadtable, and its associated block of code; begin and end delimit these blocks of code. Comment lines cannot appear between the statement and its section. The preprocessor interprets such comments as COBOL host code, which causes preprocessor syntax errors. For example, the following statement causes a syntax error on the COBOL comment:

```
    EXEC FRS UNLOADTABLE empform employee
            (:NAMEVAR = ename) END-EXEC
* Illegal comment before statement body
    EXEC FRS BEGIN END-EXEC
* Comment legal here
        EXEC FRS MESSAGE :NAMEVAR END-EXEC
    EXEC FRS END END-EXEC.
```

- In statements that are made up of more than one compound statement. An example of such a statement is the display statement, which typically consists of the display clause, an initialize section, activate sections and a finalize section. It cannot have COBOL comments between any of the components. The preprocessor translates these comments as host code, which causes syntax errors on subsequent statement components.

Note that the preprocessor ignores comment lines between string literal continuation lines.

The preprocessor also treats as comments any line whose indicator area contains a slash (/) to indicate a new listing page or a D to indicate a conditional compilation line.

You can also use the SQL comment delimiter (--). The preprocessor considers everything between this delimiter and the end of the line as a comment. For example:

```
EXEC SQL DELETE -- Delete all employees
FROM employee
END-EXEC
```

String Literals

Single quotes (') delimit embedded SQL string literals. To embed a single quote in a string literal, use two single quotes, as follows:

```
EXEC SQL INSERT
      INTO employee (ename)
      VALUES ('Edward ''Ted'' Smith')
      END-EXEC.
```

You can continue string literals over multiple lines. Following COBOL rules, if the continued line ends without a closing quotation mark, the continuation line must contain a hyphen (-) in the indicator area. The first non-blank character after the hyphen must be a single quotation mark, followed by the continued string as follows:

```
EXEC SQL UPDATE employee
      SET comments = 'Completed all projects on time.
-   ' Recommended for promotion.'
      WHERE name = 'Jones'
      END-EXEC.
```

VMS

As discussed earlier, the indicator area is either column 1 or column 7, depending on whether the format you are using is VAX terminal or ANSI.

In the context of a declare section, use double quotes to delimit strings in compliance with the syntax rules of the COBOL compiler.

```
01 dbname PIC X(20) VALUE "personnel".
```

String Literals and Statement Strings

The Dynamic SQL statements prepare and execute immediate both use statement strings, which specify an SQL statement. To specify the statement string, use a string literal or character string variable, as follows:

```
EXEC SQL EXECUTE IMMEDIATE 'drop employee' END-EXEC

MOVE "drop employee" TO str.
EXEC SQL EXECUTE IMMEDIATE :str END-EXEC
```

As with regular embedded SQL string literals, the statement string delimiter is the single quote. However, quotes embedded in statement strings must conform to the *runtime* rules of SQL when the statement is executed.

For example, the following two dynamic insert statements are equivalent:

```
EXEC SQL PREPARE s1 FROM
   INSERT INTO t1 VALUES (''single''''double" '')'
END-EXEC
```

and:

```
MOVE "INSERT INTO t1 VALUES ('single''double"" ')"
      TO str.
EXEC SQL PREPARE s1 FROM :str END-EXEC
```

In fact, the string literal the embedded SQL/COBOL preprocessor generates for the first example is identical to the string literal assigned to the variable str in the second example.

The *runtime* evaluation of the above statement string is:

```
INSERT INTO t1 VALUES ('single''double" ')
```

As a general rule, it is best to avoid using a string literal for a statement string whenever it may contain the single or double quote character. Instead, try to build the statement string using the COBOL language’s rules for string literals together with the SQL rules for the *runtime* evaluation of the string.

The Create Procedure Statement

The create procedure statement, according to the *SQL Reference Guide*, has language-specific syntax rules for line continuation, string literal continuation, comments, and the final terminator. These syntax rules follow the rules this section discusses — for example, the final terminator is end-exec. Regardless of the number of statements inside the procedure body, the preprocessor treats the create procedure statement as a single statement, and, when you use it as an embedded SQL/COBOL statement, you must use end-exec to terminate it. In addition, terminate all statements *within* the body of the procedure with a semicolon.

The following example shows a create procedure statement that follows the embedded SQL/COBOL syntax rules:

```
EXEC SQL
  CREATE PROCEDURE proc (parm INTEGER) AS
    DECLARE
            var INTEGER;
    BEGIN
* COBOL comment line
      IF parm > 10 THEN
      MESSAGE 'COBOL strings can continue (use hyphen)
-           ' over lines';
      INSERT INTO tab VALUES (:parm);
      ENDIF;
    END
END-EXEC.
```

COBOL Data Items and Data Types

This section describes how to declare and use COBOL program variables in Embedded SQL.

Variable and Type Declarations

Embedded SQL statements use COBOL data items, also called *variables*, to transfer data from the database or a form into the program and conversely. You must declare COBOL data items to SQL before using them in any embedded SQL statements.

Embedded SQL Variable Declaration Sections

Declare COBOL data items to SQL in a *declaration section*. This section has the following syntax:

exec sql begin declare section end-exec

COBOL variable declarations

exec sql end declare section end-exec

Place the declaration section in either the File or Working-Storage Section of the Data Division.

Embedded SQL variable declarations are global to the program file from the point of declaration onwards. You can incorporate multiple declaration sections into a single file when, for example, multiple COBOL programs appear in the same file. Each program can have its own declaration section. For more information, see Scope of Variables in this chapter.

Data Item Declaration Syntax

This section describes rules and restrictions for declaring COBOL data items in embedded SQL declaration sections. All data items in a declaration section must be declared with the correct syntax. Embedded SQL recognizes only a subset of legal COBOL declarations.

The following template is the complete data item declaration format that embedded SQL accepts:

level-number

> [*data-name* | **FILLER**]
> [**REDEFINES** *data-item*]
> [[**IS**] **GLOBAL**]
> [[**IS**] **EXTERNAL**]
> [**PICTURE** [**IS**] *pic-string*]
> [[**USAGE** [**IS**]] *use-type*]
> [**SIGN** *clause*]
> [**SYNCHRONIZED** *clause*]
> [**JUSTIFIED** *clause*]
> [**BLANK** *clause*]
> [**VALUE** *clause*]
> [**OCCURS** *clause*]

Syntax Notes:

- Data declaration clauses can be in any order, with the following two exceptions:
  - The *data-name* or FILLER clause, if given, must immediately follow the level number.
  - The REDEFINES clause, if given, must immediately follow the *data-item* or FILLER clause.
- The *level-number* can range from 01 to 49. Level number 77 (for noncontiguous data items) is also valid and the preprocessor regards it as identical to level 01. The embedded SQL preprocessor does not support Level 66 (which identifies RENAMES items) and Level 88 (which associates condition names with values).

  Follow the COBOL rules for specifying the organization of data when you assign level numbers to your embedded SQL data items. Like the COBOL compiler, the preprocessor recognizes that a data item belongs to a record or group if its level number is greater than the record or group level number.
- The *data-name* must begin with an alphabetic character, which can be followed by alphanumeric characters, hyphens, and underscores. The word FILLER can appear in place of *data-name*; however, you cannot explicitly reference a FILLER item in an embedded SQL statement. If the *data-name* or FILLER clause is omitted, FILLER is the default.

- The preprocessor accepts but does not use the REDEFINES,

  GLOBAL, EXTERNAL, SIGN, SYNCHRONIZED, JUSTIFIED, BLANK, and VALUE clauses. Consequently, illegal use of these clauses goes undetected at preprocessing time but generates COBOL errors later at compile time. For example, the preprocessor does not check that a GLOBAL clause appears only on an 01 level item, nor that a SIGN clause appears only on a numeric item.

- The preprocessor expects a PICTURE clause on the COMP, COMP-3, COMP-5 (UNIX), and DISPLAY *use-types*.

- Do not use a PICTURE clause on COMP-1 (VMS), COMP-2 (VMS), and INDEX *use-types.*

  Although the preprocessor recognizes all the valid COBOL PICTURE symbols, it only makes use of the type and size information needed for runtime support. It does not, for instance, complain about certain illegal combinations of editing symbols in picture strings. Embedded SQL accepts PIC as an abbreviation for PICTURE. You must specify the picture string on the same line as the keyword PICTURE.

- For information on the valid *use-types* for the USAGE clause and their interaction with picture strings, see Data Types in this chapter.

- The preprocessor accepts the OCCURS clause for all data items in the level range 02 through 49. The preprocessor does not use the information in the OCCURS clause, except to note that the item described is an array. If you use an OCCURS clause on level 01, the preprocessor issues an error but generates correct code so that you can compile and link the program.

Reserved Words in Declarations

The ESQL/COBOL words in the following table are reserved when used in the DECLARE section. Additionally, the words with an asterisk are also reserved wherever they are used because they have the same name as embedded SQL keywords.

You cannot declare data items with the same name as the words that do not have an asterisk and you can only use them in quoted string constants. However, the asterisked words that match ESQL keywords can have data items with the same name.

ASCENDING	**DEPENDING**	**ON ***
BLANK	**DESCENDING**	**PACKED_DECIMAL**
BY *	**DISPLAY ***	**PIC**
CHARACTER	**END-EXEC**	**PICTURE**
COMP-1	**EXTERNAL**	**POINTER**
COMP-2	**FILLER**	**REDEFINES**
COMP-3	**GLOBAL ***	**REFERENCE**
COMP-4	**IN ***	**SEPARATE**
COMP-5	**INDEX ***	**SIGN**
COMP-6	**INDEXED**	**SYNC**
COMP	**IS ***	**SYNCHRONIZED**
COMPUTATIONAL-1C	**JUST**	**TIMES**
COMPUTATIONAL-2	**JUSTIFIED**	**TO ***
COMPUTATIONAL-3	**KEY ***	**TRAILING**
COMPUTATIONAL-4	**LEADING**	**USAGE**
COMPUTATIONAL-5	**OCCURS**	**VALUE**
COMPUTATIONAL-6C	**OF ***	**WHEN ***
COMPUTATIONAL		**ZERO**

Data Types

Embedded SQL supports a subset of the COBOL data types. The following table maps the COBOL data types to their corresponding Ingres types. Note that the COBOL data type is determined by its category, picture, and usage.

Category	COBOL Type Picture	Usage	Ingres Type
ALPHABETIC	any	DISPLAY	character
ALPHANUMERIC	any	DISPLAY	character
ALPHANUMERICEDITED	any	DISPLAY	Character
NUMERIC	9(*p*) where *p* <=10	COMP DISPLAY	integer
NUMERIC	9(*p*)V9(*s*) where *p*+*s* <=9	COMP DISPLAY	float

	Category	COBOL Type Picture	Usage	Ingres Type
	NUMERIC	9(*p*) where *p* <=10	COMP-3	Integer
	NUMERIC	9(*p*) where *p* >10	COMP-3	decimal
	NUMERIC	9(*p*)V9(*s*)	COMP-3	decimal
	NUMERIC		INDEX	integer
	NUMERIC EDITED	any	DISPLAY	integer float
	NUMERIC		COMP-3	Decimal
VMS	NUMERIC	PACKED-DECIMAL	COMP-1	decimal
VMS	NUMERIC			float
VMS	NUMERIC		COMP-2	float

Because COBOL supports the packed decimal data type, the Ingres decimal type is mapped to it. In COBOL, the decimal data type is COMP-3. For example, the COBOL packed decimal declarations (where Pr = precision and Sc = scale):

```
01 PACK1 PIC S9(Pr-Sc)V9(Sc) USAGE COMP-3.
01 PACK2 PIC S9(Pr)   USAGE COMP-3.
```

correspond to the Ingres decimal types:

```
DECIMAL (Pr,Sc)
DECIMAL (Pr,0)
```

Note that Ingres precision includes scale, since it includes the total number of digits, and Ingres scale is the number of digits to the right of the decimal point.

The sign (S) is optional on a COBOL declaration and is ignored by the preprocessor. However, decimal values are always stored as signed by Ingres.

Note: You should always retrieve Ingres decimal data into a signed decimal variable.

COMP is an abbreviation for COMPUTATIONAL. You can use either form. Note that POINTER data items are not supported. The following sections describe the various data categories and the manner in which embedded SQL interacts with them.

Character strings containing embedded single quotes are legal in SQL, for example:

```
mary's
```

User variables may contain embedded single quotes and need no special handling unless the variable represents the entire search condition of a where clause:

```
where :variable
```

In this case you must escape the single quote by reconstructing the *:variable* string so that any embedded single quotes are modified to double single quotes, as in:

```
mary''s
```

Otherwise, a runtime error will occur. For more information on escaping single quotes, see String Literals in this chapter.

Alphabetic, Alphanumeric, and Alphanumeric Edited Categories

Embedded SQL accepts data declarations in the alphabetic, alphanumeric, and alphanumeric edited categories. The syntax for declaring data items in those categories is:

level-number data-name **PIC** [**IS**] *pic-string*

[[**USAGE** [**IS**]] **DISPLAY**].

Syntax Note: The *pic-string* can be any legal COBOL picture string for the alphabetic, alphanumeric, and alphanumeric edited classes. Embedded SQL notes only the length of the data item and that the data item is in the alphanumeric class.

You can use alphabetic, alphanumeric, and alphanumeric edited data items with any Ingres object of character (char or varchar) type. You can also use them to replace names of certain objects if the particular embedded SQL statement allows dynamic specification of object names. Note, however, that, when a value is transferred into a data item from an Ingres object, it is copied directly into the variable storage area without regard to the COBOL special insertion rules. When data in the database is in a different format from the alphanumeric edited picture, you must provide an extra variable to receive the data. You can then MOVE the data into the alphanumeric edited variable. However, if data in the database is in the same format as the alphanumeric edited picture (which would be the case, for example, if you had inserted data using the same variable you are retrieving into), you can assign the data directly into the edited data item, without any need for the extra variable. For more information on type conversion, see Data Type Conversion in this chapter.

The following example illustrates the syntax for these categories:

```
01 ENAME      PIC X(20).
01 EMP-CODE   PIC xx/99/00.
```

Indicator Data Items

An *indicator data item* is a 2-byte integer numeric data item. There are three ways to use these in an application:

- In a statement that retrieves data from Ingres, you can use an indicator variable to determine if its associated host variable was assigned a null value.
- In a statement that sets data to Ingres, you can use an indicator variable to assign a null to the database column, form field, or table field column.
- In a statement that retrieves character data from Ingres, you can use the indicator variable as a check that the associated host variable was large enough to hold the full length of the returned character string. You can use also use SQLSTATE to do this. Although you can also use SQLCODE as well, it is preferable to use SQLSTATE because SQLCODE is a deprecated feature.

An indicator variable declaration must have the following syntax:

level-number indicator-name **PIC** [**IS**] **S9(***p*) [**USAGE** [**IS**]] **COMP**

where *p* is less than or equal to 4.

The following is an example of an indicator declaration:

```
01 IND-VAR      PIC9(2) USAGE COMP.
01 IND-TABLE.
    02 IND-ARRAY  PIC S9(2) USAGE COMP OCCURS 10 TIMES.
```

When associating an indicator array (COBOL table) with a COBOL record, you must declare the indicator array as an array of 2-byte integers. In the example above, the data item IND-ARRAY can be used as an indicator array with a record assignment.

Numeric Edited Data Category

The syntax for a declaration of numeric edited data is:

level-number data-name **PIC** [**IS**] *pic-string* [[**USAGE** [**IS**]]**DISPLAY**]

Syntax Notes:

- The *pic-string* can be any legal COBOL picture string for numeric edited data. Embedded SQL notes only the type, scale, and size of the data item.

- To interact with Ingres integer-valued objects, the picture string must describe a maximum of 10 digit positions with no scaling.

While you can use numeric edited data items to assign data to, and receive data from, Ingres database tables and forms, be prepared for some loss of precision for numeric edited data items with scaling. The runtime interface communicates by integer (COMP) or uses packed (COMP-3) for UNIX or uses float (COMP-2) for VMS variables. In moving from these variables into your program's edited data items, truncation can occur due to MOVE statement rules and the COBOL standard alignment rules. For more information on type conversion, see Data Type Conversion in this chapter.

The following example illustrates the numeric edited data category:

```
01 DAILY-SALES   PIC $$$,$$9DB USAGE DISPLAY.
01 GROWTH-PERCENT  PIC ZZZ.9(3) USAGE DISPLAY.
```

The Numeric Data Category—Windows and UNIX

Embedded SQL/COBOL accepts the following declarations of numeric variables:

level-number data-name **PIC** [**IS**] *pic-string* [**USAGE** [**IS**]**COMP**|**COMP-3**
|**COMP-5**|**DISPLAY.**

level-number data-name [**USAGE** [**IS**]] **INDEX.**

Syntax Notes:

- Use the symbol S on numeric picture strings to indicate the presence of an operational sign.
- The picture string (*pic-string*) of a COMP, COMP-3, or COMP-5 data item can contain only the symbols 9, S, and V in addition to the parenthesized length.
- To interact with Ingres integer-valued objects, the picture string of a COMP, COMP-3, COMP-5, or DISPLAY item must describe a maximum of 10 digit positions with no scaling.
- Do not use a picture string for INDEX data items. While the preprocessor ignores such a picture string, the compiler does not allow it.

You can use any data items in the numeric category to assign and receive Ingres numeric data in database tables and forms. However, only use non-scaled COMP, COMP-3, COMP-5, and DISPLAY items of 10 digit positions or less to specify simple numeric objects, such as table field row numbers. Generally, try to use COMP data items with no scaling to interact with Ingres integer-valued objects, since the internal format of COMP data is compatible with Ingres integer data. Ingres effects the necessary conversions between all numeric data types, so the use of DISPLAY and COMP-3 scaled data items is allowed. For more information on type conversion, see Data Type Conversion in this chapter.

The following example contains numeric data categories:

```
01 QUAD-INTVAR  PIC  S9(10) USAGE COMP.
01 LONG-INTVAR  PIC  S9(9)  USAGE COMP.
01 SHORT-INTVAR PIC  S9(4)  USAGE COMP.
01 DISPLAY-VAR  PIC  S9(10) USAGE DISPLAY.
01 PACKED-VAR   PIC  S9(12)V9(4) USAGE COMP-3.
```

Numeric Data Items with Usage COMP-5—UNIX

Ingres supports data items declared with USAGE COMP-5. When you specify this clause, the data item is stored in the same machine storage format as the native host processor rather than in the byte-wise Micro Focus storage format. Of course, sometimes the two storage formats are identical. Since the Ingres runtime system that is linked into your COBOL runtime support module (RTS) is written in C, it is important that Ingres interact with native data types rather than Micro Focus data types. Consequently, many of your normal USAGE COMP data items are transferred (using COBOL MOVE statements) into internally declared Ingres USAGE COMP-5 data items. Data items declared with this USAGE cause a compiler information message (209 -I) to occur.

Dynamic SQL requires that your program point directly at result data items. In that case, you may be required to use USAGE COMP-5 data items, rather than having the option to use COMP or COMP-5. For details on dynamic SQL, see Dynamic Programming for COBOL in this chapter.

The Numeric Data Category—VMS

Embedded SQL accepts the following declarations of numeric variables:

level-number data-name **PIC** [**IS**] *pic-string* [**USAGE** [**IS**]]

COMP|**COMP-3**|**DISPLAY**|**PACKED-DECIMAL.**

level-number data-name [**USAGE** [**IS**]] **COMP-1**|**COMP-2**|**INDEX.**

Syntax Notes:

- The symbol S may be used on numeric picture strings to indicate the presence of an operational sign.

- The picture string (*pic-string*) of a COMP or COMP-3 data item can contain only the symbols 9, S, and V in addition to the parenthesized length.
- To interact with Ingres integer-valued objects, the picture string of a COMP, COMP-3 or DISPLAY item should describe a maximum of 10 digit positions with no scaling.
- A picture string must not be used for COMP-1, COMP-2, and INDEX data items. While such a picture string is ignored by the preprocessor, the compiler will not allow it.

Any data items in the numeric category may be used to assign and receive Ingres numeric data in database tables and forms. However, only non-scaled COMP, COMP-3, and DISPLAY items of 10 digit positions or less can be used to specify simple numeric objects, such as table field row numbers. Generally, you should use COMP data items with no scaling to interact with Ingres integer-valued objects, since the internal format of COMP data is compatible with Ingres integer data. Similarly, COMP-1 and COMP-2 data items are compatible with Ingres floating-point data. Although Ingres will effect the necessary conversions between all numeric data types, the use of DISPLAY and COMP-3 scaled data items could result in the loss of some precision. However, this does not occur if you are using COMP-3 to store decimals. For more information on type conversion, see Data Type Conversion in this chapter.

```
01 QUAD-INTVAR  PIC S9(10) USAGE COMP.
01 LONG-INTVAR  PIC S9(9)  USAGE COMP.
01 SHORT-INTVAR PIC S9(4)  USAGE COMP.
01 DISPLAY-VAR  PIC S9(10) USAGE DISPLAY.
01 SING-FLOATVAR USAGE COMP-1.
01 DOUB-FLOATVAR USAGE COMP-2.
01 PACKED-VAR PIC S9(12)V9(4) USAGE COMP-3.
```

Declaring Records

Embedded SQL accepts COBOL record and group declarations. To declare a record, use the following syntax:

```
01 data-name.
             record-item.
            {record-item.}
```

where *record-item* is a group item:

```
level-number data-name.
             record-item.
            {record-item.}
```

or an elementary item:

*level-number data-name elementary-item-description***.**

Syntax Notes:

- The record must have a level number of 01. Thereafter, the level numbers of *record-items* can be 02 through 49. Embedded SQL applies the same rules as the COBOL compiler in using the level numbers to order the groups and elementary items in a record definition into a hierarchical structure.
- If you do not specify *elementary-item-description* for a record item, the preprocessor and the COBOL compiler assume that the record item is a group item.
- The *elementary-item-description* can consist of any attributes described for data declarations in the Data Item Declaration Syntax section. The preprocessor does not confirm that the different clauses are acceptable for record items.
- The OCCURS clause, denoting a COBOL table, may appear on any record item.

The following example illustrates how to declare a record:

```
01 EMPTABLE.
  02 EMPREC OCCURS 25 TIMES.
      03 ENAME    PIC X(20).
      03 EADDRESS.
         04 ESTREET   PIC X(15).
         04 ECITY     PIC X(12).
         04 ESTATE    PIC X(2).
         04 EZIP      PIC X(5).
      03 ESALARY PIC S9(6) USAGE COMP.
```

DCLGEN Utility

DCLGEN (Declaration Generator) is a structure-generating utility that maps the columns of a database table into a *structure* (a COBOL record) that can be included in an embedded SQL declaration section.

The following command invokes DCLGEN from the operating system level:

dclgen *language dbname tablename filename structurename*

where

- *language* is the embedded SQL host language, in this case, cobol.
- *dbname* is the name of the database containing the table.
- *tablename* is the name of the database table.
- *filename* is the output file into which the structure declaration is placed.
- *structurename* is the name of the host language structure (COBOL record) that the command generates.

This command creates the declaration file *filename*, containing a record corresponding to the database table. The file also includes a declare table statement that serves as a comment and identifies the database table and columns from which the record was generated.

After the file has been generated, you can use an embedded SQL include statement to incorporate it into the embedded SQL variable declaration section. The following example demonstrates how to use DCLGEN in a COBOL program.

Assume the Employee table was created in the Personnel database as:

```
EXEC SQL CREATE TABLE employee
  (eno       integer NOT NULL,
   ename     char(20) NOT NULL,
   age       integer1,
   job       smallint,
   sal       decimal (14,2) NOT NULL,
   dept      smallint,
   vacation  float,
   resume    long varchar)
   END-EXEC.
```

and the DCLGEN system-level command is:

```
DCLGEN cobol personnel employee employee.dcl emprec
```

The employee.dcl file created by this command contains a comment and two statements. The first statement is the declare table description of employee, which serves as a comment. The second statement is a declaration of the COBOL emprec record.

The contents of the employee.dcl file are:

Windows **UNIX**

```
* Description of table "employee" from database * "personnel"

EXEC SQL DECLARE employee TABLE
(eno       integer NOT NULL,
 ename     char(20) NOT NULL,
 age       integer1,
 job       smallint,
 sal       decimal(14,2) NOT NULL,
 dept      smallint
 vacation  float,
 resume    long varchar)
END-EXEC.

01 EMPREC.
 02 ENO         PIC S9(9) USAGE COMP.
 02 ENAME       PIC X(20).
 02 AGE         PIC S9(5) USAGE COMP.
 02 JOB         PIC S9(5) USAGE COMP.
 02 SAL         PIC S9(12)V9(2) USAGE COMP-3.
 02 DEPT        PIC S9(5) USAGE COMP.
 02 VACATION    PIC S9(10)V9(8) USAGE COMP-3.
 02 RESUME      PIC X(0).
```

VMS

```
* Description of table "employee" from database * "personnel"

EXEC SQL DECLARE employee TABLE
(eno        integer NOT NULL,
 ename      char(20) NOT NULL,
 age        integer1,
 job        smallint,
 sal        decimal (14,2) NOT NULL,
 dept       smallint
 vacation   float,
 resume     long varchar)
END-EXEC.

01 EMPREC.
 02 ENO       PIC S9(9) USAGE COMP.
 02 ENAME     PIC X(20).
 02 AGE       PIC S9(5) USAGE COMP.
 02 JOB       PIC S9(4) USAGE COMP.
 02 SAL       PIC S9(12)V9(2) USAGE COMP-3.
 02 DEPT      PIC S9(4) USAGE COMP.
 02 VACATION  USAGE COMP-2.
 02 RESUME    PIC X(0).
```

Use the embedded SQL include statement, in an embedded SQL declaration section, to include this file as follows:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    EXEC SQL INCLUDE 'employee.dcl' END-EXEC.
EXEC SQL END DECLARE SECTION END-EXEC.
```

You can then use the emprec record in a select, fetch, or insert statement.

UNIX

The default generated picture string for Ingres floating-point data is S9(10)V9(8).

DCLGEN converts underscores in column names to dashes when it generates names of the elements of the COBOL record. For example, a column name of column_1 translates to a record element name of column-1. Column names that begin or end with an underscore thus generate record element names unacceptable to the COBOL compiler.

Since COBOL supports packed decimal data, the structure member's type will be packed decimal with a precision and scale that matches the scale and precision of the database column.

Both VMS and Micro Focus COBOL only allow a maximum precision of 18, otherwise a compiler error is generated. Ingres allows 39 precision. If the decimal column is greater than 18, DCLGEN displays a warning message and generates a COBOL variable of S9(10)V9(8). You must verify that this is an acceptable size for the decimal columns because if it's not, you must manually modify the DCLGEN output file.

The field names of the structure that DCLGEN generates are identical to the column names in the specified table. Therefore, if the column names in the table contain any characters that are illegal for host language variable names, you must modify the name of the field before attempting to use the variable in an application.

DCLGEN and Large Objects

When a table contains a large object column, DCLGEN will issue a warning message and map the column to a zero length character string variable. You must modify the length of the generated variable before attempting to use the variable in an application.

For example assume that the job_description table was created in the personnel database as:

```
create table job_description
          (job          smallint,
           description long varchar));
```

and the DCLGEN system-level command is:

```
dclgen cobol personnel job_description jobs.dcl jobs_rec
```

The contents of the jobs.dcl file would be:

```
* Description of the table “employee” from database “personnel”
    EXEC SQL DECLARE long_obj_table TABLE
                  (job          smallint,
                   description long varchar));
 01 JOBS_REC.
    02 JOB              PICTURE S9(4) USAGE COMP.
    02 DESCRIPTION      PICTURE X(0).
```

Compiling and Declaring External Compiled Forms

You can precompile your forms in the Visual Forms Editor (VIFRED). This saves the time otherwise required at runtime to extract the form's definition from the database forms catalogs. When you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in C. VIFRED prompts you for the name of the file with the C description. After the C file is created, you can use the following command to compile it into a linkable object module:

Windows

cl -c *filename*.c

UNIX

cc -c *filename*.c

This command produces an object file containing a global symbol with the same name as your form. Before the embedded SQL/FORMS statement addform can refer to this global object, you must use the following syntax to declare it in an embedded SQL declaration section:

01 *formname* **[IS] EXTERNAL PIC S9(9) [USAGE [IS]] COMP-5.**

Some platforms do not accept the above syntax. If EXTERNAL data items cannot be referenced in your COBOL program, see Including External Compiled Forms in the RTS in this chapter for an alternate procedure.

Syntax Notes:

- The *formname* is the actual name of the form. VIFRED gives this name to the global object. The *formname* is used to refer to the form in embedded SQL statements *after* the form has been made known to the FRS using the addform statement.
- The EXTERNAL clause causes the linker to associate the *formname* data item with the external *formname* symbol.

The following example shows a typical form declaration and illustrates the difference between using the form's global object definition and the form's name. However, currently, this example does not work on all Micro Focus platforms.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 empform IS EXTERNAL PIC S9(9) USAGE COMP-5.
* Other embedded SQL data declarations.
EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
* Program initialization.
* Making the form known to the FRS via the global
* form object.
EXEC FRS ADDFORM :empform END-EXEC.
* Displaying the form via the name of the form.
EXEC FRS DISPLAY empform END-EXEC.
* The program continues.
```

For information on linking your embedded SQL program with external compiled forms, see Including External Compiled Forms in the RTS in this chapter.

Assembling and Declaring External Compiled Forms—VMS

You can precompile your forms in VIFRED. This saves the time otherwise required at runtime to extract the form's definition from the database forms catalogs. When you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED prompts you for the name of the file with the MACRO description. When the MACRO file is created, you can use the following VMS command to assemble it into a linkable object module:

macro *filename*

This command produces an object file containing a global symbol with the same name as your form. Before the embedded SQL/FORMS statement addform can refer to this global object, it must be declared in an embedded SQL declaration section, with the following syntax:

01 *formid* **PIC S9(9)** [**USAGE** [**IS**]] **COMP VALUE** [**IS**] **EXTERNAL** *formname***.**

Syntax Notes:

- The *formid* is a COBOL data item. It is used with the addform statement to declare the form to the Forms Runtime System (FRS).
- The *formname* is the actual name of the form. VIFRED gives this name to the global object. The formname is used to refer to the form in embedded SQL statements *after* the form has been made known to the FRS *via* the addform statement.
- The EXTERNAL clause causes the VAX linker to associate the *formid* data item with the external formname symbol.

The example below shows a typical form declaration and illustrates the difference between using the form's object definition (the *formid*) and the form's name (the *formname*).

```
DATA DIVISION.
WORKING-STORAGE SECTION.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 EMPFORM-ID PIC S9(9) USAGE COMP VALUE IS EXTERNAL
   empform.
* Other embedded SQL data declarations.
EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
* Program initialization.
* Making the form known to the FRS via the global form object.
EXEC FRS ADDFORM :EMPFORM-ID END-EXEC.
* Displaying the form via the name of the form.
EXEC FRS DISPLAY empform END-EXEC.
* The program continues.
```

For information on linking your embedded SQL program with external compiled forms, see Assembling and Declaring External Compiled Forms—VMS in this chapter.

Concluding Example

The following UNIX, Windows, and VMS examples demonstrate some simple embedded SQL/COBOL declarations:

Windows

```
EXEC SQL INCLUDE SQLCA END-EXEC.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
```

UNIX

```
EXEC SQL INCLUDE SQLCA END-EXEC.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
```

```
* Data item to hold database name.
 01 DBNAME PIC X(9) VALUE IS "Personnel".

* Scaled data
 01 SALARY PIC S9(8)V9(2) USAGE COMP.
 01 MONEY  PIC S999V99 USAGE COMP-3.

* Array of numerics
 01 NUMS.
   02 NUM-ARR PIC S99 OCCURS 10 TIMES.

* Record of a full name and a redefinition of its parts.
 01 NAME-REC.
   02 FULL-NAME         PIC X(20).
   02 NAME-PARTS REDEFINES FULL-NAME.
      03 FIRST-NAME     PIC X(8).
      03 MIDDLE-INIT    PIC X(2).
      03 LAST-NAME      PIC X(10).

* Record for fetching and displaying.
 01 OUT-REC.
   02 FILLER    PIC X(15) VALUE "Value fetched: ".
   02 FROM-DB   PIC S9(4) USAGE DISPLAY.

* Miscellaneous attributes (ignored by preprocessor).
 01 SALES-TOT PIC S9(6)V99 SIGN IS TRAILING.
 01 SYNC-REC.
 02 NUM1     PIC S99 USAGE COMP SYNCHRONIZED.
 02 FILLER   PIC XX VALUE SPACES.
 02 NUM2     PIC S99 USAGE COMP SYNCHRONIZED.
01 RIGHT-ALIGN PIC X(30) JUSTIFIED RIGHT.
01 NUM-OUT PIC S99V99 USAGE DISPLAY BLANK WHEN ZERO.
EXEC SQL END DECLARE SECTION END-EXEC.
```

VMS

```
EXEC SQL INCLUDE SQLCA END-EXEC.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.

* Data item to hold database name.
 01 DBNAME PIC X(9) VALUE IS "Personnel".

* Scaled data
 01 SALARY USAGE COMP-1.
 01 MONEY  PIC S999V99 USAGE COMP-3.

* Array of numerics
 01 NUMS.
   02 NUM-ARR  PIC S99 OCCURS 10 TIMES.

* Record of a full name and a redefinition of its parts.
 01 NAME-REC.
   02 FULL-NAME PIC X(20).
   02 NAME-PARTS REDEFINES FULL-NAME.
      03 FIRST-NAME   PIC X(8).
      03 MIDDLE-INIT  PIC X(2).
      03 LAST-NAME    PIC X(10).

* Record for fetching and displaying.
 01 OUT-REC.
   02 FILLER       PIC X(15) VALUE "Value fetched: ".
   02 FROM-DB      PIC S9(4) USAGE DISPLAY.

* Miscellaneous attributes (ignored by preprocessor).
 01 SALES-TOT       PIC S9(6)V99 SIGN IS TRAILING.
```

```
 01 SYNC-REC.
   02 NUM1          PIC S99 USAGE COMP SYNCHRONIZED.
   02 FILLER        PIC XX VALUE SPACES.
   02 NUM2          USAGE COMP-2 SYNCHRONIZED.
 01 RIGHT-ALIGN     PIC X(30) JUSTIFIED RIGHT.
 01 NUM-OUT         PIC S99V99 USAGE DISPLAY BLANK
            WHEN ZERO.
EXEC SQL END DECLARE SECTION END-EXEC.
```

Scope of Variables

All variables declared in an embedded SQL declaration section can be referenced in ESQL statements and the preprocessor accepts them, from the point of declaration to the end of the file. This is not true for the COBOL compiler, which generally allows references to only those variables declared in the current program. Because the preprocessor does not terminate the scope of a variable in the same way the COBOL compiler does, do not redeclare variables of the same name to the preprocessor in a single file even where the variables are declared in separately compiled program units. If two programs in a single file each use variables of the same name and type in embedded SQL statements, only declare the first in an embedded SQL declaration section.

Variable Usage

COBOL variables (that is, data items) declared in an embedded SQL declaration section can substitute for most elements of embedded SQL statements that are not keywords. Of course, the variable and its data type must make sense in the context of the element. When you use a COBOL variable in an embedded SQL statement, you must precede it with a colon. As an example, the following select statement uses the data items NAMEVAR and NUMVAR to receive data and the data item IDNO as an expression in the where clause:

```
EXEC SQL SELECT ename, eno
    INTO :NAMEVAR, :NUMVAR
    FROM employee
    WHERE eno = :IDNO END-EXEC.
```

Various rules and restrictions apply to the use of COBOL variables in embedded SQL statements. The following sections describe the usage syntax of different categories of variables and provide examples of such use.

To distinguish the minus sign used as a subtraction operator in an embedded SQL statement from the hyphen used as a character in a data item name, you must delimit the minus sign by blanks. For example, the statement:

```
EXEC SQL INSERT INTO employee (ename, eno)
    VALUES ('Jones', :ENO-2)
    END EXEC.
```

indicates that the data item ENO-2 is to be inserted into the database column. To insert a value two less than the value in the data item ENO, you must instead use the following statement:

```
EXEC SQL INSERT INTO employee (ename, eno)
    VALUES ('Jones', :ENO - 2)
    END EXEC.
```

Note the spaces surrounding the minus sign.

Elementary Data Items

To refer to a simple scalar-valued data item (numeric, alphanumeric, or alphabetic), use the following syntax:

:*simplename*

The following program fragment demonstrates a typical error handling paragraph. The data items BUFFER and SECONDS are scalar-valued variables.

```
DATA DIVISION.
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

01 SECONDS PIC S9(4) USAGE COMP.
01 BUFFER  PIC X(100).

EXEC SQL END DECLARE SECTION END-EXEC.

* Program code

ERROR-HANDLE.

EXEC FRS MESSAGE :BUFFER END-EXEC.
EXEC FRS SLEEP :SECONDS END-EXEC.

*More error code.
```

COBOL Tables

To refer to a COBOL table, use the following syntax:

:*tablename***(***subscript*{**,***subscript*}**)**

Syntax Notes:

- You must subscript the tablename because only elementary data items are legal SQL values.

- When you declare a COBOL table, the preprocessor notes from the OCCURS clause that it is a table and not some other data item. When the table is later referenced in an ESQL statement, the preprocessor confirms that a subscript is present but does not check the legality of the subscript inside the parentheses. Consequently, you must ensure that the subscript is legal and that the correct number of subscripts is used.
- If you use COBOL tables as null indicator arrays with COBOL record assignments, do not include subscripts.

The following example uses the variable SUB1 as a subscript that does not need to be declared in the embedded SQL declaration section because the preprocessor ignores it.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 FORMNAMES.
  02 FORM-TABLE PIC X(8) OCCURS 3 TIMES.

EXEC SQL END DECLARE SECTION END-EXEC.

01 SUB1 PIC S9(4) USAGE COMP VALUE ZEROES.

PROCEDURE DIVISION.
BEGIN.

* Program code

PERFORM VARYING SUB1 FROM 1 BY 1
   UNTIL SUB1 > 3

EXEC FRS FORMINIT :FORM-TABLE(SUB1) END-EXEC

END-PERFORM.

* More program code.
```

Record Data Items

You can use a record data item (also known as a *structure variable*) in two different ways. First, you can use the record or a group item in the record as a simple variable, implying that all its elementary items (also known as *structure members*) are used. This is appropriate in the embedded SQL select, fetch, and insert statements. Second, you can refer to an elementary data item in the record alone.

Using a Record as a Collection of Variables

Use the following syntax to refer to a record or group item:

:{*groupname* ***IN*** | ***OF*** }*recordname*

Alternatively, you can use the following “dot” notation, in which the record or group item is specified from the outer level inwards:

:*recordname*{**.***groupname*}

Syntax Notes:

- The *recordname* can refer to either a record or a group item. It can be an element of a table of group items. Any reference that yields a record or group item is acceptable. For example:

```
* A record or unambiguous group item reference
      :EMPREC

* A group item in a table of group items
      :EMPREC-TABLE(SUB1)

* A group item subordinate to two group items
      :GROUP3 IN GROUP2 IN REC
      :REC.GROUP2.GROUP3
```

- To be used as a collection of variables, the record (or group item) referenced must have no subordinate groups or tables. The preprocessor enumerates all the elements of the record, which must be elementary items. The preprocessor generates code as though the program had listed each elementary item of the record in the order in which it was declared.

- The qualification of a record item can be elliptical; that is, you do not need to specify all the names in the hierarchy in order to reference the item. You must not, however, use an ambiguous reference that does not clearly qualify an item. For example, assume the following declaration:

```
01 PERSON.
  02 NAME.
      03 LAST  PIC X(18).
      03 FIRST PIC X(12).
  02 AGE     PIC S9(4) USAGE COMP.
  02 ADDR    PIC X(50).
```

  If the variable NAME was referenced, the preprocessor would assume the reference was to the group item NAME IN PERSON. However, if there also existed the declaration:

```
01 CHILD.
  02 NAME.
      03 LAST  PIC X(18).
      03 FIRST PIC X(12).
  02 PARENT   PIC X(30).
```

  the reference to NAME would be ambiguous, because it could refer to either NAME IN PERSON or NAME IN CHILD.

The following example uses the employee.dcl file, generated by DCLGEN, to retrieve values into a record.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.

* See above for description.
EXEC SQL INCLUDE 'employee.dcl' END-EXEC.

EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL SELECT *
INTO :EMPREC
FROM employee
WHERE eno = 123
END-EXEC.
```

The example above generates code as though the following statement had been issued instead:

```
EXEC SQL SELECT *
 INTO :ENO IN EMPREC, :ENAME IN EMPREC, :AGE IN EMPREC,
     :JOB IN EMPREC, :SAL IN EMPREC, :DEPT IN EMPREC
 FROM employee
 WHERE eno = 123
 END-EXEC.
```

The following example fetches the values associated with all the columns of a cursor into a record:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.

* See above for description.
EXEC SQL INCLUDE 'employee.dcl' END-EXEC.

EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL DECLARE empcsr CURSOR FOR
SELECT *
FROM employee
ORDER BY ename
END-EXEC.
...
EXEC SQL FETCH empcsr INTO :EMPREC END-EXEC.
```

The following example inserts values by looping through a locally declared table of records whose items have been initialized:

```
DATA DIVISION.
WORKING-STORAGE SECTION.

EXEC SQL INCLUDE SQLCA END-EXEC.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

EXEC SQL DECLARE person TABLE
(pname  char(30),
 page   integer1,
 paddr  varchar(50)) END-EXEC.

01 PERSON-REC.
 02 PERSON OCCURS 10 TIMES.
      03 NAME    PIC X(30).
      03 AGE     PIC S9(4) USAGE COMP.
      03 ADDR    PIC X(50).

EXEC SQL END DECLARE SECTION END-EXEC.

01 SUB1          PIC S9(4) USAGE COMP.

PROCEDURE DIVISION.
BEGIN.
```

```
* Initialization code.

PERFORM VARYING SUB1 FROM 1 TO 10
    UNTIL SUB1 > 10

EXEC SQL INSERT INTO person
    VALUES (:PERSON(SUB1))
    END-EXEC
END-PERFORM.
```

The insert statement in the example just shown generates code as though the following statement had been issued instead:

```
EXEC SQL INSERT INTO person
VALUES (:NAME IN PERSON(SUB1), :AGE IN PERSON(SUB1),  :ADDR IN PERSON(SUB1))
 END-EXEC
```

Using an Elementary Item from a Record

The syntax embedded SQL uses to refer to an elementary record item is the same as in COBOL:

:*elementary-item-name* **IN** | **OF**{ *groupname* **IN** | **OF**} *recordname*

Alternatively, you can use the following "dot" notation, in which the elementary item is specified from the outer level inwards:

:*recordname*{**.***groupname*}**.***elementary-item-name*

Syntax Notes:

- The referenced item must be a scalar value (numeric, alphanumeric, or alphabetic). There can be any combination of tables and records, but the last referenced item must be a scalar value. Thus, the following references are all legal:

```
* Element of a record
  :SAL IN EMPLOYEE
  :SAL OF EMPLOYEE
  :EMPLOYEE.SAL
* Element of a record as an item of a table
  :NAME IN PERSON(3)
  :PERSON(3).NAME
* Deeply nested element
  :ELEMENTARY-ITEM OF GROUP3 OF GROUP2 OF REC
  :REC.GROUP2.GROUP3.ELEMENTARY-ITEM
```

- The qualification of an elementary item in a record can be elliptical; that is, you do not need to specify all the names in the hierarchy in order to reference the item. You must not, however, use an ambiguous reference that does not clearly qualify an item. For example, assume the following declaration:

```
01 PERSON.
   02 NAME     PIC X(30).
   02 AGE      PIC S9(4) USAGE COMP.
   02 ADDR     PIC X(50).
```

If the variable NAME was referenced in your program, the preprocessor would assume the reference was to the elementary item NAME IN PERSON. However, if there also existed the declaration:

```
01 CHILD.
    02 NAME      PIC X(30).
    02 PARENT    PIC X(30).
```

the reference to NAME would be ambiguous because it could refer to either NAME IN PERSON or NAME IN CHILD.

- Subscripts, if present, must qualify the data item declared with the OCCURS clause.

The following example uses the record EMPREC in the employee.dcl file generated by DCLGEN to put values into the empform form:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.

* See above for description.
    EXEC SQL INCLUDE 'employee.dcl' END-EXEC.

EXEC SQL END DECLARE SECTION END-EXEC.

EXEC FRS PUTFORM empform
(eno = :ENO IN EMPREC, ename = :ENAME IN EMPREC,
 age = :AGE IN EMPREC, job   = :JOB IN EMPREC,
 sal = :SAL IN EMPREC, dept  = :DEPT IN EMPREC)
END-EXEC.
```

You could also write the putform statement without the EMPREC qualifications, assuming there are no ambiguous references to the item names:

```
EXEC FRS PUTFORM empform
     (eno = :ENO, ename = :ENAME, age = :AGE,
     job = :JOB, sal = :SAL, dept = :DEPT)
     END-EXEC.
```

Using Indicator Data Items

The syntax for referring to an *indicator* data item is the same as for an elementary data item, except that an indicator variable is always associated with another COBOL data item:

:data_item:indicator_item

or

:data_item **indicator** *:indicator_item*

Syntax Notes:

- The indicator data item can be an elementary data item or an element of a table that yields a 2-byte integer numeric data item. For example:

```
01 IND-1    PIC S9(4) USAGE COMP.
01 IND-TABLE.
  02 IND-2  PIC S9(4) USAGE COMP OCCURS 5 TIMES.

  :ITEM-1:IND-1
  :ITEM-2:IND-2(4)
```

- If the data item associated with the indicator data item is a record, the indicator data item must be a table of indicators. In this case, do *not* subscript the table (see the following example).
- When an indicator table is used, the first element of the table is associated with the first member of the record, the second element with the second member, and so on. Table elements begin at subscript 1.

The following example uses the employee.dcl file that DCLGEN generates, to retrieve values into a record and null values into the EMPIND table:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.

* See above for description.

EXEC SQL INCLUDE 'employee.dcl' END-EXEC.
01 INDS.
 02 EMPIND PIC S9(4) USAGE COMP OCCURS 10 TIMES.
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL SELECT *
 INTO :EMPREC:EMPIND
 FROM employee
 END-EXEC
```

The example just shown generates code as though the following statement had been issued:

```
EXEC SQL SELECT *
INTO :ENO IN EMPREC:EMPIND(1),
     :ENAME IN EMPREC:EMPIND(2),
     :AGE IN EMPREC:EMPIND(3),
     :JOB IN EMPREC:EMPIND(4),
     :SAL IN EMPREC:EMPIND(5),
     :DEPT IN EMPREC:EMPIND(6),
  FROM employee
  END-EXEC
```

Data Type Conversion

A COBOL data item must be compatible with the Ingres value it represents. Numeric Ingres values can be set by and retrieved into COBOL numeric and numeric edited items, and Ingres character values can be set by and retrieved into COBOL character data items, that is, alphabetic, alphanumeric, and alphanumeric edited items.

Data type conversion occurs automatically for different numeric types such as from floating-point Ingres database column values into integer (COMP) COBOL data items, and for different length character strings, such as from varying-length Ingres character fields into COBOL alphabetic and alphanumeric data items.

Ingres does not automatically convert between numeric and character types, such as from Ingres integer fields into COBOL alphanumeric data items. You must use the Ingres type conversion functions, the Ingres ascii function, or the COBOL STRING statement to effect such conversions.

The following table shows the default type compatibility for each Ingres data type in UNIX and VMS. Note that some COBOL types are omitted from the table because they do not exactly match an Ingres type. Use of those types necessitates some runtime conversion, which may possibly result in some loss of precision.

Windows **UNIX**

There is no exact match for float, so use COMP-3.

Ingres types and corresponding COBOL data types are listed in the following table:

Ingres Type	UNIX and Windows COBOL Types	VMS COBOL Type
char(N)	PIC X(N).	PIC X(N).
varchar(N)	PIC X(N).	PIC X(N).
integer1	PIC S9(2) USAGE COMP.	PIC S9(2) USAGE COMP.
smallint	PIC S9(4) USAGE COMP.	PIC S9(4) USAGE COMP.
integer	PIC S9(9) USAGE COMP.	PIC S9(9) USAGE COMP.
bigint	PIC S9(18) USAGE COMP*	PIC S9(18) USAGE COMP*
long varchar	PIC X(N).	PIC X(N).
float4	PIC S9(10)V9(8) USAGE COMP-3.	USAGE COMP-1.
float	PIC S9(10)V9(8) USAGE COMP-3.	USAGE COMP-2.
date	PIC X(25).	PIC X(25).
money	PIC S9(10)V9(8) USAGE COMP-3.	USAGE COMP-2.
table_key	PIC X(8).	PIC X(8).
object_key	PIC X(16).	PIC X(16).
decimal	PICS9(P-S)V(S) USAGE COMP-3.	PICS9(P-S)V(S) USAGE COMP-3.

*This type may not map to 8-byte integers with some COBOL compilers.

Note that Ingres stores decimal as signed. Thus, use a signed decimal variable if it interacts with an Ingres decimal type. Also, Ingres allows a maximum precision of 39 while COBOL allows only 18.

Decimal Type Conversion

An Ingres decimal value that will not fit into a COBOL variable will either be truncated if there is loss of scale or cause a runtime error if loss of significant digits.

Runtime Numeric Type Conversion

The Ingres runtime system provides automatic data type conversion between numeric-type values in the database and the forms system and numeric COBOL data items. It follows the standard COBOL type conversion rules. For example, if you assign the value in a scaled COMP-3 data item (UNIX and Windows) or COMP-1 data item (VMS) to an integer-valued field in a form, the digits after the decimal point of the data item's value are truncated. Runtime errors are generated for overflow on conversion.

The preprocessor generates COBOL MOVE statements or calls Ingres convert routines that convert various COBOL data types. These can again be converted at runtime by Ingres based on the final value being set or retrieved. The standard COBOL data conversion rules hold for all these generated MOVE statements, with a potential loss of precision.

Floats are coerced to decimal types by Ingres at runtime.

The preprocessor uses temporary data items when moving values between numeric DISPLAY data items and Ingres objects. Depending on the PICTURE clause of the DISPLAY item shown below, these temporary data items are either:

- **COMP-3** or 4-byte **COMP-5** (UNIX)

  or

- **COMP-2** or 4-byte **COMP** (VMS)

The following table lists numeric DISPLAY items and temporary data items:

Numeric DISPLAY Item's Picture	Temporary Item's Data Type—UNIX and Windows	Temporary Item's Data Type—VMS
With scaling	PIC S9(9)V9(9) USAGE COMP-3	COMP-2
With > 10 numeric digits	PIC S9(9)V9(9) USAGE COMP-3	Not applicable
No scaling and 10 numeric digits	4-byte COMP-5	4-byte COMP

COMP-3 items used to set or receive Ingres values also require some runtime conversion. This is not true if you are setting or receiving decimal data. This is true for Micro Focus COBOL when float values are received into COMP-3.

The preprocessor also generates code to use a temporary data item when Ingres data is to interact with a COBOL unscaled COMP data item whose picture string is exactly 10. Because a COBOL non-scaled numeric item whose picture contains 10 or fewer digits is regarded as compatible with the Ingres integer type, ESQL/COBOL assigns such data to a temporary COBOL 4-byte COMP-5 data item to allow it to interact with Ingres integer data. Note that the range of the Ingres i4 type does not include *all* 10-digit numbers. If you have 10-digit numeric data outside the Ingres range you, should use a COMP-3 (UNIX) or for VMS use COMP-1 or COMP-2 data item and choose the Ingres float type. Or with decimal you can use COMP-3 and choose a decimal Ingres type.

You can use only COMP data items or items that get assigned to temporary 4-byte COMP-5 (UNIX) or COMP (VMS) data items to set the values of Ingres integer objects, such as table field row numbers. You can, however, use any numeric data items to set and retrieve numeric values in Ingres database tables or forms.

Windows **UNIX**

The Ingres money type is represented as a COMP-3 data item.

VMS

The Ingres money type is represented as an 8-byte floating-point value, COMP-2.

Recall that a COBOL non-scaled numeric item with a picture that contains 10 or fewer digits is regarded as compatible with the Ingres integer type. (For details, see Variable and Type Declarations in this chapter.) However, the VAX standard data type for an unscaled 10-digit COMP item is a quadword (8 bytes). Therefore, ESQL/ COBOL assigns such data to a temporary COBOL 4-byte COMP data item to allow it to interact with Ingres integer data. Note that the range of the Ingres integer4 type does not include all 10-digit numbers.

Runtime Character and Varchar Type Conversion

Automatic conversion occurs between Ingres character string values and COBOL character variables (alphabetic, alphanumeric, and alphanumeric edited data items). The string-valued Ingres objects that can interact with character string variables are:

- Ingres names, such as form and column names
- Database columns of type character
- Database columns of type varchar
- Form fields of type character
- Database columns of type long varchar

Several considerations apply when dealing with character string conversions, both to and from Ingres.

The conversion of COBOL character variables used to represent Ingres names is simple: trailing blanks are truncated from the variables because the blanks make no sense in that context. For example, the string constants empform and empform refer to the same form.

The conversion of other Ingres objects is a bit more complicated. First, the storage of character data in Ingres differs according to whether the medium of storage is a database column of type character, a database column of type varchar, or a character form field. Ingres pads columns of type character with blanks to their declared length. Conversely, it does not add blanks to the data in columns of type varchar or long varchar, or in form fields.

Second, the COBOL convention is to blank-pad fixed-length character strings. For example, the character string abc may be stored in a COBOL PIC X(5) data item as the string abc followed by two blanks.

When character data is retrieved from a database column or form field into a COBOL character variable and the variable is longer than the value being retrieved, the variable is padded with blanks. If the variable is shorter than the value being retrieved, the value is truncated. You must always ensure that the variable is at least as long as the column or field, in order to avoid truncation of data. You should note that, when a value is transferred into a data item from an Ingres object, it is copied directly into the variable storage area without regard to the COBOL special insertion rules.

When inserting character data into an Ingres database column or form field from a COBOL variable, note the following conventions:

- When data is inserted from a COBOL variable into a database column of type character and the column is longer than the variable, the column is padded with blanks. If the column is shorter than the variable, the data is truncated to the length of the column.
- When data is inserted from a COBOL variable into a database column of type varchar or long varchar and the column is longer than the variable, no padding of the column takes place. Furthermore, by default, all trailing blanks in the data are truncated before the data is inserted into the varchar column. For example, when a string abc stored in a COBOL PIC X(5) data item as abc (see above) is inserted into the varchar column, the two trailing blanks are removed and only the string abc is stored in the database column. To retain such trailing blanks, you can use the Ingres notrim function. It has the following syntax:

  notrim(:_charvar_**)**

  where _charvar_ is a character variable. The following example demonstrates this feature. If the varchar column is shorter than the variable, the data is truncated to the length of the column.
- When data is inserted from a COBOL variable into a character form field and the field is longer than the variable, no padding of the field takes place. In addition, all trailing blanks in the data are truncated before the data is inserted into the field. If the field is shorter than the data (even after all trailing blanks have been truncated), the data is truncated to the length of the field.

  When comparing character data in an Ingres database column with character data in a COBOL variable, note the following convention:
- When comparing data in character or varchar database columns with data in a character variable, all trailing blanks are ignored. Initial and embedded blanks are significant.

Caution! *As described above, the conversion of character string data between Ingres objects and COBOL variables often involves the trimming or padding of trailing blanks, with resultant change to the data. If trailing blanks have significance in your application, give careful consideration to the effect of any data conversion. For a more complete description of the significance of blanks in string comparisons, see the SQL Reference Guide.*

The Ingres date data type is represented as a 25-byte character string: PIC X(25).

The program fragment in the following example demonstrates the notrim function and the truncation rules explained above.

```
DATA DIVISION.
WORKING-STORAGE SECTION.

EXEC SQL INCLUDE SQLCA END-EXEC.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

EXEC SQL DECLARE varychar TABLE
    (row integer,
     data varchar(10))
    END-EXEC.
01 ROW   PIC S9(4) USAGE COMP.
01 DATA  PIC X(7).
EXEC SQL END DECLARE SECTION END-EXEC.

PROCEDURE DIVISION.
BEGIN.

* DATA will hold "abc    " followed by 4 blanks.
    MOVE "abc    " TO DATA.

* The following INSERT adds the string "abc"
* (blanks truncated).
    EXEC SQL INSERT INTO varychar (row, data)
        VALUES (1, :DATA)
        END-EXEC.

* This statement adds the string "abc ", with 4 trailing
* blanks left intact by using the NOTRIM function

    EXEC SQL INSERT INTO varychar (row, data)
        VALUES (2, NOTRIM(:DATA))
        END-EXEC.

* This SELECT will retrieve the second row,
* because the NOTRIM
* function of the previous INSERT statement
* left trailing blanks in the "data" variable.

    EXEC SQL SELECT row
        INTO :ROW
        FROM varychar
        WHERE length(data) = 7
        END-EXEC.
    DISPLAY "Row found = " ROW.
```

The SQL Communications Area

This section describes the SQL communications area as implemented in COBOL.

The Include SQLCA Statement

You should issue the include sqlca statement in the Working-Storage Section of the Data Division of your COBOL program. For example:

```
DATA DIVISION.
WORKING-STORAGE SECTION.

EXEC SQL INCLUDE SQLCA END-EXEC.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    * declarations

EXEC SQL END DECLARE SECTION END-EXEC.
```

If you have multiple programs in a run unit, you must issue the include sqlca statement in each program.

The include sqlca statement instructs the preprocessor to generate code to call Ingres runtime libraries. It generates a COBOL COPY directive to make all the generated calls acceptable to the compiler.

Whether or not you intend to use the SQLCA for error handling, you must issue an include sqlca statement. If you do not issue it, the COBOL compiler will generate errors about undeclared data items in CALL statements.

Contents of the SQLCA

One of the results of issuing the include sqlca statement is the declaration of the SQLCA (SQL Communications Area) structure, which you can use for error handling in the context of database statements. You must only issue the statement once, because it generates a record declaration. The record declaration for the SQLCA is:

Windows

```
01 SQLCA.
    05 SQLCAID       PIC X(8).
    05 SQLCABC       PIC S9(9) USAGE COMP-5.
    05 SQLCODE       PIC S9(9) USAGE COMP-5.
    05 SQLERRM.
       10 SQLERRML   PIC S9(4) USAGE COMP-5.
       10 SQLERRMC   PIC X(70).
```

```
    05 SQLERRP       PIC X(8).
    05 SQLERRD       PIC S9(9) USAGE COMP-5
                     OCCURS 6 TIMES..
    05 SQLWARN.
      10 SQLWARN0  PIC X(1).
      10 SQLWARN1  PIC X(1).
      10 SQLWARN2  PIC X(1).
      10 SQLWARN3  PIC X(1).
      10 SQLWARN4  PIC X(1).
      10 SQLWARN5  PIC X(1).
      10 SQLWARN6  PIC X(1).
      10 SQLWARN7  PIC X(1).
    05 SQLEXT      PIC X(8).
```

UNIX

```
01 SQLCA.
    05 SQLCAID       PIC X(8).
    05 SQLCABC       PIC S9(9) USAGE COMP-5.
    05 SQLCODE       PIC S9(9) USAGE COMP-5.
    05 SQLERRM.
       10 SQLERRML   PIC S9(4) USAGE COMP-5.
       10 SQLERRMC   PIC X(70).
    05 SQLERRP       PIC X(8).
    05 SQLERRD       PIC S9(9) USAGE COMP-5
                     OCCURS 6 TIMES..
    05 SQLWARN.
      10 SQLWARN0  PIC X(1).
      10 SQLWARN1  PIC X(1).
      10 SQLWARN2  PIC X(1).
      10 SQLWARN3  PIC X(1).
      10 SQLWARN4  PIC X(1).
      10 SQLWARN5  PIC X(1).
      10 SQLWARN6  PIC X(1).
      10 SQLWARN7  PIC X(1).
    05 SQLEXT      PIC X(8).
```

VMS

```
01 SQLCA.
    05 SQLCAID       PIC X(8).
    05 SQLCABC       PIC S9(9) USAGE COMP.
    05 SQLCODE       PIC S9(9) USAGE COMP.
    05 SQLERRM.
        49 SQLERRML   PIC S9(4) USAGE COMP.
        49 SQLERRMC   PIC X(70).
    05 SQLERRP       PIC X(8).
    05 SQLERRD       PIC S9(9) USAGE COMP
                    OCCURS 6 TIMES.
```

```
05 SQLWARN.
  10 SQLWARN0 PIC X(1).
  10 SQLWARN1 PIC X(1).
  10 SQLWARN2 PIC X(1).
  10 SQLWARN3 PIC X(1).
  10 SQLWARN4 PIC X(1).
  10 SQLWARN5 PIC X(1).
  10 SQLWARN6 PIC X(1).
  10 SQLWARN7 PIC X(1).
05 SQLEXT     PIC X(8).
```

For a full description of the SQLCA data items, see the *SQL Reference Guide.*

The SQLCA is initialized at load-time. The fields SQLCAID and SQLCABC are initialized to the string SQLCA and the constant 136, respectively.

Note that the preprocessor is not aware of the record declaration. Therefore, you cannot use the record items in an embedded SQL statement. For example, the following statement, which attempts to insert the string SQLCA into a table, generates an error:

```
* This statement is illegal
    EXEC SQL INSERT INTO employee (ename)
         VALUES (:SQLCAID);
```

Windows **UNIX**

The SQLCA is local to the program that issued the include sqlca statement.

VMS

All modules from different languages that are linked together share the same SQLCA.

Using the SQLCA for Error Handling

User-Defined Error, Message, and DBevent Handlers offer the most flexibility for handling errors, database procedure messages, and database events. For more information, see Advanced Processing in this chapter.

However you can do error handling with the SQLCA by using whenever statements or explicitly by checking the contents of the SQLCA fields SQLCODE, SQLERRD, and SQLWARN0.

Error Handling with the Whenever Statement

The syntax of the whenever statement is:

exec sql whenever *condition action* **end-exec**

Condition is sqlwarning, sqlerror, sqlmessage, dbevent, or not found.

Action is continue, stop, goto a COBOL paragraph name, or call a COBOL paragraph name. The call action causes the preprocessor to generate a COBOL PERFORM statement for the specified paragraph name. For a detailed description of the whenever statement, see the *SQL Reference Guide.*

If the paragraph name in a goto or call action is an embedded SQL reserved word, specify it in quotes. The paragraph name targeted by the goto or call action must be in the scope of all subsequent embedded SQL statements until another whenever statement is encountered for the same action. This is necessary because when the preprocessor interprets a whenever goto statement, it generates the COBOL statement:

IF (*condition***) THEN**
        GO TO *paragraph_name*
END-IF

after an embedded SQL statement. Similarly, in interpreting a whenever call statement, the preprocessor generates the COBOL statement:

IF (*condition***) THEN**
        PERFORM *paragraph_name*
END-IF

after subsequent embedded SQL statements. If the paragraph name is invalid, the COBOL compiler generates an error.

You can also use user-defined handlers for error handling. For more information, see the *SQL Reference Guide*. Note that the reserved procedure sqlprint, which can substitute for a paragraph name in a whenever call statement, is always in the scope of the program.

When the *condition* specified for a call action occurs, control passes to the first statement in the named paragraph. After the last statement contained in the paragraph has been executed, control returns to the statement following the statement that caused the call to occur. Consequently, after handling the whenever condition in the called paragraph, you may want to take some action, instead of merely allowing execution to continue with the statement following the embedded SQL statement that generated the error.

The following example demonstrates use of the whenever statements in the context of printing some values from the employee table. The comments do not relate to the program but to the use of error handling.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
EXEC SQL INCLUDE SQLCA END-EXEC.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
     01 E-REC.
         02 ENO      PIC S9(8) USAGE DISPLAY.
         02 FILLER   PIC X(2) VALUE SPACES.
         02 ENAME    PIC X(20).
         02 AGE      PIC S9(4) USAGE DISPLAY.
      01 ERRMSG      PIC X(200).
```

```
EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
BEGIN.
        EXEC SQL DECLARE empcsr CURSOR FOR
            SELECT eno, ename, age
            FROM employee
            END-EXEC.
* An error when opening the "personnel" database will
* cause the error to be printed and the program to
* abort.
        EXEC SQL WHENEVER SQLERROR STOP END-EXEC.
        EXEC SQL CONNECT personnel END-EXEC.
* Errors from here on will cause the program to clean up
         EXEC SQL WHENEVER SQLERROR
             GOTO CLEAN-UP END-EXEC.
         EXEC SQL OPEN empcsr END-EXEC.
         DISPLAY "Some values from
                      the ""employee"" table".
* When no more rows are fetched, close the cursor
        EXEC SQL WHENEVER NOT FOUND GOTO CLOSE-CSR
            END-EXEC.
* The last statement was an OPEN, so we know that the
* value of SQLCODE cannot be SQLERROR or NOT FOUND.
* Loop is broken by NOT FOUND
        PERFORM UNTIL SQLCODE NOT = 0
            EXEC SQL FETCH empcsr
                INTO :ENO, :ENAME, :AGE END-EXEC
* The DISPLAY does not execute after the previous FETCH * returns the NOT FOUND
condition.
            DISPLAY E-REC
        END-PERFORM.
* From this point in the file onwards, ignore all
* errors.  Also, turn off the NOT FOUND condition,
* for consistency.
        EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
        EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
    CLOSE-CSR.
        EXEC SQL CLOSE empcsr END-EXEC.
        EXEC SQL DISCONNECT END-EXEC.
        STOP RUN.
    CLEAN-UP.
        EXEC SQL INQUIRE_SQL(:ERRMSG = ERRORTEXT)
            END-EXEC.
        DISPLAY "Aborting because of error".
        DISPLAY ERRMSG.
        EXEC SQL DISCONNECT END-EXEC.
        STOP RUN.
```

The Whenever Goto Action In Embedded SQL Blocks

The words begin and end delimit an embedded SQL block-structured statement is a statement. For example, the select loop and the unloadtable loops are both block-structured statements. You can only terminate these statements using the methods specified for the particular statement in the *SQL Reference Guide.* For example, the select loop is terminated either when all the rows in the database result table have been processed or by an endselect statement. The unloadtable loop is terminated either when all the rows in the forms table field have been processed or by an endloop statement.

Therefore, if you use a whenever statement with the goto action in an SQL block, you must avoid going to a paragraph outside the block. Such a goto causes the block to be terminated without issuing the runtime calls necessary to clean up the information that controls the loop. (For the same reason, you must not issue a COBOL GO TO statement that causes control to leave or enter the middle of an SQL block.) The target of the whenever goto statement must be a paragraph in the block. If, however, it is a paragraph containing a block of code that cleanly exits the program, you do not need to take the above precaution.

The above information does not apply to error handling for database statements issued outside an SQL block, or to explicit hard-coded error handling. For an example of hard-coded error handling, see the The Table Editor Table Field Application in this chapter.

Explicit Error Handling

The program can also handle errors by inspecting values in the SQLCA structure at various points. For additional information, see the *SQL Reference Guide*.

The following example is functionally the same as the previous example, except that the error handling is hard-coded in COBOL statements.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
EXEC SQL INCLUDE SQLCA END-EXEC.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  01 E-REC.
     02 ENO         PIC S9(8) USAGE DISPLAY.
     02 ENAME       PIC X(20).
     02 AGE         PIC S9(4) USAGE DISPLAY.
  01 NOT-FOUND      PIC S9(4) USAGE COMP VALUE 100.
  01 REASON         PIC X(14).
  01 ERRMSG         PIC X(100).
EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
BEGIN.
    EXEC SQL DECLARE empcsr CURSOR FOR
        SELECT eno, ename, age
        FROM employee
        END-EXEC.
* Exit if database cannot be opened
    EXEC SQL CONNECT personnel END-EXEC.
    IF SQLCODE < 0 THEN
        DISPLAY "Cannot access database"
        STOP RUN.
* Error if cannot open cursor
    EXEC SQL OPEN empcsr END-EXEC.
    IF SQLCODE < 0 THEN
        MOVE "OPEN ""empcsr""" TO REASON
        PERFORM CLEAN-UP.
    DISPLAY "Some values from the ""employee"" table".
* The last statement was an OPEN, so we know that the
* value of SQLCODE cannot be SQLERROR or NOTFOUND.
    PERFORM UNTIL SQLCODE NOT = 0
        EXEC SQL FETCH empcsr
               INTO :ENO, :ENAME, :AGE
```

```
              END-EXEC.

        IF SQLCODE < 0 THEN
            MOVE "FETCH ""empcsr""" TO REASON
            PERFORM CLEAN-UP
* Do not print the last values twice
        ELSE
            IF SQLCODE NOT = NOT-FOUND THEN
                DISPLAY E-REC
            END-IF
        END-IF
    END-PERFORM.
    EXEC SQL CLOSE empcsr END-EXEC.
    EXEC SQL DISCONNECT END-EXEC.
    STOP RUN.
  CLEAN-UP.
* Error handling paragraph
    DISPLAY "Aborting because of error in " REASON.
    EXEC SQL INQUIRE_SQL(:ERRMSG = ERRORTEXT) END-EXEC.
    DISPLAY ERRMSG.
    EXEC SQL DISCONNECT END-EXEC.
    STOP RUN.
```

Determining the Number of Affected Rows

The SQLCA variable SQLERRD(3) indicates how many rows were affected by the last insert, update, or delete statement. The following program fragment, which deletes all employees whose employee numbers are greater than a given number, demonstrates how SQLERRD is used:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
EXEC SQL BEGIN DECLARE SECTION.
  01 LOWER-BOUND-NUM PIC S9(8) USAGE COMP.
EXEC SQL END DECLARE SECTION.
01 SQLERRD-DISP PIC Z9(4) USAGE DISPLAY.
PROCEDURE DIVISION.
BEGIN.
    ...
    EXEC SQL DELETE FROM employee
        WHERE eno > :LOWER-BOUND-NUM
        END-EXEC.
* Print the number of employees deleted
    MOVE SQLERRD(3) TO SQLERRD-DISP.
    DISPLAY SQLERRD-DISP " rows were deleted."
    ...
```

Using the SQLSTATE Variable

You can use the SQLSTATE variable in an ESQL/COBOL program to return status information about the last SQL statement that was executed. SQLSTATE must be declared in a DECLARE SECTION and its declaration must be valid for the entire file being preprocessed.

To declare this variable, use:

```
01 SQLSTATE     PICTURE X(5).
```

or :

```
77 SQLSTATE     PICTURE X(5).
```

Dynamic Programming for COBOL

Ingres provides Dynamic SQL and Dynamic FRS to allow you to write generic programs. Dynamic SQL allows a program to build and execute SQL statements at runtime. For example, an application can include an expert mode in which the runtime user can type in select queries and browse the results at the terminal. Dynamic FRS allows a program to interact with any form at runtime. For example, an application can load in any form, allowing the runtime user to retrieve new data from the form and insert it into the database.

The Dynamic SQL and Dynamic FRS statements are described in the *SQL Reference Guide* and the *Forms-based Application Development Tools User Guide*. This section discusses the COBOL-dependent issues of dynamic programming. For a complete example of using Dynamic SQL to write an SQL Terminal Monitor application, see The SQL Terminal Monitor Application in this chapter. For an example of using both Dynamic SQL and Dynamic FRS to browse and update a database using any form, see A Dynamic SQL/Forms Database Browser in this chapter.

Windows The Windows examples in this section are written exclusively for Micro Focus COBOL Windows and make use of the MF extensions to the COBOL language, in particular the POINTER usage clause.

UNIX The UNIX examples in this section are written exclusively for Micro Focus COBOL II and make use of the MF extensions to the COBOL language, in particular the POINTER usage clause.

VMS The VMS examples in this section make use of the VMS extensions to the COBOL language, in particular the POINTER usage clause.

The SQLDA Record

You can use the SQLDA SQL Descriptor Area (SQLDA) to pass type and size information about an SQL statement, an Ingres form, or an Ingres table field, between Ingres and your program.

In order to use the SQLDA, issue the include sqlda statement in the COBOL program units that reference the SQLDA. The include sqlda statement generates a COBOL COPY directive of a file that defines an external reference to an SQLDA-like COBOL record. The file declares a COBOL record called SQLDA. Additionally in VMS, it marks it as EXTERNAL.

You can also code this record directly, instead of using the include sqlda statement. You can choose any name for the structure and you can declare more than one in a single program.

The definition of the SQLDA (as specified in the COPY file) is shown below in Windows, UNIX and VMS:

Windows

```
*
* SQL Descriptor Area
*
 78 IISQ-MAX-COLS    VALUE 1024.

01 SQLDA.
   05 SQLDAID         PIC X(8).
   05 SQLDABC         PIC S9(9) USAGE COMP-5.
   05 SQLN            PIC S9(4) USAGE COMP-5
                      VALUE IISQ-MAX-COLS.
   05 SQLD            PIC S9(4) USAGE COMP-5.
   05 SQLVAR          OCCURS IISQ-MAX-COLS TIMES.
      07 SQLTYPE      PIC S9(4) USAGE COMP-5.
      07 SQLLEN       PIC S9(4) USAGE COMP-5.
      07 SQLDATA      USAGE POINTER.
      07 SQLIND       USAGE POINTER.
      07 SQLNAME.
          49 SQLNAMEL PIC S9(4) USAGE COMP-5.
          49 SQLNAMEC PIC X(34).
*
* SQLDA Type Codes
* Type Name  Value  Length
* ---------  -----  ------
* DATE          3     25
* MONEY         5      8
* DECIMAL      10   SQLLEN = 256*P+S
* CHAR         20   SQLLEN
* VARCHAR      21   SQLLEN
* BYTE         23   SQLLEN
* BYTE VARYING 24   SQLLEN
* LONG BYTE    25   SQLLEN
* INTEGER      30   SQLLEN
* FLOAT        31   SQLLEN
* 4GL OBJECT   45   SQLLEN
* TABLE-FIELD  52   0
*
```

```
 78 IISQ-DTE-TYPE   VALUE 3.
 78 IISQ-DTE-LEN    VALUE 25.
 78 IISQ-MNY-TYPE   VALUE 5.
 78 IISQ-DEC-TYPE   VALUE 10.
 78 IISQ-CHA-TYPE   VALUE 20.
 78 IISQ-VCH-TYPE   VALUE 21.
 78 IISQ-BYTE-TYPE  VALUE 23.
 78 IISQ-VBYTE-TYPE VALUE 24.
 78 IISQ-LBYTE-TYPE VALUE 25.
 78 IISQ-INT-TYPE   VALUE 30.
 78 IISQ-FLT-TYPE   VALUE 31.
 78 IISQ-OBJ-TYPE   VALUE 45.
 78 IISQ-TBL-TYPE   VALUE 52.
 78 IISQ-LVCH-TYPE  VALUE 22.
```

UNIX

```
*
* SQL Descriptor Area
*
 78 IISQ-MAX-COLS    VALUE 1024.

01 SQLDA.
   05 SQLDAID          PIC X(8).
   05 SQLDABC          PIC S9(9) USAGE COMP-5.
   05 SQLN             PIC S9(4) USAGE COMP-5
                       VALUE IISQ-MAX-COLS.
   05 SQLD             PIC S9(4) USAGE COMP-5.
   05 SQLVAR           OCCURS IISQ-MAX-COLS TIMES.
      07 SQLTYPE       PIC S9(4) USAGE COMP-5.
      07 SQLLEN        PIC S9(4) USAGE COMP-5.
      07 SQLDATA       USAGE POINTER.
      07 SQLIND        USAGE POINTER.
      07 SQLNAME.
          49 SQLNAMEL PIC S9(4) USAGE COMP-5.
          49 SQLNAMEC PIC X(34).
*
* SQLDA Type Codes
* Type Name  Value  Length
* ---------  -----  ------
* DATE          3     25
* MONEY         5      8
* DECIMAL      10   SQLLEN = 256*P+S
* CHAR         20   SQLLEN
* VARCHAR      21   SQLLEN
* BYTE         23   SQLLEN
* BYTE VARYING 24   SQLLEN
* LONG BYTE    25   SQLLEN
* INTEGER      30   SQLLEN
* FLOAT        31   SQLLEN
* 4GL OBJECT   45   SQLLEN
* TABLE-FIELD  52   0
*
 78 IISQ-DTE-TYPE   VALUE 3.
 78 IISQ-DTE-LEN    VALUE 25.
 78 IISQ-MNY-TYPE   VALUE 5.
 78 IISQ-DEC-TYPE   VALUE 10.
 78 IISQ-CHA-TYPE   VALUE 20.
 78 IISQ-VCH-TYPE   VALUE 21.
 78 IISQ-BYTE-TYPE  VALUE 23.
 78 IISQ-VBYTE-TYPE VALUE 24.
 78 IISQ-LBYTE-TYPE VALUE 25.
 78 IISQ-INT-TYPE   VALUE 30.
```

```
78 IISQ-FLT-TYPE   VALUE 31.
78 IISQ-OBJ-TYPE   VALUE 45.
78 IISQ-TBL-TYPE   VALUE 52.
78 IISQ-LVCH-TYPE  VALUE 22.
```

VMS

```
*
* SQL Descriptor Area
*
  01 SQLDA EXTERNAL.
     05 SQLDAID          PIC X(8).
     05 SQLDABC          PIC S9(9) USAGE COMP.
     05 SQLN             PIC S9(4) USAGE COMP.
     05 SQLDA            PIC S9(4) USAGE COMP.
     05 SQLVAR           OCCURS 1024 TIMES.
          07 SQLTYPE     PIC S9(4) USAGE COMP.
          07 SQLLEN      PIC S9(4) USAGE COMP.
          07 SQLDATE     USAGE POINTER.
          07 SQLIND      USAGE POINTER.
          07 SQLNAME.
              49 SQLNAMEL PIC S9(4) USAGE COMP.
              49 SQLNAMEC PIC X(34).
  01 IISQLHDR
     05 SQLARG          USAGE POINTER.
     05 SQLHDLR         PIC S9(9) USAGE COMP.
*
* SQLDA Type Codes
*
* Type Name   Value   Length
* ---------   -----   ------
* DATE          3      25
* MONEY         5       8
* DECIMAL      10     SQLLEN = 256*P+S
* CHAR         20     QLLEN
* VARCHAR      21     SQLLEN
* BYTE         23     SQLLEN
* BYTE VARYING 24     SQLLEN
* LONG BYTE    25     SQLLEN
* INTEGER      30     SQLLEN
* FLOAT        31     SQLLEN
* TABLE        52       0
* LONG VARCHAR 22       0
* 4GL OBJECT   45     SQLLEN
* DATAHANDLER  46
```

Structure Definition and Usage Notes:

- The sqlvar array (COBOL table) has 1024 elements. If you code your own SQLDA, you can supply a different number of elements.
- The sqlvar array begins at subscript 1.
- The sqldata and sqlind fields are declared with USAGE POINTER. These must be set to point at the addresses of other data items using the COBOL SET statement with the ADDRESS OF clause (UNIX) or the REFERENCE clause (VMS).
- If your program declares its own SQLDA record, you must confirm that the record layout is identical to that of the Ingres-defined SQLDA record, although you can declare a different number of sqlvar elements.

- The nested group sqlname is a varying length character string consisting of a length and data area. The sqlnamec field contains the name of a result field or column after the describe (or prepare into) statement. The length of the name is specified by sqlnamel. The characters in the sqlnamec field are blank padded. The sqlname group may also be set by a program using Dynamic FRS. The program is not required to pad sqlnamec with blanks. (See Setting SQLNAME for Dynamic FRS in this chapter.)
- The comment listing the type codes represents the types that are returned by the describe statement and the types used by the program when using an SQLDA to retrieve or set data. The type code 52 indicates a table field and is set by the FRS when describing a form that contains a table field.

Windows **UNIX**

- If you code your own SQLDA, you can declare it EXTERNAL and share it with other programs.
- Because the SQLDA is passed directly to Ingres without preprocessor intervention (and generated MOVE statements), all numeric fields of the SQLDA are declared as COMP-5. If, on your machine, the internal storage format of a USAGE COMP data item is identical to the storage format of USAGE COMP-5 then you may use either USAGE COMP or USAGE COMP-5 for the corresponding SQLDA fields when you code your own. If you use USAGE COMP and the internal storage format is not the same then Ingres issues runtime errors about unknown data type codes and invalid data type lengths.

VMS

- The SQLDA record definition is an EXTERNAL definition. This allows multiple COBOL program modules and source files to reference and process the same SQLDA. If you code your own SQLDA, you are not required to share it with other program modules by declaring it EXTERNAL.

Declaring the SQLDA Record

To declare the SQLDA record, issue include sqlda or hard code the record as previously defined. This declaration must be in the Working-Storage Section of the COBOL Data Division but not in an SQL declare section because the preprocessor does not understand the special meaning of the fields of the SQLDA. When the SQLDA record is used, the preprocessor accepts any object name and assumes that the data item refers to a legally declared SQLDA record.

If a program requires an SQLDA with the same number of sqlvar elements as in the Ingres definition, it can accomplish this by including the following line in the Working-Storage Section:

```
EXEC SQL INCLUDE SQLDA END-EXEC.
```

and by including the following lines in the Procedure Division:

```
* Set the size of the SQLDA
MOVE 1024 to SQLN.
```

```
...
EXEC SQL DESCRIBE s1 INTO :SQLDA END-EXEC.
```

Note that the sqln is given an initial value of 1024.

If a program requires another SQLDA record or an SQLDA with a different number of sqlvar elements (not 1024), it can declare its own COBOL record. For example:

Windows

```
* In Working-Storage Section.

01 MY-SQLDA EXTERNAL.
  02 MY-SQID                PIC X(8).
  02 MY-SQSIZE              PIC S9(9) USAGE COMP-5.
  02 MY-VARS                PIC S9(4) USAGE COMP-5.
  02 RESULT-VARS            PIC S9(4) USAGE COMP-5.
  02 COLUMN-VARS            OCCURS 20 TIMES.
      03 COL-TYPE           PIC S9(4) USAGE COMP-5.
      03 COL-LEN            PIC S9(4) USAGE COMP-5.
      03 COL-ADDR           USAGE POINTER.
      03 COL-NULL           USAGE POINTER.
      03 COL-NAME.
          04 NAME-LEN       PIC S9(4) USAGE COMP-5.
          04 NAME-DAT       PIC X(34).

* In Procedure Division set the size of the SQLDA

MOVE 20 to MY-VARS.
```

UNIX

```
* In Working-Storage Section.

01 MY-SQLDA EXTERNAL.
  02 MY-SQID                PIC X(8).
  02 MY-SQSIZE              PIC S9(9) USAGE COMP-5.
  02 MY-VARS                PIC S9(4) USAGE COMP-5.
  02 RESULT-VARS            PIC S9(4) USAGE COMP-5.
  02 COLUMN-VARS            OCCURS 20 TIMES.
      03 COL-TYPE           PIC S9(4) USAGE COMP-5.
      03 COL-LEN            PIC S9(4) USAGE COMP-5.
      03 COL-ADDR           USAGE POINTER.
      03 COL-NULL           USAGE POINTER.
      03 COL-NAME.
          04 NAME-LEN       PIC S9(4) USAGE COMP-5.
          04 NAME-DAT       PIC X(34).

* In Procedure Division set the size of the SQLDA

MOVE 20 to MY-VARS.
```

VMS

```
* In Working-Storage Section.

01 MY-SQLDA EXTERNAL.
   02 MY-SQID             PIC X(8).
   02 MY-SQSIZE           PIC S9(9) USAGE COMP.
   02 MY-VARS             PIC S9(4) USAGE COMP.
   02 RESULT-VARS         PIC S9(4) USAGE COMP.
```

```
   02 COLUMN-VARS OCCURS 20 TIMES.
       03 COL-TYPE      PIC S9(4) USAGE COMP.
       03 COL-LEN       PIC S9(4) USAGE COMP.
       03 COL-ADDR      USAGE POINTER.
       03 COL-NULL      USAGE POINTER.
       03 COL-NAME.
            04 NAME-LEN PIC S9(4) USAGE COMP.
            04 NAME-DAT PIC X(34).

* In Procedure Division set the size of the SQLDA

MOVE 20 to MY-VARS.
```

In the above declaration the names of the record components are not the same as those of the SQLDA record, but their layout is identical.

Using the SQLVAR Table

The *SQL Reference Guide* discusses the legal values of the sqlvar table (array). The describe and prepare into statements set the type, length, and name information of the SQLDA. This information refers to the result columns of a prepared select statement, the fields of a form, or the columns of a table field. When the program uses the SQLDA to retrieve or set Ingres data, it must assign the type and length information that now refers to the data items being pointed at by the SQLDA.

COBOL Data Item Type Codes

The type codes listed in the COBOL comment appearing in the the SQLDA Record section are the types that describe Ingres result fields or columns. For example, the SQL types date and money do not describe program variables but rather data types that are compatible with COBOL character and numeric types.

Character data and the SQLDA have the same rules as character data in regular embedded SQL statements. They are also described in the COBOL Data Items and Data Types section.

The following Windows, UNIX, and VMS sections describe the Ingres type codes to use with COBOL data items that will be pointed at by the sqldata pointers.

COBOL Type Codes and Ingres Type Codes—Windows and UNIX

The left column of the following table shows the COBOL pictures and usages of the COBOL data items pointed at by sqldata, while the middle and the right columns show the equivalent SQL type codes and lengths:

COBOL Data Type	SQL Type Code (sqltype)	Length (sqllen)
PIC S9(4) USAGE COMP-5 SYNC	30 (INTEGER)	2
PIC S9(9) USAGE COMP-5 SYNC	30 (INTEGER)	4
PIC S9(10)V9(8) COMP-3 SYNC	31 (FLOAT)	256*18+8
PIC S9(P-S)V9(S) USAGE COMP-3	10 (DECIMAL)	256*P+S
PIC X(LEN)	20 (CHARACTER)	LEN

First, note that since the preprocessor does not generate any conversions for the data items pointed at by sqldata you must confirm that the storage format of the values being pointed at are completely compatible with the storage formats known by Ingres (C storage formats). Consequently, 4-byte integers are USAGE COMP-5 SYNC rather than just USAGE COMP. If, on your machine, you verify that the internal storage format of unscaled COMP and COMP-5 data items are identical then you can use USAGE COMP.

The preprocessor does not need to generate any conversions for the decimal data type. So, sqldata can be pointed directly at a COMP-3. Ingres expects precision and scale to be encoded in the sqllen field. The precision is stored in the first byte of a 2-byte integer while scale is stored in the last byte of a 2-byte integer. For example, decimal(18,8) length is stored as (256*18)+8.

All other Ingres types are compatible with the above types. For more information, see COBOL Data Items and Data Types in this chapter, which describes runtime data conversion. For example, the SQL date data type can be retrieved into a 25-byte character string, while the SQL money or float data type can be retrieved using a COMP-3 data item. Ingres will coerce float or money to packed decimal at runtime.

Nullable data types (those data items associated with a null indicator) are specified by assigning the negative of the type code to the sqltype field. If the type is negative when you use the SQLDA to retrieve or set Ingres data, then a null indicator must be pointed at by the corresponding sqlind field. In this case, the COBOL data type of the null indicator must be PIC S9(4) USAGE COMP-5 SYNC. Once again, USAGE COMP-5 may be replaced by USAGE COMP if you verify that COMP is identical to COMP-5 on your machine.

COBOL Type Codes and Ingres Type Codes—VMS

The left column of the following table shows the COBOL pictures and usages of the COBOL data items pointed at by sqldata, while the middle and the right columns show the equivalent SQL type codes and lengths:

COBOL Type Codes	SQL Type Codes (sqltype)	Length (sqllen)
PIC S9(4) USAGE COMP	30 (INTEGER)	2

COBOL Type Codes	SQL Type Codes (sqltype)	Length (sqllen)
PIC S9(9) USAGE COMP	30 (INTEGER)	4
USAGE COMP-1	31 (FLOAT)	4
IISQLHDLR	46 (Datahandler)	0
USAGE COMP-2	31 (FLOAT)	8
PIC S9(P-S)V9(S) USAGE COMP-3	10 (DECIMAL)	256*P+S
PIC X(LEN)	20 (CHARACTER)	LEN

All other types are compatible with these types, as described in the COBOL Data Items and Data Types in this chapter, which describes runtime data conversion. For example, the SQL date data type can be retrieved into a COBOL 25-byte character string, while the SQL money type can be retrieved into a COMP-2 data item.

Nullable data types (those data items that are associated with a null indicator) are specified by assigning the negative of the type code to the sqltype field. If the type is negative, a null indicator (a 2-byte integer data item) must be pointed at by the sqlind field.

Character data and the SQLDA have the exact same rules as character data in regular embedded SQL statements. For more information, see COBOL Data Items and Data Types in this chapter.

Pointing at COBOL Data Items

In order to fill an element of the sqlvar array, you must set the type information and assign a valid address to sqldata. The address must be that of a legally declared data item. If the element is nullable, the sqlind field must point at a legally declared null indicator.

As a concluding example, the following fragment sets the type information of, and points at, a 4-byte integer data item, an 8-byte nullable floating-point data item, and an sqllen-specified character sub-string. The following examples demonstrate how a program can maintain a pool of available data items, such as large arrays of the few different typed variables and a large string space. The next available spot is chosen from the pool:

Windows **UNIX**

```
* Assume SQLDA has been declared, as well as the
* following COBOL tables:
*   INT-4-TABLE, FLOAT-TABLE and INDICATOR-TABLE
* Also assume that a large character string buffer has
```

```
* been declared:

* CHAR-STRING

  MOVE 30        TO SQLTYPE(1).
  MOVE 4         TO SQLLEN(1).
  SET SQLIND(1)  TO NULL.
  SET SQLDATA(1) TO ADDRESS OF INT-4-TABLE(CUR-INT).
  ADD 1          TO CUR-INT.

  MOVE -31       TO SQLTYPE(2).
  MOVE 8         TO SQLLEN(2).
  SET SQLIND(2)  TO ADDRESS OF INDICATOR-TABLE(CUR-IND).
  SET SQLDATA(2) TO ADDRESS OF FLOAT-8-TABLE(CUR-FLT).
  ADD 1          TO CUR-IND.
  ADD 1          TO CUR-FLT.

* SQLLEN has been assigned by DESCRIBE to be the length
* of a specific result column. This length is used to
* pick off a sub-string out of a large character string
* space.

  MOVE SQLLEN(3)   TO NEED-LEN.
  MOVE 20          TO SQLTYPE(3).
  SET SQLIND(3)    TO NULL.
  SET SQLDATA(3)   TO ADDRESS OF
         CHAR-STRING(CUR-CHAR:NEED-LEN).
 ADD NEED-LEN     TO CUR-CHAR.
```

It is advisable to set sqlind to point to a null address if the data represented by the sqlvar element is not nullable, that is, the sqlvar.sqltype is positive. However, because some COBOL compilers do not accept the SET *pointer-item* TO NULL syntax, Ingres will ignore the sqlind pointer if the sqltype is positive, which allows you to leave out that particular step if necessary.

VMS

```
* Assume SQLDA has been declared, as well as the

* following COBOL tables:

*     INT-4-TABLE, FLOAT-8-TABLE and INDICATOR-TABLE

* Also assume that a large character string buffer has

* been declared:

*     CHAR-STRING

    MOVE 30         TO SQLTYPE(1).
    MOVE 4          TO SQLLEN(1).
    MOVE 0          TO SQLIND(1).
    SET SQLDATA(1)  TO REFERENCE
                     OF INT-4-TABLE(CUR-INT).
    ADD 1           TO CUR-INT.

    MOVE -31        TO SQLTYPE(2).
    MOVE 8          TO SQLLEN(2).
    SET SQLIND(2)   TO REFERENCE
                     OF INDICATOR-TABLE(CUR-IND).
    SET SQLDATA(2)  TO REFERENCE
                     OF FLOAT-8-TABLE(CUR-FLT).
    ADD 1           TO CUR-IND.
    ADD 1           TO CUR-FLT.

* SQLLEN has been assigned by DESCRIBE to be the length
```

```
* of a specific result column. This length is used to
* pick off a sub-string out of a large character
* string space.

      MOVE SQLLEN(3)  TO NEED-LEN.
      MOVE 20         TO SQLTYPE(3).
      MOVE 0          TO SQLIND(3).
      SET SQLDATA(3)  TO REFERENCE OF
      CHAR-STRING(CUR-CHAR:NEED-LEN).
      ADD NEED-LEN    TO CUR-CHAR.
```

Setting SQLNAME for Dynamic FRS

When using the SQLVAR with Dynamic FRS statements, a few extra steps are required. These extra steps relate to the differences between Dynamic FRS and Dynamic SQL and are described in the *SQL Reference Guide.*

When using the SQLDA in a forms input or output using clause, the value of sqlname must be set to a valid field or column name. If the name was set by a previous describe statement, it must be retained or reset by the program. If the name refers to a hidden table field column, the program must set sqlname directly. If your program sets sqlname directly, it must also set sqlnamel and sqlnamec. The name portion does not need to be padded with blanks.

For example, a dynamically named table field has been described, and the application always initializes any table field with a hidden 6-byte character column called rowid. The code used to retrieve a row from the table field including the hidden column and _state variable would have to construct the two named columns:

Windows

```
  ...


  01 ROWID    PIC X(6).

  01 ROWSTATE PIC S9(8) USAGE COMP-5.


  ...
  EXEC FRS DESCRIBE TABLE :FORMNAME :TABLENAME
    INTO :SQLDA END-EXEC.

  ...

  ADD 1 TO SQLD.

* Set up to retrieve rowid.
  MOVE 20            TO SQLTYPE(SQLD).
  MOVE 6             TO SQLLEN(SQLD).
  SET SQLIND(SQLD)   TO NULL.
  MOVE 5             TO SQLNAMEL(SQLD).
  MOVE "rowid"       TO SQLNAMEC(SQLD)(1:5).
  SET SQLDATA(SQLD)  TO ADDRESS OF ROWID.

  ADD 1              TO SQLD.
```

```
* Set up to retrieve _STATE.
  MOVE 30             TO SQLTYPE(SQLD).
  MOVE 4              TO SQLLEN(SQLD).
  SET SQLIND(SQLD)    TO NULL.
  MOVE 6              TO SQLNAMEL(SQLD).
  MOVE "_state"       TO SQLNAMEC(SQLD)(1:6).
  SET SQLDATA(SQLD)   TO ADDRESS OF ROWSTATE.

....

  EXEC FRS GETROW :FORMNAME :TABLENAME
    USING DESCRIPTOR :SQLDA END_EXEC.
```

UNIX

```
  ...

  01 ROWID    PIC X(6).

  01 ROWSTATE PIC S9(8) USAGE COMP-5.

  ...
  EXEC FRS DESCRIBE TABLE :FORMNAME :TABLENAME
    INTO :SQLDA END-EXEC.

  ...

  ADD 1 TO SQLD.

* Set up to retrieve rowid.
  MOVE 20             TO SQLTYPE(SQLD).
  MOVE 6              TO SQLLEN(SQLD).
  SET SQLIND(SQLD)    TO NULL.
  MOVE 5              TO SQLNAMEL(SQLD).
  MOVE "rowid"        TO SQLNAMEC(SQLD)(1:5).
  SET SQLDATA(SQLD)   TO ADDRESS OF ROWID.

  ADD 1               TO SQLD.

* Set up to retrieve _STATE.
  MOVE 30             TO SQLTYPE(SQLD).
  MOVE 4              TO SQLLEN(SQLD).
  SET SQLIND(SQLD)    TO NULL.
  MOVE 6              TO SQLNAMEL(SQLD).
  MOVE "_state"       TO SQLNAMEC(SQLD)(1:6).
  SET SQLDATA(SQLD)   TO ADDRESS OF ROWSTATE.

....

  EXEC FRS GETROW :FORMNAME :TABLENAME
    USING DESCRIPTOR :SQLDA END_EXEC.
```

VMS

```
  ...

  01 ROWID       PIC X(6).

  01 ROWSTATE    PIC S9(8) USAGE COMP.

  ...
```

```
  EXEC FRS DESCRIBE TABLE :FORMNAME :TABLENAME
    INTO :SQLDA END-EXEC.
....

  ADD 1 TO SQLD.

* Set up to retrieve rowid.
  MOVE 20        TO SQLTYPE(SQLD).
  MOVE 6         TO SQLLEN(SQLD).
  MOVE 0         TO SQLIND(SQLD).
  MOVE 5         TO SQLNAMEL(SQLD).
  MOVE "rowid"   TO SQLNAMEC(SQLD)(1:5).
  SET SQLDATA(SQLD) TO REFERENCE OF ROWID.

  ADD 1 TO SQLD.

* Set up to retrieve _STATE.
  MOVE 30        TO SQLTYPE(SQLD).
  MOVE 4         TO SQLLEN(SQLD).
  MOVE 0         TO SQLIND(SQLD).
  MOVE 6         TO SQLNAMEL(SQLD).
  MOVE "_state"  TO SQLNAMEC(SQLD)(1:6).
  SET SQLDATA(SQLD) TO REFERENCE OF ROWSTATE.

....

  EXEC FRS GETROW :FORMNAME :TABLENAME
    USING DESCRIPTOR :SQLDA END_EXEC.
```

You may also set the SQLVAR to point to a datahandler for large object columns. For more information on data handlers, see Advanced Processing in this chapter.

Advanced Processing

This section describes user-defined handlers. It includes information about user-defined error, dbevent, and message handlers as well as data handlers for large objects.

User-Defined Error, DBevent, and Message Handlers

You can use user-defined handlers to capture errors, messages, or events during the processing of a database statement. Use these handlers instead of the sql whenever statements with the SQLCA when you want to do the following:

- Capture more than one error message on a single database statement.
- Capture more than one message from database procedures fired by rules.
- Trap errors, events, and messages as the DBMS raises them. If an event is raised when an error occurs during query execution, the WHENEVER mechanism detects only the error and defers acting on the event until the next database statement is executed.

User-defined handlers offer you flexibility. If, for example, you want to trap an error, you can code a user-defined handler to issue an inquire_sql to get the error number and error text of the current error. You can then switch sessions and log the error to a table in another session; however, you must switch back to the session from which the handler was called before returning from the handler. When the user handler returns, the original statement continues executing. User code in the handler cannot issue database statements for the session from which the handler was called.

The handler must be declared to return an integer. However, the Ingres runtime system ignores the return value.

Syntax Notes:

Windows
UNIX

Because Micro Focus COBOL does not support a function pointer data type, you must write a short embedded SQL/C procedure to register your handler with the Ingres runtime system. For more information, see Including User-Defined Handlers in the Micro Focus RTS—UNIX in this chapter.

The following syntax describes the three types of handlers.

Use the following embedded SQL/C procedure to set the handlers:

```
exec sql set_sql (errorhandler   = error_routine);
exec sql set_sql (messagehandler = message_routine);
exec sql set_sql (dbeventhandler = event_routine);
```

VMS

```
exec sql set_sql (errorhandler   = error_routine) end-exec.
exec sql set_sql (messagehandler = message_routine) end-exec.
exec sql set_sql (dbeventhandler = event_routine) end-exec.
```

They may be unset directly from your embedded SQL/COBOL program:

```
exec sql set_sql (errorhandler   = 0) end-exec.
exec sql set_sql (messagehandler = 0) end-exec.
exec sql set_sql (dbeventhandler = 0) end-exec.
```

1. Errorhandler, dbeventhandler, and messagehandler denote a user-defined handler to capture errors, events, and database messages respectively, as follows:

   - error_routine is the name of the function the Ingres runtime system calls when an error occurs.
   - event_routine is the name of the function the Ingres runtime system calls when a database event is raised.
   - message_routine is the name of the function the Ingres runtime system calls whenever a database procedure generates a message.

   Errors that occur in the error handler itself do not cause the error handler to be reinvoked. You must use inquire_sql to handle or trap any errors that may occur in the handler.

2. Unlike regular variables, the handler in the embedded SQL SET_SQL statement is not prefaced by a colon. The handler must not be declared in an embedded SQL declare section although you must declare the handler to the compiler.

3. If you specify a zero (0) instead of a name, the zero will unset the user-defined handlers are also described in the *SQL Reference Guide.*

Declaring and Defining User-Defined Handlers

The following examples show how to declare a user-defined handler in ESQL/COBOL:

UNIX

```
IDENTIFICATION DIVISION.

PROGRAM-ID.  TEST-PROG.

ENVIRONMENT DIVISION.

DATA DIVISION.


WORKING_STORAGE SECTION.
     EXEC SQL INCLUDE SQLCA END-EXEC.
...
PROCEDURE DIVISION.
BEGIN.

     EXEC SQL CONNECT dbname END-EXEC.

*    Call "C" routine to set error handler on.
*    ErrProg will be called whenever an error occurs.
     CALL "ErrTrap".
     ...
     ...
*    Suppress display of error number (don't call ErrProg) for next statement by
*    turning error handler off.
     EXEC SQL SET_SQL(ERRORHANDLER = 0) END-EXEC.
     EXEC SQL .....
*    Turn error handler back on. ErrProg will now be
*    called again whenever an error occurs.
     CALL "ErrTrap".
        ...
END PROGRAM TEST-PROG.
```

The following is an example of a user-defined error handler:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  ErrProg.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
EXEC SQL INCLUDE SQLCA END-EXEC.

EXEC SQL BEGIN DECLARE SECTION END_EXEC.
01 errnum    PIC S9(9) USAGE DISPLAY.
EXEC SQL END DECLARE SECTON END-EXEC.
PROCEDURE DIVISION.
BEGIN.
```

```
        EXEC SQL INQUIRE_SQL(:errnum = ERRORNO) END-EXEC.
        DISPLAY "Errnum is " errnum.
    END PROGRAM ErrProg.
```

The following example is an embedded SQL/C routine that declares ErrProg to the Ingres runtime system:

```
ErrTrap()
{
      extern int ErrProg();
      EXEC SQL SET_SQL (ERRORHANDLER = ErrProg);
}
```

VMS

```
IDENTIFICATION DIVISION.

PROGRAM-ID.  Test-prog.

ENVIRONMENT DIVISION.

DATA DIVISION.


WORKING-STORAGE SECTION.
         EXEC SQL INCLUDE SQLCA END-EXEC.
         01 error_func PIC S9(9) USAGE COMP VALUE EXTERNAL
         ErrProg.
...

PROCEDURE DIVISION
BEGIN.
          EXEC SQL CONNECT dbname END-EXEC.
          EXEC SQL SET_SQL (ERRORHANDLER = error_func)
                END-EXEC.
          .  .  .
END PROGRAM Test-prog.
IDENTIFICATION DIVISION.
PROGRAM-ID.  ErrProg.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
        EXEC SQL INCLUDE SQLCA END-EXEC.
        EXEC SQL BEGIN DECLARE SECTION END-EXEC.
                  01 errnum PIC S9(9) USAGE DISPLAY.
                  EXEC SQL END DECLARE SECTION END-EXEC.
PROCEDURE DIVISION.
BEGIN.

      EXEC SQL INQUIRE_SQL (:errnum = ERRORNO) END-EXEC.
      DISPLAY "Errnum is " errnum.
END PROGRAM ErrProg.
```

Including User-Defined Handlers in the Micro Focus RTS—UNIX

You must follow the procedures below to include user-defined handlers in the new Micro Focus Runtime System (RTS) that you create. For a complete description of how to incorporate Ingres into the Micro Focus RTS, see Incorporating Ingres into the Micro Focus RTS—UNIX in this chapter.

1. For each user-defined handler, build the object code as follows:

```
% esqlcbl     errhandler.scb
```

```
% cob -x -c   errhandler.cbl

% esqlcbl     msghandler.scb
% cob -x -c   msghandler.cbl

% esqlcbl     evthandler.scb
% cob -x -c   evthandler.cbl
```

2. Because Micro Focus COBOL does not support a Function Pointer data type, you must write a short embedded SQL/C procedure to register your user-defined handler with the Ingres Runtime System. This embedded SQL/C procedure only needs to declare the handler, and execute the appropriate set_sql statement. For example:

```
ErrTrap()
{
    extern int ErrProg();
    exec sql set_sql(errorhandler = ErrProg);
}
MsgTrap()
{
    extern int MsgProg();
    exec sql set_sql(messagehandler = MsgProg);
}
EvtTrap()
{
    extern int EvtProg();
    exec sql set_sql(dbeventhandler = EvtProg);
}
```

   ErrProg, MsgProg and EvtProg are embedded SQL/COBOL programs that handle Ingres errors, database procedure messages and database events respectively.

3. Build the object code of the embedded SQL/C registration procedure, as follows:

```
% esqlc cproc.sc
% cc -c cproc.c
```

   Where cproc.sc is the name of the file containing the procedure(s) that you wrote for Step 2.

4. Link the compiled handlers and the C registration procedure(s) into your RTS by modifying the COB command line to include the object files. Specify the object files before the list of system libraries, as follows:

```
cob -x -e "" -o ingrts
    iimfdata.o iimflibq.o\
    cproc.o              \
    errhandler.o msghandler.o evthandler.o\
    $II_SYSTEM/ingres/lib/libingres.a\
    -lc -lm
```

   cproc.o is the name of the object file that Step 3 produces. It contains the C registration procedure(s) for the user-defined handlers.

5. Add COBOL CALL statements to your source program wherever you wish to set the handler. For example:

```
* To set the errorhandler on:
    CALL "ErrTrap".
* To set the messagehandler on:
    CALL "MsgTrap".
* To set the dbeventhandler on:
    CALL "EvtTrap".
```

You may unset the user-defined handler directly from your embedded SQL/COBOL program with the SET_SQL statement:

```
exec sql set_sql (errorhandler   = 0) end-exec.
exec sql set_sql (messagehandler = 0) end-exec.
exec sql set_sql (dbeventhandler = 0) end-exec.
```

User-Defined Data Handlers for Large Objects

Note: User-defined data handlers for large objects are not supported in MFCOBOL on UNIX.

You can use user-defined data handlers to transmit large object column values to or from the database a segment at a time. For more details on Large Objects, the datahandler clause, the get data statement and the put data statement, see the *SQL Reference Guide* and the *Forms-based Application Development Tools User Guide*.

ESQL/COBOL Usage Notes

- The datahandler, and the datahandler argument, should not be declared in an ESQL declare section. Therefore do not use a colon before the datahandler or its argument.
- You must ensure that the datahandler argument is a valid COBOL record. ESQL will not do any syntax or datatype checking of the argument.
- The datahandler must be declared to return an integer. However, the actual return value will be ignored.

Data Handlers and the SQLDA

You may specify a user-defined data handler as an SQLVAR element of the SQLDA, to transmit large objects to/from the database. The eqsqlda.h file included using the include sqlda statement declares one IISQLHDLR record which may be used to specify one data handler and its argument. It is defined:

```
* Declare IISQLHDLR

        01 IISQLHDLR EXTERNAL.
           05 SQLARG   USAGE POINTER.
           05 SQLHDLR  PIC S9(9) USAGE COMP.
```

You can also code this record directly, instead of using the include sqlda statement. You can choose any name for the structure and you can declare more than one in a single program. The program must set the values:

```
* Declare the argument to be passed to datahandler
     01 HDLR-ARG.
            05  ARG-CHAR   PIC X(100).
            05  ARG-INT    PIC S9(9) USAGE COMP.

* Declare the datahandler
     01 GET-HANDLER PIC S9(9) USAGE COMP VALUE
             EXTERNAL GET-HANDLER.

* Set the IISQLHDLR values for SQLHDLR and SQLARG
     MOVE GET-HANDLER TO SQLHDLR.
     SET SQLARG TO REFERENCE HDLR-ARG.
```

The sqltype and sqllen fields of the SQLVAR element of the SQLDA should then be set as follows:

```
MOVE 46 TO SQLTYPE(COL).
MOVE  0 TO SQLLEN(COL).
```

Sample Programs

The programs in this section are examples of how to declare and use user-defined data handlers in an ESQL/COBOL program. There are examples of a handler program, a put handler program, a get handler program and a dynamic SQL handler program.

Handler Program

This example assumes that the book table was created with the statement:

```
EXEC SQL CREATE TABLE BOOK
      (CHAPTER_NUM  INTEGER,
       CHAPTER_NAME CHAR(50),
       CHAPTER_TEXT LONG VARCHAR) END-EXEC.
```

This program inserts a row into the table book using the data handler PUT_HANDLER to transmit the value of column chapter_text from a text file to the database. Then it selects the column chapter_text from the table book using the data handler GET-HANDLER to retrieve the chapter_text column a segment at a time.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  HANDLER-PROG.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

        EXEC SQL INCLUDE SQLCA END-EXEC.
* Do not declare the data handlers nor the
* datahandler argument to the ESQL preprocessor.

        01 PUT-HANDLER PIC S9(9) USAGE COMP VALUE
                        EXTERNAL PUT-HANDLER.
        01 GET-HANDLER PIC S9(9) USAGE COMP VALUE
                        EXTERNAL GET-HANDLER.

        01 HDLR-ARG.
                05 ARG-CHAR     PIC X(100).
                05 ARG-INT      PIC S9(9) USAGE COMP.

* Argument passed through to the DATAHANDLER must be * of type POINTER.

        01 ARG-ADDR         USAGE POINTER.

* Null indicator for data handler must be declared to * ESQL.

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01 CHAPTER_NUM    S9(9) USAGE COMP.
        01 IND-VAR       S9(4) USAGE COMP.
    EXEC SQL END DECLARE SECTION END-EXEC.

PROCEDURE DIVISION.
BEGIN.

* INSERT a long varchar value chapter_text into the
* table book using the datahandler PUT_HANDLER.
* The argument passed to the datahandler is a pointer to the record HDLR-ARG.
    ...
    SET ARG-ADDR TO REFERENCE HDLR-ARG.

    EXEC SQL INSERT INTO book (chapter_num,
        chapter_name, chapter_text)
            VALUES (5, 'One dark and stormy night',
            DATAHANDLER (PUT-HANDLER (ARG-ADDR)))
    END-EXEC.
    ...

* Select the column chapter_num and the long varchar * column chapter_text from
the table book.
* The Datahandler (GET-HANDLER) will be invoked for each non-null value
* of column chapter_text retrieved. For null values the indicator variable
* will be set to "-1" and the datahandler will not be called. Again, the argument
* passed to the handler is a pointer to the record HDLR-ARG.
```

```
    ...

    EXEC SQL SELECT chapter_num, chapter_text INTO
         :CHAPTER_NUM,
            DATAHANDLER (GET-HANDLER(ARG-ADDR)):IND-VAR
            FROM book END-EXEC
    EXEC SQL BEGIN END-EXEC
            process row ...
    EXEC SQL END END-EXEC.

    ...

END PROGRAM HANDLER-PROG.
```

Put Handler

This example shows how to read the long varchar chapter_text from a text file and insert it into the database a segment at a time.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  PUT-HANDLER.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

        EXEC SQL INCLUDE SQLCA END-EXEC.

        EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01      SEG-BUF       PIC X(1000).
        01      SEG-LEN       PIC s9(9) USAGE COMP.
        01      DATA-END      PIC s9(9) USAGE COMP.
        EXEC SQL END DECLARE SECTION END-EXEC.

LINKAGE SECTION.
        01   HDLR-ARG.
            02 ARG-CHAR PIC X(100).
            02 ARG-INT  PIC S9(9) USAGE COMP.

PROCEDURE DIVISION USING ARG-ADDR.
BEGIN.

    ...
    process information passed in via the HDLR-ARG...
    open file...

    ...

    MOVE 0 TO DATA-END.

    PERFORM UNTIL DATA-END = 1
        read segment of less than 1000 chars from file into segbuf...
        IF end-of-file
            MOVE 1 TO DATA-END
        END-IF.

            EXEC SQL PUT DATA (SEGMENT = :SEG-BUF,
                SEGMENTLENGTH = :SEG-LEN, DATAEND = :DATA-END)
            END-EXEC
        END-PERFORM.
    ...
        close file ...
        set HDLR-ARG to return appropriate values...
    ...
    END PROGRAM PUT-HANDLER.
```

Get Handler

This example shows how to get the long varchar chapter_text from the database and write it to a text file.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  GET-HANDLER.

ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

        EXEC SQL INCLUDE SQLCA END-EXEC.

        EXEC SQL BEGIN DECLARE SECTION END-EXEC.
        01      SEG-BUF       PIC X(2000).
        01      SEG-LEN       PIC Z9(6) USAGE COMP.
        01      DATA-END      PIC Z9(9) USAGE COMP.
        01      MAX-LEN       PIC S9(9) USAGE COMP.
        EXEC SQL end DECLARE SECTION END-EXEC.

LINKAGE SECTION.
        01   HDLR-ARG.
            02 ARG-CHAR PIC X(100).
            02 ARG-INT  PIC S9(9) USAGE COMP.

PROCEDURE DIVISION USING HDLR-ARG.
BEGIN.

    ...
    process information passed in via the HDLR-ARG...
    open file...

*   Get a maximum segment length of 2000 bytes.

    MOVE 0 TO DATA-END.
    MOVE 2000 TO MAX-LEN.

*   seg-len:    will contain the length of the segment retrieved.
*   seg-buf:    will contain a segment of the column chapter_text.
*   data-end:   will be set to '1' when the entire value in chapter_text has been
*               retrieved.

    PERFORM UNTIL DATA-END = 1
            EXEC SQL GET DATA (:SEG-BUF = SEGMENT,
                :SEG-LEN = SEGMENTLENGTH,
                :DATA-END = DATAEND)
                WITH MAXLENGTH = :MAX-LEN
            END-EXEC.

                write segment to file...

    END-PERFORM.
    ...
    set HDLR-ARG to return appropriate values...

    END PROGRAM GET-HANDLER.
```

Dynamic SQL Handler Program

The following is an example of a dynamic SQL handler program. This program fragment shows the declaration and usage of a datahandler in a dynamic SQL program, using the SQLDA. It uses the datahandler GET-HANDLER and the HDLR-ARG structure.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  DYNHDLR-PROG.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

    EXEC SQL INCLUDE SQLCA END-EXEC.
    EXEC SQL INCLUDE SQLDA END-EXEC.

*   Do not declare the data handlers nor the
*   data handler argument to the ESQL preprocessor.

    01 PUT-HANDLER PIC S9(9) USAGE COMP VALUE
                               EXTERNAL PUT-HANDLER.
    01 GET-HANDLER PIC S9(9) USAGE COMP VALUE
                               EXTERNAL GET-HANDLER.

*   Declare argument to be passed to datahandler.

    01 HDLR-ARG.
         05 ARG-CHAR   PIC X(100).
         05 ARG-INT    PIC S9(9) USAGE COMP.

C Declare IISQLHDLR

        01 IISQLHDLR EXTERNAL.
             05 SQLARG      USAGE POINTER.
             05 SQLHDLR     PIC S9(9) USAGE COMP.

    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
            01 INDVAR       PIC s9(4) USAGE COMP.
    EXEC SQL END DECLARE SECTION END-EXEC.

PROCEDURE DIVISION.
BEGIN.

        .  .  .

* Set the IISQLHDLR structure with the appropriate datahandler and
* datahandler argument.

        MOVE GET-HANDLER TO SQLHDLR.
        SET SQLARG TO REFERENCE HDLR-ARG.

* Describe the statement into the SQLDA.

        STMT-BUF = "select * from book".
        EXEC SQL PREPARE stmt FROM :STMT-BUF END-EXEC.
        EXEC SQL DESCRIBE stmt INTO :SQLDA END-EXEC.

* Set the SQLDATA variables correctly.

        PERFORM SETUP-COLUMN VARYING COL FROM 1 BY 1
                UNTIL (COL > SQLD).

* The Datahandler (GET-HANDLER) will be invoked for
* each non-null value of column "chapter_text"
* retrieved. For null values the SQLIND will be set * to "-1" and the datahandler
* will not be called.
```

```
        EXEC SQL EXECUTE IMMEDIATE :STMT-BUF USING
    :SQLDA END-EXEC
        EXEC SQL BEGIN END-EXEC
                process row ...
        EXEC SQL END END-EXEC
        .   .   .

SETUP-COLUMN.
        .   .   .

* The Describe statement will return 22 for long
* varchar and -22 for Nullable Long Varchar

        IF (SQLTYPE(COL) = 22)
            MOVE 46 TO SQLTYPE(COL)
            SET SQLDATA(COL) TO REFERENCE IISQLHDLR
            SET SQLIND(COL) TO REFERENCE INDVAR
        ELSE
            .   .   .

        END-IF.

            .   .   .

END PROGRAM DYNHLDR-PROG.
```

Preprocessor Operation

This section describes the embedded SQL preprocessor for COBOL and the steps required to create, compile, and link an Embedded SQL program.

Include File Processing

The following sections describe include file processing for Windows, UNIX, and VMS.

Including Files—Windows and UNIX

The embedded SQL include statement provides a means to include external files in your program's source code. Its syntax is:

exec sql include *filename* **end-exec**

Filename is a single-quoted string constant specifying a file name or an environment variable that points to the file name. If no extension is given to the filename (or to the file name pointed at by the environment variable), the default COBOL input file extension .scb is assumed.

This statement is normally used to include variable declarations although it is not restricted to such use. For more details on the include statement, see the *SQL Reference Guide.*

The included file is preprocessed and an output file with the same name but with the default output extension .cbl is generated. You can override this default output extension with the -o.*ext* flag on the command line. In the original source file that specified the include statement, a new reference is made to the output file with the COBOL COPY statement. If the -o flag is used (with no extension), an output file is not generated for the include statement.

For example, assume that no overriding output extension was explicitly given on the command line. The embedded SQL statement:

```
EXEC SQL INCLUDE 'employee.scb' END-EXEC.
```

is preprocessed to the COBOL statement:

```
COPY "employee.cbl".
```

and the file employee.scb is translated into the COBOL file employee.cbl.

As another example, assume that a source file called inputfile contains the following include statement:

```
EXEC SQL INCLUDE 'mydecls' END-EXEC.
```

Windows

The name mydecls can be defined as a system environment variable pointing to the file c:\src\headers\myvars.scb by means of the following command at the system level:

```
setenv mydecls c:\src\headers\myvars.scb
```

UNIX

The name mydecls can be defined as a system environment variable pointing to the file /src/headers/myvars.scb by means of the following command at the system level:

```
setenv mydecls /src/headers/myvars.scb
```

Because the extension .scb is the default input extension for embedded SQL include files, it need not be specified when defining an environment variable for the file.

Assume now that inputfile is preprocessed with the command:

```
esqlcbl -o.hdr inputfile
```

The command line specifies .hdr as the output file extension for include files. As the file is preprocessed, the include statement shown earlier is translated into the COBOL statement:

Windows

COPY "c:\src\headers\myvars.hdr".

And the COBOL file c:\src\headers\myvars.hdr is generated as output for the original include file, c:\src\headers\myvars.scb.

You can also specify include files with a relative path. For example, if you preprocess the file c:\src\headers\myvars.scb, the embedded SQL statement:

```
EXEC SQL INCLUDE '../headers/myvars.scb' END-EXEC.
```

is preprocessed to the COBOL statement:

```
include "../headers/myvars.cbl".
```

And the COBOL file c:\src\headers\myvars.cbl is generated as output for the original include file, c:\src\headers\myvars.cbl.

UNIX

COPY "/src/headers/myvars.hdr".

And the COBOL file /src/headers/myvars.hdr is generated as output for the original include file, /src/headers/myvars.scb.

You can also specify include files with a relative path. For example, if you preprocess the file /src/source/myprog.scb, the embedded SQL statement:

```
EXEC SQL INCLUDE '../headers/myvars.scb' END-EXEC.
```

is preprocessed to the COBOL statement:

```
include "../headers/myvars.cbl".
```

And the COBOL file /src/headers/myvars.cbl is generated as output for the original include file, /src/headers/myvars.scb.

Including Files—VMS

The embedded SQL include statement provides a means to include external files in your program's source code. Its syntax is:

exec sql include *filename* **end-exec**

Filename is a single-quoted string constant specifying a file name or an environment variable that points to the file name. If no extension is given to the filename (or to the file name pointed at by the environment variable), the default COBOL input file extension .scb is assumed.

This statement is normally used to include variable declarations although it is not restricted to such use. For more details on the include statement, see the *SQL Reference Guide.*

The included file is preprocessed and an output file with the same name but with the default output extension .lib is generated. You can override this default output extension with the -o.*ext* flag on the command line. In the original source file that specified the include statement, a new reference is made to the output file with the COBOL COPY statement. If the -o flag is used with no extension, an output file is not generated for the include statement. This is useful for program libraries using MMS dependencies.

If you use both the -o.*ext* and the -o flags, then the preprocessor will generate the specified extension for the translated include statements in the program but will not generate new output files for the statements.

For example, assume that no overriding output extension was explicitly given on the command line. The embedded SQL statement:

```
EXEC SQL INCLUDE 'employee.scb' END-EXEC.
```

is preprocessed to the COBOL statement:

```
COPY "employee.lib".
```

And the file employee.scb is translated into the COBOL file employee.lib.

As another example, assume that a source file called inputfile contains the following include statement:

```
EXEC SQL INCLUDE 'mydecls' END-EXEC.
```

The name mydecls can be defined as a system logical name pointing to the file dra1:[headers]myvars.scb by means of the following command at the system level:

```
define mydecls dra1:[headers]myvars
```

Because the extension .scb is the default input extension for embedded SQL include files, it need not be specified when defining a logical name for the file.

Assume now that inputfile is preprocessed with the command:

```
esqlcbl -o.hdr inputfile
```

The command line specifies .hdr as the output file extension for include files. As the file is preprocessed, the include statement shown earlier is translated into the COBOL statement:

```
COPY "dra1:[headers]myvars.hdr"
```

And the COBOL file dra1:[headers]myvars.hdr is generated as output for the original include file, dra1:[headers]myvars.scb.

You can also specify include files with a relative path. For example, if you preprocess the file dra1:[mysource]myfile.scb, the embedded SQL statement:

```
EXEC SQL INCLUDE '[-.headers]myvars.scb' END-EXEC.
```

is preprocessed to the COBOL statement:

```
COPY "[-.headers]myvars.lib"
```

And the COBOL file dra1:[headers]myvars.lib is generated as output for the original include file, dra1:[headers]myvars.scb.

Including Source Code with Labels

Some embedded SQL statements generate labels. If you include a file containing such statements, you must be careful to include the file only once in a given COBOL program unit. Otherwise, you may find that the compiler later complains that the generated labels are multiple defined.

The embedded SQL select loop generates labels and all the embedded SQL/FORMS block-type statements, such as display and unloadtable.

Coding Requirements for Writing Embedded SQL Programs

This section describes the code generated by the preprocessor and how that code can affect your program.

Comments Embedded in COBOL Output

Each embedded SQL statement generates one comment and a few lines of COBOL code. You may find that the preprocessor translates 50 lines of embedded SQL into 200 lines of COBOL. This may result in confusion about the line numbers when you try to debug the original source code. To facilitate debugging, each group of COBOL statements associated with a particular statement is delimited by a comment corresponding to the original embedded SQL source. Each comment is one line long and describes the file name, line number and type of statement in the original source file.

Embedded SQL Statements In IF and PERFORM Blocks

The preprocessor may produce several COBOL statements for a single embedded SQL statement. In most circumstances, the statements can be simply nested in the scope of a COBOL IF or PERFORM statement.

There are some embedded SQL statements for which the preprocessor generates COBOL paragraphs and paragraph names. These statements are:

select-loop
display
formdata
unloadtable
submenu

These statements cannot be nested in the scope of a COBOL IF or PERFORM statement because of the paragraph names the preprocessor generates for them.

These statements must not contain labels.

Another consequence of these generated paragraphs is that they may terminate the scope of a local COBOL paragraph, thus modifying the intended flow of control. For example, a paragraph generated by the preprocessor in a source paragraph may cause the program to return prematurely to the statement following the PERFORM statement that called the source paragraph. To ensure that control does not return prematurely, you must use the THROUGH clause in the PERFORM statement.

The following example demonstrates the use of PERFORM-THROUGH and an EXIT paragraph to force correct control flow:

```
      DATA DIVISION.
      WORKING-STORAGE SECTION.
      EXEC SQL BEGIN DECLARE SECTION END-EXEC.
       01 ENAME PIC X(20).
      EXEC SQL END DECLARE SECTION END-EXEC.
* Include SQLCA, declare program variables, etc.
      PROCEDURE DIVISION.
      BEGIN.
* Initialization of program
* Note the THROUGH clause to ensure correct
* control flow.
      PERFORM UNLOAD-TAB THROUGH END-UNLOAD.
* User code
      UNLOAD-TAB.
* This paragraph includes a paragraph generated
* by the preprocessor
      EXEC FRS UNLOADTABLE Empform Employee
         (:ENAME = Lastname) END-EXEC.
      EXEC FRS BEGIN END-EXEC.
            EXEC SQL INSERT into person (name)
                 VALUES (:ENAME)
                 END-EXEC.
      EXEC FRS END END-EXEC.
* This paragraph-name and EXIT statement cause
```

```
* control to pass back to the caller's scope
      END-UNLOAD.
            EXIT.
      USER-PARAGRAPH.
* Program continues
```

COBOL Periods and Embedded SQL Statements

You can place a period following the END-EXEC statement terminator (for more information, see Embedded SQL Statement Syntax for COBOL in this chapter), although the preprocessor does not require this. If you do include a period at the end of an embedded SQL statement, the preprocessor places a period at the end of the last COBOL statement generated by that embedded SQL statement. Therefore, when you include periods in embedded SQL statements, be careful to follow the same guidelines that you use for placing periods in COBOL statements. For example, do not add a period at the end of an embedded SQL statement occurring in the middle of the scope of a COBOL IF or PERFORM statement. If you include the separator period in such a case, you will prematurely end the scope of the COBOL statement. Similarly, when an embedded SQL statement is the *last* statement in the scope of a COBOL IF, you *must* follow it with a period (or, alternatively, an END-IF) to terminate the scope of the IF. For example:

```
    IF ERR-NO > 0 THEN
* Do not use a separating period in the middle
* of an IF statement.
        EXEC FRS MESSAGE 'You cannot update the database'
END-EXEC
* Be sure to use a separating period at the
* end of an IF statement.
        EXEC FRS SLEEP 2 END-EXEC.
```

In the example above, the absence of the period after the first end-exec causes the preprocessor to generate code *without* the separator period, thus preserving the scope of the IF statement. The period following the second end-exec causes the preprocessor to generate code *with* a final separator period, terminating the scope of the IF.

The embedded SQL preprocessor always generates necessary separator periods when translating embedded SQL block structured statements, such as a select or unloadtable loop, into COBOL paragraphs. The end-exec statement terminator associated with these statements and with their begin clauses cannot be followed by a period. A period will cause a preprocessor syntax error on the subsequent components of the block structured statement.

In a display loop, periods are allowed in the statement blocks of initialize and activate statements and following the finalize statement.

The following example shows the use of the period in block-structured statements and display loops.

```
EXEC FRS FORMS END-EXEC.                -- Period allowed

EXEC FRS DISPLAY empform END-EXEC       -- No period
```

```
EXEC FRS INITIALIZE END-EXEC            -- No period
EXEC FRS BEGIN END-EXEC                 -- No period
    ESQL, COBOL statements              -- Periods allowed

    EXEC SQL SELECT * INTO :emp_rec
           FROM employee END-EXEC       -- No period
    EXEC SQL BEGIN END-EXEC             -- No period

    ESQL, COBOL statements.             -- Periods allowed

    EXEC SQL END END-EXEC.              -- Period allowed

EXEC FRS END END-EXEC                   -- No period

EXEC FRS ACTIVATE FIELD emp_name END-EXE -- No period
EXEC FRS BEGIN END-EXEC                 -- No period

    EXEC FRS SUBMENU END-EXEC           -- No period

    EXEC FRS ACTIVATE frskey3 END-EXEC  -- No period
    EXEC FRS BEGIN END-EXEC             -- No period

        ESQL, COBOL statements.         -- Periods allowed

        IF condition THEN
            ESQL, COBOL statements      -- No periods
        ELSE
            ESQL, COBOL statements      -- No periods
        END-IF.                         -- Period optional
                                           (COBOL rules)

        PERFORM UNTIL condition
            ESQL, COBOL statements      -- No periods
        END-PERFORM.                    -- Period optional
                                           (COBOL rules)

     EXEC FRS END END-EXEC              -- No period

EXEC FRS END END-EXEC                   -- No period

EXEC FRS ACTIVATE MENUITEM 'Save' END-EXE-- No period
EXEC FRS BEGIN END-EXEC                 -- No period

  EXEC FRS UNLOADTABLE empform employee END-EXEC -- No period
  EXEC FRS BEGIN END-EXEC               -- No period

        ESQL, COBOL statements.         -- Periods allowed

    EXEC FRS END END-EXEC.              -- Period allowed

EXEC FRS END END-EXEC                   -- No period

EXEC FRS FINALIZE END-EXEC.             -- Period allowed
```

A period after the END-EXEC terminator of any of the following statements will cause a preprocessor error:

display
initialize
activate
submenu
formdata
unloadtable
end, except when used as the final statement of **display**, **unloadtable**, **formdata**, and **select** loops
select, when opening a select loop

For more information on COBOL paragraphs and embedded SQL structured statements, see the preceding section.

Embedded SQL Statements That Do Not Generate Code

The following embedded SQL declarative statements do not generate any COBOL code:

declare cursor
declare statement
declare table
whenever

Do not code these statements as the only statements in COBOL constructs that do not allow *null* statements. Also, these statements must not contain labels. For example, coding a declare cursor statement as the only statement in a COBOL IF statement causes compiler errors:

```
IF USING-DATABASE=1 THEN
    EXEC SQL DECLARE empcsr CURSOR FOR
         SELECT ename FROM employee END-EXEC
ELSE
    DISPLAY "You have not accessed the database".
The code generated by the preprocessor is:

IF USING-DATABASE=1 THEN
ELSE
    DISPLAY "You have not accessed the database".
```

This is an illegal use of the COBOL ELSE clause.

Also, do not precede these statements (declare cursor, declare statement, declare table, and whenever) with a COBOL paragraph label (on the same line) if that label is referenced elsewhere in your program.

Efficient Code Generation

This section describes the COBOL code generated by the embedded SQL preprocessor.

COBOL Strings and Embedded SQL Strings

COBOL stores string and character data in a machine-dependent data item (UNIX and Windows) or descriptor (VMS). The embedded SQL runtime routines are written in another language (C) that verifies lengths of strings by the location of a null (LOW-VALUE) byte. Consequently, COBOL strings must be converted to embedded SQL runtime strings before the call to the runtime routine is made.

In some languages, embedded SQL generates a nested function call that accepts as its argument the character data item (UNIX and Windows) or VAX string descriptor (VMS) and returns the address of the embedded SQL null-terminated string. COBOL does not have nested function calls, and simulating this would require two expensive COBOL statements. Embedded SQL/COBOL knows the context of the statement, and in most cases will MOVE the COBOL string constant or data item in a known area that has already been null-terminated. This extra statement is cheaper than the nested function call of other languages, as it generates a single machine instruction. Even though your COBOL-generated code may look wordier and longer than other embedded SQL-generated code, it is actually as efficient.

COBOL IF-THEN-ELSE Blocks

There are some statements that normally generate an IF-THEN-ELSE construct in other languages that instead generate IF-GOTO constructs in COBOL. The reason for this is that there is no way to ensure that no embedded SQL-generated (or programmer-generated) period will appear in an IF block. Consequently, in order to allow any statement in this scope, embedded SQL generates an IF-GOTO construct.

The code generated by embedded SQL for this construct is actually very similar to the code generated by any compiler for an IF-THEN-ELSE construct and is no less efficient.

COBOL Function Calls

COBOL supports function calls with the USING clause (UNIX) or the GIVING clause (VMS). This allows a function to return a value into a declared data item. Embedded SQL generates many of these statements by assigning the return values into internally declared data items, and then checking the result of the function by checking the value of the data item. This is obviously less efficient than other languages that check the return value of a function by means of its implicit value (stored in a register).

COBOL has the overhead of assigning the value to a variable. An embedded SQL/COBOL generated function call that tests the result may look like:

Windows **UNIX**

```
CALL "IIFUNC" USING IIRESULT
IF (IIRESULT = 0) THEN ...
```

VMS

```
CALL "IIFUNC" GIVING IIRESULT
IF (IIRESULT = 0) THEN ...
```

Command Line Operations

This section describes the operations you must perform from the operating system command line in order to create an executable image of an embedded SQL program. These operations include preprocessing the embedded program and compiling the generated code.

The Embedded SQL Preprocessor Command

The following command line invokes the COBOL preprocessor:

esqlcbl {*flags*} {*filename*}

where *flags* are:

VMS	-a	Accepts input and generates output in ANSI format. Use this flag if your source code is in ANSI format and you want to compile the program with the cobol command line qualifier ansi_format. The code the preprocessor generates will also be in ANSI format. If this flag is omitted, the preprocessor accepts input and generates output in VAX COBOL terminal format. For more information, see Source Code Format in this chapter.
	-d	Adds debugging information to the runtime database error messages embedded SQL generates. The source file name, line number and statement in error are printed with the error message.
	-f[*filename*]	Writes preprocessor output to the named file. If no filename is specified, the output is sent to standard output, one screen at a time.
	-l	Writes preprocessor error messages to the preprocessor's listing file, as well as to the terminal. The listing file includes preprocessor error messages and your source text in a file named filename.lis, where filename is the name of the input file.
	-lo	Like -l, but the generated COBOL code also appears in the listing file.
	-o.*ext*	Specifies the extension the preprocessor gives to both the translated include statements in the main program and the generated output files. If this flag is not provided the default is .cbl (UNIX) or .lib (VMS). If you use this flag with the -o flag, then the preprocessor generates the specified extension for the translated include statements but does not generate new output files for the include statements.

VMS	-a	Accepts input and generates output in ANSI format. Use this flag if your source code is in ANSI format and you want to compile the program with the cobol command line qualifier ansi_format. The code the preprocessor generates will also be in ANSI format. If this flag is omitted, the preprocessor accepts input and generates output in VAX COBOL terminal format. For more information, see Source Code Format in this chapter.
	-o	Directs the preprocessor not to generate output files for include files. This flag does not affect the translated include statements in the main program. The preprocessor will generate a default extension for the translated include file statements unless you use the -o.*ext* flag.
	-s	Reads input from standard input and generates COBOL code to standard output. This is useful for testing statements you are not familiar with. If the -l option is specified with this flag, the listing file is called "stdin.lis." To terminate the interactive session, type CtrlD (UNIX) or Ctrl Z (VMS).
	-sqlcode	Indicates the file declares an integer variable named SQLCODE to receive status information from SQL statements. That declaration need not be in an exec sql begin/end declare section. This feature is provided for ISO Entry SQL92 conformity. However the ISO Entry SQL92 specification describes SQLCODE as a "deprecated feature" and recommends using the SQLSTATE variable.
	-nosqlcode	Tells the preprocessor not to assume the existence of a status variable named SQLCODE. The -nosqlcode flag is the default.
	-w	Prints warning messages.
	-wopen	This flag is identical to -wsql= open. However,-wopen is supported only for backwards capability. See -wsql = open for more information.
	-wsql= entry_SQL92	Causes the preprocessor to flag any usage of syntax or features that do not conform to the ISO Entry SQL92 entry level standard. (This is also known as the FIPS flagger option.)
	-wsql=open	Use open only with OpenSQL syntax. -wsql = open generates a warning if the preprocessor encounters an embedded SQL statement that does not conform to OpenSQL syntax. (For OpenSQL syntax, see the *OpenSQL Reference Guide*.) This flag is useful if you intend to port an application across different Enterprise Access products. The warnings do not affect the generated code and the output file may be compiled. This flag does not validate the statement syntax for any Enterprise Access product whose syntax is more restrictive than that of OpenSQL.

VMS	-a	Accepts input and generates output in ANSI format. Use this flag if your source code is in ANSI format and you want to compile the program with the cobol command line qualifier ansi_format. The code the preprocessor generates will also be in ANSI format. If this flag is omitted, the preprocessor accepts input and generates output in VAX COBOL terminal format. For more information, see Source Code Format in this chapter.
Windows	-?	Shows the command line options for esqlcbl.
UNIX	--	Shows the command line options for esqlcbl.
VMS	-?	Shows the command line options for esqlcbl.

The embedded SQL/COBOL preprocessor assumes that input files are named with the extension .scb.

To override this default, specify the file extension of the input file(s) on the command line. The output of the preprocessor is a file of generated COBOL statements with the same name and the extension .cbl (UNIX and Windows) or .cob (VMS).

If you enter only the command, without specifying any flags or a filename, a list of flags available for the command is displayed.

The following table presents examples of the range of the options available with esqlcbl:

Command	Comment
esqlcbl file1	Preprocesses “file1.scb” to: file1.cbl (Windows and UNIX) file1.cob (VMS)
esqlcbl file2.xcb	Preprocesses “file2.xcb” to file2.cbl (Windows and UNIX) file2.cob (VMS)
esqlcbl -l file3	Preprocesses file3.scb to file3.cbl (Windows and UNIX) file3.cob (VMS) and creates listing file3.lis
esqlcbl -s	Accepts input from standard input

Command	Comment
esqlcbl -ffile4.out file4	Preprocesses file4.scb to file4.out
esqlcbl	Displays a list of available flags

Source Code Format

The following sections discuss source code formatting considerations for Windows, UNIX, and VMS.

Format Considerations—Windows and UNIX

The preprocessor produces Micro Focus COBOL source code in ANSI format.

You must place all string continuation indicators (-) in column 7. Comment indicators (*) may be in column 1 or column 7. For details on comments and continued string literals, see Embedded SQL Statement Syntax for COBOL in this chapter.

The preprocessor generates code using certain conventions. Indicators for comments and continued string literals are placed in column 7. The 01 level number for data declarations known to the preprocessor and any optional labels before embedded SQL statements are output in Area A, starting at column 8. All other embedded SQL statements are placed in Area B, starting at column 12. No statements generated extend beyond column 72. COBOL statements and declarations unknown to the preprocessor appear in the preprocessor output file unchanged from the input file.

The preprocessor does not generate any code in columns 1-6 (the Sequence Area). Do not, however, precede embedded SQL statements with sequence numbers—only the white space of a label can precede the exec keyword. Also, although the preprocessor never generates code beyond column 72 no matter which format is used, it does accept code in columns 73 - 80. Therefore, anything placed in that area on an embedded SQL line must be valid embedded SQL code.

Format Considerations—VMS

The preprocessor can produce source code written in either VAX COBOL terminal format or ANSI format. The default is terminal format; if you require ANSI format, use the -a flag on the preprocessor command line. The COBOL code that the preprocessor generates for embedded SQL statements will follow the format convention you have chosen.

In order to specify the -a flag, you must place all comment and string continuation indicators (* and -) in column 7. If you do not intend to use the -a flag, those indicators must instead be located in column 1. For details on comments and continued string literals, see Embedded SQL Statement Syntax for COBOL in this chapter.

When the -a flag is specified, the preprocessor generates code using certain conventions. Indicators for comments and continued string literals are placed in column 7. The 01 level number for data declarations known to the preprocessor and any optional labels before embedded SQL statements are output in Area A, starting at column 8. All other embedded SQL statements are placed in Area B, starting at column 12. No statements generated extend beyond column 72. COBOL statements and declarations unknown to the preprocessor appear in the preprocessor output file unchanged from the input file.

The preprocessor may generate sequence numbers in columns 1 - 6 (the Sequence Area). For information on sequence numbers, see COBOL Sequence Numbers in this chapter. Also, although the preprocessor never generates code beyond column 72 no matter which format is used, it does accept code in columns 73 - 80. Therefore, anything placed in that area on an embedded SQL line must be valid embedded SQL code.

The COBOL Compiler—Windows and UNIX

To compile this code use the cob command. The following example preprocesses and compiles the file test1.

```
esqlcbl test1.scb
cob test1.cob
```

When you use the cob command to compile the generated COBOL code, the compiler issues the following informational message:

```
    01 SQLABC PIC S9(9) USAGE COMP-5 SYNC VALUE 0
**209-I***********************************
**  COMP-5 is machine specific format.
```

As mentioned in the COBOL Data Items and Data Types section in this chapter, COMP-5 is an Ingres-compatible numeric data type and a data item of the type is included in the Ingres system COPY file. You can ignore this warning or suppress it by using the cob compiler directive or command line flag:

```
cob -C warning=1
```

Also, because the program will be run through the COBOL interpreter that is linked to the Ingres runtime system, do not modify the default values of the COBOL compiler align and ibmcomp directives. To run your embedded SQL/COBOL test program, use the ingrts command (an alias to your Ingres-linked RTS):

```
ingrts test1
```

For more information on building and linking the Interpreter (or RTS), see Incorporating Ingres into the Micro Focus RTS—UNIX in this chapter.

Note: For any operating system specific information on compiling and linking ESQL/COBOL programs, see the Readme file.

The COBOL Compiler—Windows Micro Focus Net Express

On Windows, to compile the COBOL code generated by the preprocessor, use the cobol command. Then use the cblnames command to extract all public symbols into a cbllds.obj file for the linker, and the link utility to bind the objects into a executable.

The following example preprocesses and compiles the file test1:

```
esqlcbl test1.scb
cobol   test1.cbl  /case  /litlink
cblnames -t  -mtest1  test1.obj
link      /OUT: test1.exe \
          /SUBSYSTEM:CONSOLE \
          /MACHINE:i386 \
          /NOD \
          test1.obj \
          cbllds.obj \
          libingres.lib \
          msvcrt.lib \
          oldnames.lib \
          mfrts32s.lib \
          kernel32.lib \
          user32.lib \
          gdi32.lib \
          advapi32.lib
```

The COBOL Compiler—VMS

The preprocessor generates COBOL code. To compile this code use the VMS COBOL command. The following example preprocesses and compiles the file test1. Both the embedded SQL preprocessor and the COBOL compiler assume the default extensions.

```
esqlcbl test1
cobol/list test1
```

As of Ingres II 2.0/0011 (axm.vms/00) Ingres uses member alignment and IEEE floating-point formats. Embedded programs must be compiled with member alignment turned on. In addition, embedded programs accessing floating-point data (including the MONEY data type) must be compiled to recognize IEEE floating-point formats.

Linking an Embedded SQL Program

Embedded SQL programs require procedures from several VMS shared libraries in order to run properly. Once you have preprocessed and compiled an embedded SQL program, you can link it. Assuming the object file for your program is called dbentry, use the following link command:

```
link dbentry.obj,-
ii_system:[ingres.files]esql.opt/opt
```

Assembling and Linking Precompiled Forms

The technique of declaring a precompiled form to the FRS is discussed in the *SQL Reference Guide* and in the COBOL Data Items and Data Types section in this chapter. To use such a form in your program, you must also follow the steps described here.

In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED lets you select the name for the file. Once you have created the MACRO file this way, you can assemble it into linkable object code with the VMS command.

macro *filename*

The output of this command is a file with the extension .obj. You then link this object file with your program by listing it in the link command, as in the following example, which links the form defined in the file empform:

```
link formentry,-
 empform.obj,-
 ii_system:[ingres.files]esql.opt/opt
```

Linking an Embedded SQL Program Without Shared Libraries

While the use of shared libraries in linking embedded SQL programs is recommended for optimal performance and ease of maintenance, non-shared versions of the libraries have been included in case you require them. Non-shared libraries required by embedded SQL are listed in the esql.noshare options file. The options file must be included in your link command *after* all user modules. The libraries must be specified in the order given in the options file.

The following example demonstrates the link command for an embedded SQL program called dbentry, which has been preprocessed and compiled:

```
link dbentry,-
ii_system:[ingres.files]esql.noshare/opt
```

Placing User-Written Embedded SQL Routines in Shareable Images

When you plan to place your code in a shareable image, note the following about the psect attributes of your global or external variables:

- As a default, some compilers mark global variables as shared (SHR: every user who runs a program linked to the shareable image sees the same variable) and others mark them as not shared (NOSHR: every user who runs a program linked to the shareable image gets their own private copy of the variable).
- Some compilers support modifiers you can place in your source code variable declaration statements to explicitly state which attributes to assign a variable.
- The attributes that a compiler assigns to a variable can be overridden at link time with the psect_attr link option, which overrides attributes of all variables in the psect.

Consult your compiler reference manual for further details

Note: For any operating system specific information on compiling and linking ESQL/COBOL programs, see the Readme file.

Incorporating Ingres into the Micro Focus RTS—UNIX

Before you can run any embedded SQL/COBOL program, you must create a new Micro Focus Runtime System (or RTS), linked with the Ingres libraries. This will enable your embedded SQL/COBOL programs to access the necessary Ingres routines at runtime.

If you are unsure whether your COBOL RTS is linked to the Ingres libraries, you can perform a simple test. Preprocess, compile, and run a simple ESQL/COBOL program that connects and disconnects from Ingres. For example, the simple test file test.scb could include the lines:

```
EXEC SQL CONNECT dbname END-EXEC.
```

```
EXEC SQL DISCONNECT END-EXEC.
```

If your COBOL RTS is not linked to the Ingres libraries, you will receive the COBOL runtime error number 173 when you run the program:

```
esqlcbl test.scb
cob test.cbl
cobrun test
    Load error: file 'IIsqConnect'
    error code: 173, pc=1A, call=1, seg=0
    173 Called program file not found in
         drive/directory
```

Building an Ingres RTS Without the Ingres FRS

If you are using the COBOL screen utilities and do not need to incorporate the Ingres forms runtime system (FRS) into your COBOL runtime support module, then you can link the RTS exclusively for database activity.

This section describes how to provide the COBOL RTS with all Ingres runtime routines.

Create a directory in which you want to store the Ingres-linked RTS. For example, if the COBOL root directory is /usr/lib/cobol, you may want to add a new directory /usr/lib/cobol/ingres to store the Ingres/COBOL RTS. From that new directory, issue the commands that extract the Ingres Micro Focus support modules, link the Ingres COBOL RTS, and supply an alias to run the new program.

The shell script shown below performs all of these steps. Note that $II_SYSTEM refers to the path-name of the Ingres root directory on your system:

```
#
# These 2 steps position you to where you want to
# build the RTS
#
mkdir /usr/lib/cobol/ingres
cd /usr/lib/cobol/ingres
#
# Extract 2 Ingres Micro Focus COBOL support modules
#
ar xv $II_SYSTEM/ingres/lib/libingres.a iimfdata.o
ar xv $II_SYSTEM/ingres/lib/libingres.a iimflibq.o
#
# Now link the new Ingres COBOL RTS (this example
# calls it "ingrts")
#
cob -x -e "" -o ingrts \
  iimfdata.o iimflibq.o \
  $II_SYSTEM/ingres/lib/libingres.a \
  -lc -lm
#
# Provide an alias to run the new program
* (distribute to RTS users)
#
alias ingrts /usr/lib/cobol/ingrts
```

Ingres shared libraries are available on some UNIX platforms. To link with these shared libraries replace libingres.a in the cob command with:

```
-L $II_SYSTEM/ingres/lib -linterp.1 -lframe.1 -lq.1 \
     -lcompat.1
```

To verify if your release supports shared libraries check for the existence of any of these four shared libraries in the $II_SYSTEM/ingres/lib directory. For example:

```
ls -l $II_SYSTEM/ingres/lib/libq.1.*
```

Any user-defined handlers must also be incorporated into the Ingres/COBOL RTS, and should be added to the cob command line. For a detailed description, see Including User-Defined Handlers in the Micro Focus RTS—UNIX in this chapter.

Since the resulting RTS is quite large, the temporary holding directory required by COBOL may need to be reset. By default, this directory is set to /usr/tmp. If you are issued “out of disk space” errors during the linking of the Ingres/COBOL RTS, you should consult your COBOL Programmer’s Reference Manual to see how to modify the TMPDIR environment variable.

You may need to specify other system libraries in addition to the -lm library on the cob command. The libraries required are the same as those need to link an embedded SQL/C program. The library names may be added to the last line of the cob command shown above. For example, if the inet and the inetd system libraries are required, the last line of the cob command would be:

```
-lc -lm -linet -linetd
```

At this point you are ready to run your embedded SQL/COBOL program.

Building an RTS with the Ingres FRS

If you are using the Ingres forms system in your embedded SQL/COBOL programs then you must include the Ingres FRS in the RTS. The link script shown below builds an RTS that includes the Ingres FRS:

```
#
# Optional: Assume you are in an appropriate directory
# as described in the previous section.
#
cd /usr/lib/cobol/ingres
#
# Extract 3 Ingres Micro Focus support modules
#
ar xv $II_SYSTEM/ingres/lib/libingres.a iimfdata.o
ar xv $II_SYSTEM/ingres/lib/libingres.a iimflibq.o
ar xv $II_SYSTEM/ingres/lib/libingres.a iimffrs.o
#
# Now link the new Ingres COBOL RTS (this example
# calls it "ingfrs")
#
cob -x -e "" -o ingfrs \
  iimfdata.o iimflibq.o iimffrs.o \
```

```
  $II_SYSTEM/ingres/lib/libingres.a \
  -lc -lm
#
# Provide an alias to run the new program
# (distribute to RTS users)
#
alias ingfrs /usr/lib/cobol/ingfrs
```

You may be required to specify other system libraries on the cob command line. For information about how to specify other system libraries on the cob command line, see Building an Ingres RTS Without the Ingres FRS in this chapter.

Including External Compiled Forms in the RTS

The description of how to build an Ingres RTS that can access the Ingres forms system does not include a method with which to include compiled forms into the RTS. Recall that compiled forms are precompiled form objects that do not need to be retrieved from the database. Since the compiled forms are externals objects (in object code) you must link them into your RTS.

Because some UNIX platforms allow you to use the Micro Focus EXTERNAL clause to reference objects linked into your RTS and some do not, two procedures are given here. The first procedure describes how to include external compiled forms in the RTS on a platform that does permit the use of the EXTERNAL clause. The second procedure describes how to perform this task on a platform that does not allow EXTERNAL data items to reference objects linked to the RTS.

Procedure 1

Use this procedure if your platform accepts the EXTERNAL clause to reference objects linked into your RTS.

1. Build and compile the form in VIFRED.

   When you compile a form in VIFRED, you are prompted for the name of the file, and VIFRED then creates the specified file in your directory, describing the form in C.

2. Compile the C file into object code:

   ```
   % cc -c formfile.c
   ```

3. Link the compiled form(s) into your RTS by modifying the cob command line to include the object files for the forms. List the files before listing the system libraries that will be linked.

   For example:

   ```
   cob -x -e "" -o ingfrs \
       iimfdata.o iimflibq.o iimffrs.o \
       form1.o form2.o \
       ...
   ```

Procedure 2

Use this procedure if your platform does not allow you to use the Micro Focus EXTERNAL clause to reference objects linked into your RTS. The extra steps force the external object to be loaded into your RTS and allow access to it through your ESQL/COBOL program.

1. Build and compile the form in VIFRED.

   When you compile a form in VIFRED, you are prompted for the name of the file, and VIFRED then creates the specified file in your directory, describing the form in C.

2. Compile the C file into object code:

   ```
   % cc -c formfile.c
   ```

3. Write a small embedded SQL/C procedure that just references the form and initializes it to the Ingres FRS using the addform statement.

   Make sure that the name of the procedure follows conventions allowed for externally called names. For example, external names may be restricted to 14 characters on some versions of COBOL.

   For example:

   ```
   EXEC SQL BEGIN DECLARE SECTION;
         extern int *form1;
         extern int *form2;
   EXEC SQL END DECLARE SECTION;
   add_form1()
   {
       EXEC FRS ADDFORM :form1;
   }
   add_form2()
   {
       EXEC FRS ADDFORM :form2;
   }
   ```

4. Build the object code for the initialization of the compiled forms:

   ```
   % esqlc filename.sc
   % cc -c filename.c
   ```

   where *filename.sc* is the name of the file containing the procedure written in Step 3.

5. Link the compiled form(s) and the initialization references to the form(s) into your RTS by modifying the cob command line to include the object files for the forms and the procedure. Specify the object files before the list of system libraries.

   For example:

   ```
   cob -x -e "" -o ingfrs \
     iimfdata.o iimflibq.o iimffrs.o \
     filename.o form1.o form2.o \
     ...
   ```

   where *filename.o* is the name of the object file resulting from Step 4, containing the initialization references to the forms form1 and form2.

6. Replace the addform statement in your source program with a COBOL CALL statement to the appropriate C initialization procedure. For example, what would have been:

   ```
   EXEC FRS ADDFORM :form1 END-EXEC.
   ```

   becomes:

   ```
   CALL "add_form1".
   ```

7. To illustrate this procedure, assume you have compiled two forms in VIFRED, empform and deptform, and need to access them from your embedded SQL/COBOL program without incurring the overhead (or database locks) of the forminit statement. After compiling them into C from VIFRED, turn them into object code:

   ```
   % cc -c empform.c deptform.c
   ```

8. Now create an embedded SQL/C file, for example, addforms.sc, that includes a procedure (or two) that initializes each one using the addform statement:

   ```
   EXEC SQL BEGIN DECLARE SECTION;
       extern int *empform;
       extern int *deptform;

   EXEC SQL END DECLARE SECTION;

   add_empform()
   {
      EXEC FRS ADDFORM :empform;

   add_deptform()
   {
      EXEC FRS ADDFORM :deptform;
      }
   ```

9. Now build the object code for the initialization of these 2 compiled forms:

   ```
   esqlc addforms.sc
   cc -c addforms.c
   ```

10. Then link the compiled forms and the initialization references to those forms into your RTS:

    ```
    cob -x -e "" -o ingfrs \
    iimfdata.o iimflibq.o iimffrs.o \
    addforms.o empform.o deptform.o \
    ...
    ```

11. Finally, be sure to replace the appropriate addform statements in your source code with COBOL CALL statements.

You can store all your compiled forms in an archive library so that the constant modification of a link script will not be required. The sample programs near the end of this section were built using such a method that included a single file, addforms.sc, and an archive library, compforms.a, that included all the compiled forms referenced in the sample programs.

If, at a later time you are able to directly reference EXTERNAL data items from your COBOL source code then the intermediate step of creating an embedded SQL/C ADDFORM procedure can be skipped, and your compiled forms declared as EXTERNAL PIC S9(9) COMP-5 data-items in your embedded SQL/COBOL source code:

```
01 empform IS EXTERNAL PIC S9(9) USAGE COMP-5.
...
EXEC FRS ADDFORM :empform END-EXEC.
```

The external object code for each form must still be linked into the RTS but there is no need to write an embedded SQL/C intermediate file, or call an external C procedure to initialize the compiled form for you.

Embedded SQL/COBOL Preprocessor Errors

To list most errors, you can run the embedded SQL preprocessor with the listing (-l) option on. The listing will be sufficient for locating the source and reason for the error.

For preprocessor error messages specific to COBOL, see Preprocessor Error Messages in this chapter.

Preprocessor Error Messages

The following is a list of error messages specific to the COBOL language:

E_DC000A

"Table '%0c' contains column(s) of unlimited length."

Explanation: Character strings(s) of zero length have been generated. This causes a compile-time error. You must modify the output file to specify an appropriate length.

E_E40001

"Ambiguous qualification of COBOL data item '%0c'."

Explanation: This data item is not sufficiently qualified to distinguish it from another data item. It is likely that the data item is an elementary member of a COBOL record or group. To avoid reference ambiguity, qualify the data item further by using IN or OF. When using COBOL table subscripts (by means of parenthesis), the subscripted item must be unambiguous when the left parenthesis is processed.

The preprocessor will generate code using the most recently declared instance of the ambiguous data item.

E_E40002 “Unsupported COBOL numeric PICTURE string ’%0c’.”

Explanation: An invalid picture character was encountered while processing a numeric picture string. A numeric picture string can include the following characters:

S
9
(
)
V

The preprocessor will treat the data item as though it was declared:

PICTURE S9(8) USAGE COMP.

E_E40003 “COMP picture ’%0c’ requires too many storage bytes. Try USAGE COMP-3.”

Explanation: The COMPUTATIONAL data type must fit into a maximum of 4 bytes. Numeric integers of more than 9 digits require VAX quad-word integer storage (8 bytes), which is incompatible with the Ingres internal runtime data types. Try reducing the picture string or declaring the data item as COMP-3 or COMP-2, which is compatible with Ingres floating-point data.

An exception is made to allow non-scaled 10-digit numeric picture strings (PICTURE S(10) USAGE COMP), which is representable by a 4-byte integer.

E_E40005 “’%0c’ is not an elementary data item. Records cannot be used.”

Explanation: In this usage, COBOL records or tables cannot be used. In order to use this data item you must refer to an elementary data item that is a member of the record, or an element of the COBOL table.

E_E40006 “COBOL declaration level %0c is out of bounds.”

Explanation: Only levels 01 through 49 and 77 are accepted for COBOL data item declarations. Level numbers outside of this range will be treated as though they were level 01. Syntax errors caused in leading clauses of a COBOL declaration may cascade and generate this error message for the OCCURS and VALUE clauses of the erroneous declaration.

E_E40007 “Data item requires a PICTURE string in this USAGE.”

Explanation: The specified USAGE clause requires a COBOL PICTURE string in order to determine preprocessor data item type information. Not all USAGE clauses require a PICTURE string. Data items with USAGE COMP, COMP-3 and DISPLAY do require a PICTURE string. If no PICTURE string is specified the preprocessor will treat the data item as though it was declared:

PICTURE X(10) USAGE DISPLAY.

E_E40008 "Data item on level %0c has no parent of lesser level."

Explanation: A data item declared on a level that is greater than the level of the most recently declared data item is considered to be a subordinate member of that group. The previous level, therefore, must be the level number of a COBOL record or group declaration. This is typical of a COBOL record containing a few elementary data items. A data item declared on a level that is less than the level of the most recently declared data item is considered to be on the same level as the "parent" of that data item. Level numbers violating this rule will be treated as though they were level 01.

E_E40009 "Keyword PICTURE and the describing string must be on the same line."

Explanation: When the preprocessor scans the COBOL PICTURE string, it must find the PICTURE keyword and the corresponding string description on the same line in the source file. The PICTURE word and the string may be separated by the IS keyword. The preprocessor will treat the declaration as though there was no PICTURE clause.

E_E4000A "'%0c' is not a legally declared data item."

Explanation: The specified data item must has not been declared but has been used in place of a COBOL variable in an embedded statement.

E_E4000B "Unsupported PICTURE '%0c' is numeric-display. USAGE COMP assumed."

Explanation: Some versions of the COBOL preprocessor do not support numeric display data items. For example:

PICTURE S9(8) USAGE DISPLAY.

If this is the case, you should use COMPUTATIONAL data items and assign to and from DISPLAY items before using the data item in embedded statements.

E_E4000C "COBOL OCCURS clause is not allowed on level 01."

Explanation: The OCCURS clause must be used with a data item that is declared on a level greater than 01. This error is only a warning, and treats the data item correctly (as though declared as a COBOL table). A warning may also be generated by the COBOL compiler.

E_E4000E "PICTURE '%0c' is too long. The maximum length is %1c."

Explanation: COBOL PICTURE strings must not exceed the maximum length specified in the error message. Try to collapse consecutive occurrences of the same PICTURE symbol into a repeat count.

For example: PICTURE S99999999 becomes PICTURE S9(8)

E_E4000F

"PICTURE '%0c' contains non-integer repeat count, %1c."

Explanation: A COBOL repeat count in a PICTURE string was either too long or was not an integer. The preprocessor treats the data item as though declared with a PICTURE with a repeat count of 1. For example: S9(1) or X(1)

E_E40011

"USAGE type '%0c' is not supported."

Explanation: his usage type is currently not supported.

E_E40012

"PICTURE '%0c' has two sign symbols (S)."

Explanation: The specified numeric PICTURE string has two sign symbols. The preprocessor will treat the data item as though it was declared:

PICTURE S9(8) USAGE COMP.

E_E40013

"PICTURE '%0c' has two decimal point symbols (V)."

Explanation: The specified numeric PICTURE string has two decimal point symbols. The preprocessor will treat the data item as though it was declared:

PICTURE S9(8) USAGE COMP.

E_E40014

"Missing quotation mark on continued string literal."

Explanation: The first non-blank character of a continued string literal must be a quotation mark in the indicator area. A missing quotation mark in the continued string literal or the wrong quotation mark will generate this error.

E_E40015

"COBOL data item '%0c' is a table and must be subscripted."

Explanation: The data item is a COBOL table and must be subscripted in order to yield an elementary data item to retrieve or set Ingres data.

E_E40016

"COBOL data item '%0c' is not a table and must not be subscripted."

Explanation: You have included subscripts when referring to a data item that was not declared as a COBOL table.

E_E40017

"Duplicate COBOL data declaration '%0c' clause found."

Explanation: You have included either a duplicate USAGE, PICTURE or OCCURS data declaration clause when declaring a data item.

Sample Applications

This section contains sample applications. Samples are shown for the UNIX, Windows, and VMS environments.

The Department-Employee Master/Detail Application

This application uses two database tables joined on a specific column. This typical example of a department and its employees demonstrates how to process two tables as a master and a detail.

The program scans through all the departments in a database table, in order to reduce expenses. Based on certain criteria, the program updates department and employee records. The conditions for updating the data are the following:

Departments:

- If a department has made less than $50,000 in sales, the department is dissolved.

Employees:

- If an employee was hired since the start of 1985, the employee is terminated.
- If the employee's yearly salary is more than the minimum company wage of $14,000 and the employee is not nearing retirement (over 58 years of age), the employee takes a 5% pay cut.
- If the employee's department is dissolved and the employee is not terminated, the employee is moved into a state of limbo to be resolved by a supervisor.

This program uses two cursors in a master/detail fashion. The first cursor is for the Department table, and the second cursor is for the Employee table. Both tables are described in declare table statements at the start of the program. The cursors retrieve all the information in the tables, some of which is updated. The cursor for the Employee table also retrieves an integer date interval whose value is positive if the employee was hired after January 1, 1985. The tables contain no null values.

Each row that is scanned, from both the Department table and the Employee table, is recorded into the system output file. This file serves both as a log of the session and as a simplified report of the updates that were made.

Each section of code is commented for the purpose of the application and also to clarify some of the uses of the Embedded SQL statements. The program illustrates table creation, multi-statement transactions, all cursor statements, direct updates and error handling.

Windows **UNIX**

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  EXPENSE-PROCESS.

ENVIRONMENT DIVISION.

DATA DIVISION.
WORKING-STORAGE SECTION.
        EXEC SQL INCLUDE SQLCA END-EXEC.
        EXEC SQL BEGIN DECLARE SECTION END-EXEC.
*       The department table
        EXEC SQL DECLARE dept TABLE
            (name         char(12)       NOT NULL,
             totsales     decimal(9,2)   NOT NULL,
             employees    smallint       NOT NULL)
        END-EXEC.

*       The employee table
        EXEC SQL DECLARE employee TABLE
            (name            char(20)     NOT NULL,
             age             integer1     NOT NULL,
             idno            integer      NOT NULL,
             hired           date         NOT NULL,
             dept            char(12)     NOT NULL,
             salary          decimal(8,2) NOT NULL)
        END-EXEC.

*       "State-of-Limbo" for employees who lose their department
        EXEC SQL DECLARE toberesolved TABLE
           (name             char(20)     NOT NULL,
            age              integer1     NOT NULL,
            idno             integer      NOT NULL,
            hired            date         NOT NULL,
            dept             char(12)     NOT NULL,
            salary           decimal(8,2) NOT NULL)
        END-EXEC.

*      Minimum sales of department
       01    MIN-DEPT-SALES    PIC S9(5)V9(2) USAGE COMP
                                VALUE IS 50000.00.
*      Minimum employee salary
       01    MIN-EMP-SALARY    PIC S9(5)V9(2) USAGE COMP
                                VALUE IS 14000.00.
*       Age above which no salary-reduction will be made
       01    NEARLY-RETIRED    PIC S9(2) USAGE COMP
                                        VALUE IS 58.
*      Salary-reduction percentage
       01   SALARY-REDUC    PIC S9(1)V9(2) USAGE COMP
                                       VALUE IS 0.95.
*      Record corresponding to the "dept" table.
       01      DEPT.
               02 DNAME         PIC X(12).
               02 TOTSALES      PIC S9(7)V9(2) USAGE COMP.
               02 EMPLOYEES     PIC S9(4) USAGE COMP.
*      Record corresponding to the "employee" table
       01      EMP.
                02 ENAME          PIC X(20).
                02 AGE            PIC S9(2) USAGE COMP.
                02 IDNO           PIC S9(8) USAGE COMP.
                02 HIRED          PIC X(26).
                02 SALARY         PIC S9(6)V9(2) USAGE COMP.
                02 HIRED-SINCE-85 PIC S9(4) USAGE COMP.
*      Count of employees terminated.
       01      EMPS-TERM          PIC S99 USAGE COMP.
```

```
*      Indicates whether the employee's dept was deleted
       01      DELETED-DEPT       PIC S9 USAGE COMP.
*      Error message buffer used by CHECK-ERRORS.
       01     ERRBUF              PIC X(200).
*      Formatting values for output
       01      DEPT-OUT.
                02 FILLER        PIC X(12) VALUE "Department: ".
                02 DNAME-OUT     PIC X(12).
                02 FILLER        PIC X(13) VALUE "Total Sales: ".
                02 TOTSALES-OUT  PIC $,$$$,$$9.9(2) USAGE DISPLAY.
                02 DEPT-FORMAT   PIC X(19).
       01      EMP-OUT.
                02 FILLER          PIC XX VALUE SPACES.
                02 TITLE           PIC X(11).
                02 IDNO-OUT        PIC Z9(6) USAGE DISPLAY.
                02 FILLER          PIC X VALUE SPACE.
                02 ENAME-OUT       PIC X(20).
                02 AGE-OUT         PIC Z9(2) USAGE DISPLAY.
                02 FILLER          PIC XX VALUE SPACES.
                02 SALARY-OUT      PIC $$$,$$9.9(2) USAGE DISPLAY.
                02 FILLER          PIC XX VALUE SPACES.
                02 DESCRIPTION      PIC X(24).
       EXEC SQL END DECLARE SECTION END-EXEC.

**
* Procedure Division
*
*     Initialize the database, process each department and
*     terminate the session.
**
      PROCEDURE DIVISION.
      EXAMPLE SECTION.
      XBEGIN.
      DISPLAY "Entering application to process expenses".
      PERFORM INIT-DB THRU END-INITDB.
      PERFORM PROCESS-DEPTS THRU END-PROCDEPTS.
      PERFORM END-DB THRU END-ENDDB.
      DISPLAY "Successful completion of application".
      STOP RUN.
**
* Paragraph: INIT-DB
*
*     Start up the database, and abort if there is an error
*     Before processing employees, create the table for employees
*     who lose their department, "toberesolved".
**
      INIT-DB.
      EXEC SQL WHENEVER SQLERROR STOP END-EXEC.
      EXEC SQL CONNECT personnel END-EXEC.
      DISPLAY "Creating ""To_Be_Resolved"" table".
      EXEC SQL CREATE TABLE toberesolved
                (name     char(20),
                 age      integer1,
                 idno     integer,
                 hired    date,
                 dept     char(12),
                 salary   decimal(8,2)
      END-EXEC.
      END-INITDB.
      EXIT.
```

```
**
* Paragraph: END-DB
*
*     Commit the multi-statement transaction and close access to
*     the database after successful completion of the application.
**
      END-DB.
      EXEC SQL COMMIT END-EXEC.
      EXEC SQL DISCONNECT END-EXEC.
      END-ENDDB.
      EXIT.
**
* Paragraph: PROCESS-DEPTS
*
*    Scan through all the departments, processing each one.
*    If the department has made less than $50,000 in sales, then
*    the department is dissolved. For each department process
*    all the employees (they may even be moved to another table).
*    If an employee was terminated, then update the department's
*    employee counter.
**
      PROCESS-DEPTS.
      EXEC SQL DECLARE deptcsr CURSOR FOR
              SELECT name, totsales, employees
              FROM dept
              FOR DIRECT UPDATE OF name, employees
              END-EXEC.
*     All errors from this point on close down the application.
      EXEC SQL WHENEVER SQLERROR GOTO CLOSE-DOWN END-EXEC.
      EXEC SQL WHENEVER NOT FOUND GOTO CLOSE-DEPT-CSR END-EXEC.
      EXEC SQL OPEN deptcsr END-EXEC.
      PERFORM UNTIL SQLCODE NOT = 0
           EXEC SQL FETCH deptcsr INTO :DEPT END-EXEC
*          Did the department reach minimum sales?
           IF TOTSALES < MIN-DEPT-SALES THEN
                  EXEC SQL DELETE FROM dept
                       WHERE CURRENT OF deptcsr END-EXEC
                  MOVE 1 TO DELETED-DEPT
                  MOVE " -- DISSOLVED --" TO DEPT-FORMAT
           ELSE
                  MOVE 0 TO DELETED-DEPT
                  MOVE SPACES TO DEPT-FORMAT
           END-IF
*          Log what we have just done
           MOVE DNAME    TO DNAME-OUT
           MOVE TOTSALES TO TOTSALES-OUT
           DISPLAY DEPT-OUT
*          Now process each employee in the department
           PERFORM PROCESS-EMPLOYEES THRU END-PROCEMPLOYEES
*          If some employees were terminated, record this fact
           IF EMPS-TERM > 0 AND DELETED-DEPT = 0 THEN
                EXEC SQL UPDATE dept
                      SET employees = :EMPLOYEES - :EMPS-TERM
                      WHERE CURRENT OF deptcsr END-EXEC
           END-IF
        END-PERFORM.

        CLOSE-DEPT-CSR.
            EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
            EXEC SQL CLOSE deptcsr END-EXEC.
            END-PROCDEPTS.
            EXIT.
```

```
**
* Paragraph: PROCESS-EMPLOYEES
*
*    Scan through all the employees for a particular department.
*    Based on given conditions the employee may be terminated, or
*    given a salary reduction:
*      1.  If an employee was hired since 1985 then the employe
*          is terminated.
*      2.  If the employee's yearly salary is more than the
*          minimum company wage of $14,000 and the employee is
*          not close to retirement (over 58 years of age), then
*          the employee takes a 5% salary reduction.
*      3.  If the employee's department is dissolved and the
*          employee is not terminated, then the employee is moved
*          into the "toberesolved" table.
**
       PROCESS-EMPLOYEES.
*      Note the use of the Ingres functions to find out
*      who was hired since 1985.
       EXEC SQL DECLARE empcsr CURSOR FOR
                  SELECT name, age, idno, hired, salary,
                    int4(interval('days', hired -
date('01-jan-1985')))
                  FROM employee
                  WHERE dept = :DNAME
                  FOR DIRECT UPDATE OF name, salary
                  END-EXEC.
*      All errors from this point on close down the application.
       EXEC SQL WHENEVER SQLERROR GOTO CLOSE-DOWN END-EXEC.
       EXEC SQL WHENEVER NOT FOUND GOTO CLOSE-EMP-CSR END-EXEC.
       EXEC SQL OPEN empcsr END-EXEC.
*      Record how many employees are terminated
       MOVE 0 TO EMPS-TERM.
       PERFORM UNTIL SQLCODE NOT = 0
            EXEC SQL FETCH empcsr INTO :EMP END-EXEC
            IF HIRED-SINCE-85 > 0 THEN
                  EXEC SQL DELETE FROM employee
                      WHERE CURRENT OF empcsr END-EXEC
                  MOVE "Terminated:" TO TITLE
                  MOVE "Reason: Hired since 1985." TO DESCRIPTION
                  ADD 1 TO EMPS-TERM
            ELSE
*                 Reduce salary if not nearly retired
                  IF SALARY > MIN-EMP-SALARY THEN
                      IF AGE < NEARLY-RETIRED THEN
                          EXEC SQL UPDATE employee
                              SET salary = salary * :SALARY-REDUC
                              WHERE CURRENT OF empcsr END-EXEC
                          MOVE "Reduction: " TO TITLE
                          MOVE "Reason: Salary." TO DESCRIPTION
                      ELSE
*                         Do not reduce salary
                          MOVE "No Changes:" TO TITLE
                          MOVE "Reason: Retiring." TO DESCRIPTION
                      END-IF
```

```
*                 Leave employee alone
                  ELSE
                      MOVE "No Changes:" TO TITLE
                      MOVE "Reason: Salary." TO DESCRIPTION
                  END-IF
*                 Was employee's department dissolved?
                  IF DELETED-DEPT = 1 THEN
                      EXEC SQL INSERT INTO toberesolved
                          SELECT * FROM employee
                          WHERE idno = :IDNO END-EXEC
                      EXEC SQL DELETE FROM employee
                          WHERE CURRENT OF empcsr END-EXEC
                  END-IF
            END-IF
*           Log the employee's information
            MOVE IDNO     TO IDNO-OUT
            MOVE ENAME    TO ENAME-OUT
            MOVE AGE      TO AGE-OUT
            MOVE SALARY   TO SALARY-OUT
            DISPLAY EMP-OUT
        END-PERFORM.

CLOSE-EMP-CSR.
      EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC
      EXEC SQL CLOSE empcsr END-EXEC.
END-PROCEMPLOYEES.
      EXIT.
**
* Paragraph: CLOSE-DOWN
*
*    This paragraph serves as an error handler called any time
*    after INIT-DB has successfully completed. In all cases, it
*    prints the cause of the error, and aborts the transaction,
*    backing ou changes. Note that disconnecting from the
*    database will implicitly close any open cursors too.
**
CLOSE-DOWN.
*     Turn off error handling
      EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
      EXEC SQL INQUIRE_SQL(:ERRBUF = ERRORTEXT) END-EXEC.
      DISPLAY "Closing Down because of database error:".
      DISPLAY ERRBUF.
      EXEC SQL ROLLBACK END-EXEC.
      EXEC SQL DISCONNECT END-EXEC.
      STOP RUN.
```

VMS

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXPENSE-PROCESS.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OUT-FILE ASSIGN TO "EXPENSES.LOG".
```

```
DATA DIVISION.
FILE SECTION.
 FD  OUT-FILE
      LABEL RECORD IS OMITTED.
 01  PRINT-OUT     PIC X(80).
WORKING-STORAGE SECTION.
EXEC SQL INCLUDE SQLCA END-EXEC.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
* The department table
EXEC SQL DECLARE dept TABLE
        (name         char(12) NOT NULL,
         totsales      decimal(14,2) NOT NULL,
         employees     smallint NOT NULL)
        END-EXEC.
* The employee table
EXEC SQL DECLARE employee TABLE
        (name         char(20) NOT NULL,
         age          integer1 NOT NULL,
         idno         integer NOT NULL,
         hired        date NOT NULL,
         dept         char(12) NOT NULL,
         salary       decimal(14,2) NOT NULL)
      END-EXEC.
* "State-of-Limbo" for employees who lose their department
EXEC SQL DECLARE toberesolved TABLE
      (name       char(20) NOT NULL,
       age        integer1 NOT NULL,
       idno       integer NOT NULL,
       hired      date NOT NULL,
       dept       char(12) NOT NULL,
       salary     decimal(14,2) NOT NULL)
       END-EXEC.
* Minimum sales of department
    01   MIN-DEPT-SALES        USAGE COMP-2 VALUE IS 50000.00.
* Minimum employee salary
    01   MIN-EMP-SALARY        USAGE COMP-2 VALUE IS 14000.00.
* Age above which no salary-reduction will be made
    01   NEARLY-RETIRED        PIC S9(2) USAGE COMP VALUE IS 58.
* Salary-reduction percentage
    01   SALARY-REDUC      USAGE COMP-1 VALUE IS 0.95.
* Record corresponding to the "dept" table.
    01   DEPT.
        02 NAME           PIC X(12).
        02 TOTSALES       USAGE COMP-2.
        02 EMPLOYEES      PIC S9(4) USAGE COMP.
* Record corresponding to the "employee" table
    01   EMP.
        02 NAME           PIC X(20).
        02 AGE            PIC S9(2) USAGE COMP.
        02 IDNO           PIC S9(6) USAGE COMP.
        02 HIRED          PIC X(26).
        02 SALARY         USAGE COMP-2.
        02 HIRED-SINCE-85 PIC S9(4) USAGE COMP.
* Count of employees terminated.
    01   EMPS-TERM         PIC S99 USAGE COMP.
* Indicates whether the employee's dept was deleted
    01   DELETED-DEPT      PIC S9 USAGE COMP.
* Indicates whether "toberesolved" table exists in INIT-DB paragraph.
    01   FOUND-TABLE       PIC S9 USAGE COMP.
* Error message buffer used by CLOSE-DOWN
    01   ERRBUF            PIC X(200).
EXEC SQL END DECLARE SECTION END-EXEC.
```

```
* Formatting values for output
 01   DEPT-OUT.
      02 FILLER        PIC X(12) VALUE "Department: ".
      02 DNAME         PIC X(12).
      02 FILLER        PIC X(13) VALUE "Total Sales: ".
      02 TOTSALES-OUT  PIC $,$$$,$$9.9(2) USAGE DISPLAY.
      02 DEPT-FORMAT   PIC X(19).
  01   EMP-OUT.
      02 TITLE          PIC X(11).
      02 IDNO-OUT       PIC Z9(6) USAGE DISPLAY.
      02 FILLER         PIC X VALUE SPACE.
      02 ENAME          PIC X(20).
      02 AGE-OUT        PIC Z9(2) USAGE DISPLAY.
      02 FILLER         PIC XX VALUE SPACES.
      02 SALARY-OUT     PIC $$$,$$9.9(2) USAGE DISPLAY.
      02 FILLER         PIC XX VALUE SPACES.
      02 DESCRIPTION    PIC X(24).
PROCEDURE DIVISION.
SBEGIN.
* Initialize the database, process each department and
* terminate the session.
    DISPLAY "Entering application to process expenses".
    PERFORM INIT-DB THRU END-INITDB.
    PERFORM PROCESS-DEPTS THRU END-PROCDEPTS.
    PERFORM END-DB THRU END-ENDDB.
    DISPLAY "Successful completion of application".
    STOP RUN.
INIT-DB.
* This paragraph connects to the database and aborts if an error.
* Before processing employees, create the table for employees who
* lose their department, "toberesolved".
    OPEN OUTPUT OUT-FILE.
    MOVE SPACES TO PRINT-OUT.
    EXEC SQL WHENEVER SQLERROR STOP END-EXEC.
    EXEC SQL CONNECT personnel END-EXEC.
    MOVE ZERO TO FOUND-TABLE.
* Does the table exist?
    EXEC SQL SELECT 1
          INTO :FOUND-TABLE
          FROM iitables
          WHERE table_name = 'toberesolved'
          END-EXEC.
* If not, then create it.
    IF FOUND-TABLE = 0 THEN
        DISPLAY "Creating ""To_Be_Resolved"" table."
        EXEC SQL CREATE TABLE toberesolved
            (name       char(20)      NOT NULL,
             age        integer1      NOT NULL,
             idno       integer       NOT NULL,
             hired      date          NOT NULL,
             dept       char(12)      NOT NULL,
             salary     decimal(14,2) NOT NULL)
        END-EXEC.
END-INITDB.
END-DB.
* Commit the multi-statement transaction and access to the
* database.
      EXEC SQL COMMIT END-EXEC.
      EXEC SQL DISCONNECT END-EXEC.
      CLOSE OUT-FILE.
END-ENDDB.
PROCESS-DEPTS.
```

```
* This paragraph scans through all the departments, processing
* each one. If the department has made less than $50,000 in
* sales, then the department is dissolved. All employees in each
* department are processed (they may even be moved to another
* table). If an employee is terminated, the department's employee
* counter is updated.
      EXEC SQL DECLARE deptcsr CURSOR FOR
            SELECT name, totsales, employees
            FROM dept
            FOR DIRECT UPDATE OF name, employees
            END-EXEC.
* All errors from this point on close down the application.
      EXEC SQL WHENEVER SQLERROR GOTO CLOSE-DOWN END-EXEC.
      EXEC SQL WHENEVER NOT FOUND GOTO CLOSE-DEPT-CSR END-EXEC.
      EXEC SQL OPEN deptcsr END-EXEC.
      PERFORM UNTIL SQLCODE NOT = 0
          EXEC SQL FETCH deptcsr INTO :DEPT END-EXEC
* Did the department reach minimum sales?
      IF TOTSALES < MIN-DEPT-SALES THEN
          EXEC SQL DELETE FROM dept
                WHERE CURRENT OF deptcsr
                END-EXEC
          MOVE 1 TO DELETED-DEPT
          MOVE " -- DISSOLVED --" TO DEPT-FORMAT
      ELSE
          MOVE 0 TO DELETED-DEPT
          MOVE " " TO DEPT-FORMAT
      END-IF
* Log what we have just done.
      MOVE NAME IN DEPT TO DNAME
      MOVE TOTSALES TO TOTSALES-OUT
      WRITE PRINT-OUT FROM DEPT-OUT
* Now process each employee in the department.
      PERFORM PROCESS-EMPLOYEES THRU END-PROCEMPLOYEES
* If some employees were terminated, record this fact.
          IF EMPS-TERM > 0 AND DELETED-DEPT = 0 THEN
               EXEC SQL UPDATE dept
                      SET employees = :EMPLOYEES - :EMPS-TERM
                      WHERE CURRENT OF deptcsr
                      END-EXEC
          END-IF
      END-PERFORM.
CLOSE-DEPT-CSR.
      EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
      EXEC SQL CLOSE deptcsr END-EXEC.
END-PROCDEPTS.
PROCESS-EMPLOYEES.
* This paragraph scans through all the employees for a
* particular department.
* Based on given conditions, the employee may be terminated
* or given a salary
* reduction:
* 1. If an employee was hired since 1985, then the employee
*    is terminated.
*
* 2. If the employee's yearly salary is more than the
*    minimum company wage of $14,000 and the employee is not
*    close to retirement (over 58 years of age), then the
*    employee takes a 5% salary reduction.
*
* 3. If the employee's department is dissolved and the
*    employee is not terminated, then the employee is moved into
*    the "toberesolved" table.
*
* Note the use of the Ingres functions to find out who has
* been hired since 1985.
```

```
      EXEC SQL DECLARE empcsr CURSOR FOR
            SELECT name, age, idno, hired, salary,
                int4(interval('days', hired -
                date('01-jan-1985')))
            FROM employee
            WHERE dept = :DEPT.NAME
            FOR DIRECT UPDATE OF name, salary
            END-EXEC.
* All errors from this point on close down the application.
      EXEC SQL WHENEVER SQLERROR GOTO CLOSE-DOWN END-EXEC.
      EXEC SQL WHENEVER NOT FOUND GOTO CLOSE-EMP-CSR END-EXEC.
      EXEC SQL OPEN empcsr END-EXEC.
* Record how many employees are terminated.
      MOVE 0 TO EMPS-TERM.
      PERFORM UNTIL SQLCODE NOT = 0
          EXEC SQL FETCH empcsr INTO :EMP END-EXEC
          IF HIRED-SINCE-85 > 0 THEN
                EXEC SQL DELETE FROM employee
                     WHERE CURRENT OF empcsr;
                MOVE "Terminated:" TO TITLE
                MOVE "Reason: Hired since 1985."TO DESCRIPTION
                ADD 1 TO EMPS-TERM
          ELSE
* Reduce salary if not nearly retired.
                IF SALARY > MIN-EMP-SALARY THEN
                     IF AGE < NEARLY-RETIRED THEN
                           EXEC SQL UPDATE employee
                                SET salary = salary *
                                                :SALARY-REDUC
                                WHERE CURRENT OF empcsr
                                END-EXEC
                           MOVE "Reduction: " TO TITLE
                           MOVE "Reason: Salary."TO DESCRIPTION
                      ELSE
* Do not reduce salary.
                           MOVE "No Changes:" TO TITLE
                           MOVE "Reason: Retiring."TO DESCRIPTION
                      END-IF
* Leave employee alone.
               ELSE
                    MOVE "No Changes:" TO TITLE
                    MOVE "Reason: Salary."TO DESCRIPTION
               END-IF
* Was employee's department dissolved?
               IF DELETED-DEPT = 1 THEN
                    EXEC SQL INSERT INTO toberesolved
                        SELECT *
                        FROM employee
                        WHERE idno = :IDNO
                        END-EXEC
                    EXEC SQL DELETE FROM employee
                         WHERE CURRENT OF empcsr END-EXEC
               END-IF
            END-IF
* Log the employee's information.
            MOVE IDNO        TO IDNO-OUT
            MOVE NAME IN EMP TO ENAME
            MOVE AGE         TO AGE-OUT
            MOVE SALARY      TO SALARY-OUT
            WRITE PRINT-OUT FROM EMP-OUT
      END-PERFORM.
CLOSE-EMP-CSR.
      EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC
      EXEC SQL CLOSE empcsr END-EXEC.
END-PROCEMPLOYEES.
CLOSE-DOWN.
```

```
* This paragraph serves as an error handler called any time after
* INIT-DB has successfully completed. In all cases, it prints
* the cause of the error and aborts the transaction, backing
* out changes.
* Note that disconnecting from the database will implicitly close
* any open cursors.

* Turn off error handling
        EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC
        EXEC SQL INQUIRE_SQL(:ERRBUF = ERRORTEXT) END-EXEC.
        DISPLAY "Closing Down because of database error:".
        DISPLAY ERRBUF.
        EXEC SQL ROLLBACK END-EXEC.
        EXEC SQL DISCONNECT END-EXEC.
        STOP RUN.
```

The Table Editor Table Field Application

This application edits the Person table in the Personnel database. It is a forms application that allows the user to update a person's values, remove the person, or add new persons. Various table field utilities are provided with the application to demonstrate how they work.

The objects used in this application are shown in the following table:

Object	Description
personnel	The program's database environment.
person	A table in the database, with three columns: name (char(20)) age (smallint) number (integer) Number is unique.
personfrm	The VIFRED form with a single table field.
persontbl	A table field in the form, with two columns: name (char(20)) age (integer) When initialized, the table field includes the hidden number (integer) column.
personrec	A local structure, whose members correspond in name and type to columns in the Person table and the Persontbl table field.

At the start of the application, a database cursor is opened to load the table field with data from the Person table. Once the table field has been loaded, the user can browse and edit the displayed values. Entries can be added, updated or deleted. When finished, the values are unloaded from the table field, and the user's updates are transferred back into the Person table.

Windows **UNIX**

```
**
* Program: TABLE-EDIT
*
* Table Editor program. The main program initializes
* the database and displays a form that contains a

* single table field of personnel. It allows the user

* to add, change or delete the rows in the field.
* The program then makes the changes to the
* underlying database table in a multi-statement
* transaction.
**
       IDENTIFICATION DIVISION.
       PROGRAM-ID. TABLE-EDIT.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       EXEC SQL INCLUDE SQLCA END-EXEC.
       EXEC SQL DECLARE person TABLE
                (name    char(20),
                age     smallint,
                number  integer)
        END-EXEC.
       EXEC SQL BEGIN DECLARE SECTION END-EXEC.
*      Person information

         01 PERSONREC.
            02 PNAME     PIC X(20)
            02 P-AGE     PIC S99 USAGE COMP.
            02 PNUMBER   PIC S9(6) USAGE COMP.
         01 MAXID      PIC S9(6) USAGE COMP.
*      Table field entry information
         01 RECNUM     PIC S9(4) USAGE COMP.
         01 LASTROW     PIC S9 USAGE COMP.
*      Utility buffers
         01 MSGBUF        PIC X(200).
         01 RESPBUF       PIC X(20).
         01 STATE         PIC S9 USAGE COMP.
       EXEC SQL END DECLARE SECTION END-EXEC.
*      Table field row states:
*      Empty or undefined row
       01 ST-UNDEF       PIC S9 USAGE COMP VALUE 0.
*      Appended by user
       01 ST-NEW         PIC S9 USAGE COMP VALUE 1.
*      Loaded by program - not updated
       01 ST-UNCHANGED   PIC S9 USAGE COMP VALUE 2.
*      Loaded by program - since changed
       01 ST-CHANGE      PIC S9 USAGE COMP VALUE 3.
*      Deleted by program
       01 ST-DELETE      PIC S9 USAGE COMP VALUE 4.
*      SQLCA value for no rows
       01 NOT-FOUND      PIC S9(3) USAGE COMP VALUE 100.
*      Update error from database
       01 UPDATE-ERROR        PIC S9(2) USAGE COMP.
```

```
*       Transaction aborted
        01 XACT-ABORTED        PIC S9 USAGE COMP.
        PROCEDURE DIVISION.
        EXAMPLE SECTION.
        XBEGIN.
*       Set up error handling for main program
        EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC.
        EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
        EXEC SQL WHENEVER SQLERROR STOP END-EXEC.
*        Start Ingres and the Ingres/FORMS system
        EXEC SQL CONNECT personnel END-EXEC.
        EXEC FRS FORMS END-EXEC.
*       Verify that the user can edit the "person" table
        EXEC FRS PROMPT NOECHO
                ('Password for table editor: ', :RESPBUF)
        END-EXEC.
        IF RESPBUF NOT = "MASTER_OF_ALL" THEN
                 EXEC FRS MESSAGE 'No permission for task.
                      Exiting . . .' END-EXEC
                 EXEC FRS ENDFORMS END-EXEC
                 EXEC SQL DISCONNECT END-EXEC
                 STOP RUN.
*       We assume no SQL errors can happen during screen updating
          EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
        EXEC FRS MESSAGE 'Initializing Person Form . . .' END-EXEC.
        EXEC FRS FORMINIT personfrm END-EXEC.
*        Initialize "persontbl" table field with a data set in FILL
*        mode, so that the runtime user can append rows. To keep
*        track of events occurring to original rows loaded into the
*        table field, hide the unique person number.
        EXEC FRS INITTABLE personfrm persontbl FILL
                                 (number = integer)
          END-EXEC.
        PERFORM LOAD-TABLE THROUGH ENDLOAD-TABLE.
        EXEC FRS DISPLAY personfrm UPDATE END-EXEC
        EXEC FRS INITIALIZE END-EXEC
        EXEC FRS ACTIVATE MENUITEM 'Top' END-EXEC
        EXEC FRS BEGIN END-EXEC
*       Provide menu items, as well as the system FRS key,
*       to scroll to both extremes of the table field.
            EXEC FRS SCROLL personfrm persontbl TO 1 END-EXEC.
        EXEC FRS END END-EXEC
        EXEC FRS ACTIVATE MENUITEM 'Bottom' END-EXEC
        EXEC FRS BEGIN END-EXEC
            EXEC FRS SCROLL personfrm persontbl TO END END-EXEC.
        EXEC FRS END END-EXEC
        EXEC FRS ACTIVATE MENUITEM 'Remove' END-EXEC
        EXEC FRS BEGIN END-EXEC
*            Remove the person in the row the user's cursor is on.
*            If there are no persons, exit operation with message.
*            Note that this check cannot really happen, as there
*            is always an UNDEFINED row in FILL mode.
             EXEC FRS INQUIRE_FRS table personfrm
                      (:LASTROW = LASTROW(persontbl)) END-EXEC.
             IF LASTROW = 0 THEN
                   EXEC FRS MESSAGE 'Nobody to Remove' END-EXEC
                   EXEC FRS SLEEP 2 END-EXEC
                   EXEC FRS RESUME FIELD persontbl END-EXEC.
                   EXEC FRS DELETEROW personfrm persontbl END-EXEC.
        EXEC FRS END END-EXEC
        EXEC FRS ACTIVATE MENUITEM 'Find' END-EXEC
        EXEC FRS BEGIN END-EXEC
*        Scroll user to the requested table field entry. Prompt
*        the user for a name, and if one is typed in, loop through
*        the data set searching for it.
            MOVE SPACES TO RESPBUF.
```

```
                EXEC FRS PROMPT ('Person''s name : ', :RESPBUF)
                                        END-EXEC.
                IF RESPBUF = SPACES THEN
                     EXEC FRS RESUME FIELD persontbl END-EXEC.
                     EXEC FRS UNLOADTABLE personfrm persontbl
                        (:PNAME = name,
                         :RECNUM = _record,
                         :STATE = _state)
                        END-EXEC
                EXEC FRS BEGIN END-EXEC
*                   Compare name typed in with names in table, but do
*                   not compare with deleted rows.
                      IF PNAME = RESPBUF AND
                          STATE NOT = ST-DELETE THEN
                           EXEC FRS SCROLL personfrm persontbl
                              TO :RECNUM END-EXEC
                           EXEC FRS RESUME FIELD persontbl END-EXEC.
                EXEC FRS END END-EXEC.
*               Fell out of loop without finding name. Inform user.
                STRING "Person """, RESPBUF,
                      """ not found in table [HIT RETURN] "
                      DELIMITED BY SIZE INTO MSGBUF.
                EXEC FRS PROMPT NOECHO (:MSGBUF, :RESPBUF) END-EXEC.
            EXEC FRS END END-EXEC
            EXEC FRS ACTIVATE MENUITEM 'Exit' END-EXEC
            EXEC FRS BEGIN END-EXEC
                EXEC FRS VALIDATE FIELD persontbl END-EXEC.
                EXEC FRS BREAKDISPLAY END-EXEC.
            EXEC FRS END END-EXEC
            EXEC FRS FINALIZE END-EXEC.
*           Exit person table editor and unload the table field.
*           If any updates, deletions or additions were made,
*           duplicate these changes in the source table. If the
*           user added new people, assign a unique person id to
*           each person before adding the person to the table. To
*           do this, increment the previously-saved maximum id
*           number with each insert.
*           Do all the updates in a transaction
            EXEC SQL COMMIT WORK END-EXEC.
*           Hard code the error handling in the UNLOADTABLE
*           loop, as we want to cleanly exit the loop.
            EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
            MOVE 0 TO UPDATE-ERROR.
            MOVE 0 TO XACT-ABORTED.
            EXEC FRS MESSAGE
                    'Exiting Person Application .  .  .' END-EXEC.
            EXEC FRS UNLOADTABLE personfrm persontbl
                    (:PNAME = name, :P-AGE = age,
                     :PNUMBER = number, :STATE = _state)
                  END-EXEC
            EXEC FRS BEGIN END-EXEC
*                Row appended by user.  Insert into "person" table
*                with new unique id.
                 IF STATE = ST-NEW THEN
                         ADD 1 TO MAXID
                         EXEC SQL REPEATED INSERT INTO person
                             VALUES (:PNAME, :P-AGE, :MAXID) END-EXEC
*                Row updated by user.  Reflect in table.
                 ELSE IF STATE = ST-CHANGE THEN
                        EXEC SQL REPEATED UPDATE person SET
                            name = :PNAME, age = :P-AGE
                            WHERE number = :PNUMBER
                            END-EXEC
```

```
*           Row deleted by user, so delete from table. Note that
*           rows appended by the user at runtime and the
*           deleted are not saved and are therefore not unloaded.
            ELSE IF STATE = ST-DELETE THEN
                   EXEC SQL REPEATED DELETE FROM person
                           WHERE number = :PNUMBER END-EXEC
            END-IF.
*           Else rows are UNDEFINED or UNCHANGED. No updates.
*           Handle error conditions: if an error occurred, abort
*           the transaction. If no rows were updated, inform user
*           and prompt for continuation.
            IF SQLCODE < 0 THEN
                    EXEC SQL
                        INQUIRE_SQL(:MSGBUF = ERRORTEXT) END-EXEC
                    EXEC SQL ROLLBACK WORK END-EXEC
                    MOVE 1 TO UPDATE-ERROR
                    MOVE 1 TO XACT-ABORTED
                    EXEC FRS ENDLOOP END-EXEC
            ELSE IF SQLCODE = NOT-FOUND THEN
                   STRING "Person """, PNAME,
                       """ not updated. Abort all updates? "
                       DELIMITED BY SIZE INTO MSGBUF
                   EXEC FRS PROMPT (:MSGBUF, :RESPBUF) END-EXEC
                   IF RESPBUF = "Y" OR RESPBUF = "y" THEN
                             EXEC SQL ROLLBACK WORK END-EXEC
                             MOVE 1 TO XACT-ABORTED
                             EXEC FRS ENDLOOP END-EXEC
                   END-IF
            END-IF.
       EXEC FRS END END-EXEC.
       IF XACT-ABORTED = 0 THEN
                EXEC SQL COMMIT END-EXEC.
       EXEC FRS ENDFORMS END-EXEC.
       EXEC SQL DISCONNECT END-EXEC.
       IF UPDATE-ERROR = 1 THEN
               DISPLAY
                  "Your updates were aborted because of error:"
               DISPLAY msgbuf.
       STOP RUN.
**
* Paragraph: LOAD-TABLE
*
* This paragraph opens a database cursor to load the table
* field with data from the "person" table. The columns
* "name" and "age" will be displayed, and "number" will be
* hidden. It sets the maximum employee number.
**
     LOAD-TABLE.
     EXEC SQL DECLARE loadtab CURSOR FOR
              SELECT name, age, number
              FROM person
              END-EXEC.
*    Set up error handling for loading procedure
     EXEC SQL WHENEVER SQLERROR GOTO LOAD-END END-EXEC.
     EXEC SQL WHENEVER NOT FOUND GOTO LOAD-END END-EXEC.
     EXEC FRS MESSAGE
            'Loading Person Information .  .  .' END-EXEC.
*    Fetch the maximum person id number for later use
     EXEC SQL SELECT MAX(number) INTO :MAXID
            FROM person END-EXEC.
     EXEC SQL OPEN loadtab END-EXEC.
     PERFORM UNTIL SQLCODE NOT = 0
```

```
*            Fetch data into record and load table field
             EXEC SQL FETCH loadtab INTO :PERSONREC END-EXEC
             EXEC FRS LOADTABLE personfrm persontbl
              (name = :PNAME, age = :P-AGE, number = :PNUMBER)
                  END-EXEC
END-PERFORM.
LOAD-END.
     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
     EXEC SQL CLOSE loadtab END-EXEC.
ENDLOAD-TABLE.
     EXIT
```

VMS

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TABLE-EDIT.
* Table Editor program. The main program initializes the database
* and displays a form that contains a single table field of
* personnel. It allows the user to add, change or delete the rows
* in the field. The program then makes the changes to the
* underlying database table in a multi-statement transaction.

ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
EXEC SQL INCLUDE SQLCA END-EXEC.
EXEC SQL DECLARE person TABLE
     (name       char(20),
      age        smallint,
      number     integer)
      END-EXEC.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
* Person information
    01 PERSONREC.
       02 PNAME        PIC X(20).
       02 P-AGE        PIC S99 USAGE COMP.
       02 PNUMBER      PIC S9(6) USAGE COMP.
    01 MAXID           PIC S9(6) USAGE COMP.
* Table field entry information
    01 STATE           PIC S9 USAGE COMP.
    01 RECNUM          PIC S9(4) USAGE COMP.
    01 LASTROW         PIC S9 USAGE COMP.
* Utility buffers
    01 MSGBUF          PIC X(200).
    01 RESPBUF         PIC X(20).
EXEC SQL END DECLARE SECTION END-EXEC.
* Table field row states:
* Empty or undefined row
 01 ST-UNDEF      PIC S9 USAGE COMP VALUE 0.
* Appended by user
 01 ST-NEW        PIC S9 USAGE COMP VALUE 1.
* Loaded by program - not updated
 01 ST-UNCHANGED  PIC S9 USAGE COMP VALUE 2.
* Loaded by program - since changed
 01 ST-CHANGE     PIC S9 USAGE COMP VALUE 3.
* Deleted by program
 01 ST-DELETE    PIC S9 USAGE COMP VALUE 4.
* SQLCA value for no rows
 01 NOT-FOUND    PIC S9(3) USAGE COMP VALUE 100.
* Update error from database
 01 UPDATE-ERROR  PIC S9(2) USAGE COMP.
* Transaction aborted
 01 XACT-ABORTED PIC S9 USAGE COMP.
PROCEDURE DIVISION.
BEGIN.
```

```
* Set up error handling for main program
      EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC.
      EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
      EXEC SQL WHENEVER SQLERROR STOP END-EXEC.
* Start Ingres and the Ingres/FORMS system
      EXEC SQL CONNECT personnel END-EXEC.
      EXEC FRS FORMS END-EXEC.
* Verify that the user can edit the "person" table
      EXEC FRS PROMPT NOECHO
           ('Password for table editor: ', :RESPBUF)
           END-EXEC.
      IF RESPBUF NOT = "MASTER_OF_ALL" THEN
          EXEC FRS
                MESSAGE 'No permission for task. Exiting . . .'
                END-EXEC
          EXEC FRS ENDFORMS END-EXEC
          EXEC SQL DISCONNECT END-EXEC
          STOP RUN.
* We assume no SQL errors can happen during screen updating
      EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
      EXEC FRS MESSAGE 'Initializing Person Form . . .' END-EXEC.
      EXEC FRS FORMINIT personfrm END-EXEC.
* Initialize "persontbl" table field with a data set in FILL
* mode, so that the runtime user can append rows. To keep track
* of events occuring to original rows loaded into the table
* field, hide the unique person number.
      EXEC FRS INITTABLE personfrm persontbl FILL
                  (number = integer)
           END-EXEC.
      CALL "LOAD-TABLE" GIVING MAXID.
      EXEC FRS DISPLAY personfrm UPDATE END-EXEC
      EXEC FRS INITIALIZE END-EXEC
      EXEC FRS ACTIVATE MENUITEM 'Top' END-EXEC
      EXEC FRS BEGIN END-EXEC
* Provide menu items, as well as the system FRS key, to scroll
* to both extremes of the table field.
            EXEC FRS SCROLL personfrm persontbl TO 1 END-EXEC.
      EXEC FRS END END-EXEC
      EXEC FRS ACTIVATE MENUITEM 'Bottom' END-EXEC
      EXEC FRS BEGIN END-EXEC
            EXEC FRS SCROLL personfrm persontbl TO END END-EXEC.
      EXEC FRS END END-EXEC
      EXEC FRS ACTIVATE MENUITEM 'Remove' END-EXEC
      EXEC FRS BEGIN END-EXEC

* Remove the person in the row the user's cursor is on. If there
* are no persons, exit operation with message. Note that this
* check cannot really happen, as there is always an UNDEFINED row
* in FILL mode.
            EXEC FRS INQUIRE_FRS table personfrm
                 (:LASTROW = LASTROW(persontbl)) END-EXEC.
            IF LASTROW = 0 THEN
                  EXEC FRS MESSAGE 'Nobody to Remove' END-EXEC
                  EXEC FRS SLEEP 2 END-EXEC
                  EXEC FRS RESUME FIELD persontbl END-EXEC.
            EXEC FRS DELETEROW personfrm persontbl END-EXEC.
      EXEC FRS END END-EXEC
      EXEC FRS ACTIVATE MENUITEM 'Find' END-EXEC
      EXEC FRS BEGIN END-EXEC
```

```
* Scroll user to the requested table field entry. Prompt the user
* for a name, and if one is typed in, loop through the data set
* searching for it.
           MOVE SPACES TO RESPBUF.
           EXEC FRS PROMPT ('Person''s name : ', :RESPBUF)
                       END-EXEC.
           IF RESPBUF = " " THEN
                 EXEC FRS RESUME FIELD persontbl END-EXEC.
           EXEC FRS UNLOADTABLE personfrm persontbl
                (:PNAME = name,
                 :RECNUM = _record,
                 :STATE = _state)
                 END-EXEC
           EXEC FRS BEGIN END-EXEC
* Compare name typed in with names in table, but do not compare
* with deleted rows.
                IF PNAME = RESPBUF AND STATE NOT = ST-DELETE
                THEN
                     EXEC FRS SCROLL personfrm persontbl
                           TO :RECNUM END-EXEC
                     EXEC FRS RESUME FIELD persontbl END-EXEC.
         EXEC FRS END END-EXEC.
* Fell out of loop without finding name. Inform user.
         STRING "Person """ RESPBUF
             """ not found in table [HIT RETURN] "
             DELIMITED BY SIZE INTO MSGBUF.
         EXEC FRS PROMPT NOECHO (:MSGBUF, :RESPBUF) END-EXEC.
   EXEC FRS END END-EXEC
   EXEC FRS ACTIVATE MENUITEM 'Exit' END-EXEC
   EXEC FRS BEGIN END-EXEC
         EXEC FRS VALIDATE FIELD persontbl END-EXEC.
         EXEC FRS BREAKDISPLAY END-EXEC.
   EXEC FRS END END-EXEC
   EXEC FRS FINALIZE END-EXEC.

* Exit person table editor and unload the table field. If any
* update, deletions or additions were made, duplicate these
* changes in the source table. If the user added new people,
* assign a unique person id to each person before adding the
* person to the table. To do this, increment the previously-saved
* maximum id number with each insert.
* Do all the updates in a transaction
     EXEC SQL COMMIT WORK END-EXEC.
* Hard code the error handling in the UNLOADTABLE loop, as we
* want to cleanly exit the loop.
     EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
     MOVE 0 TO UPDATE-ERROR.
     MOVE 0 TO XACT-ABORTED.
     EXEC FRS MESSAGE 'Exiting Person Application . . .'
                 END-EXEC.
     EXEC FRS UNLOADTABLE personfrm persontbl
         (:PNAME = name, :P-AGE = age,
          :PNUMBER = number, :STATE = _state)
          END-EXEC
     EXEC FRS BEGIN END-EXEC
* Row appended by user. Insert into "person" table with new
* unique id.
          IF STATE = ST-NEW THEN
             ADD 1 TO MAXID
              EXEC SQL INSERT INTO person (name, age, number)
                  VALUES (:PNAME, :P-AGE, :MAXID)
              END-EXEC
```

```
* Row updated by user. Reflect in table.
           ELSE IF STATE = ST-CHANGE THEN
               EXEC SQL UPDATE person SET
                    name = :PNAME, age = :P-AGE
                    WHERE number = :PNUMBER
                    END-EXEC
* Row deleted by user, so delete from table. Note that rows x
* unique by the user at runtime and then deleted are not saved
* and are therefore not unloaded.
           ELSE IF state = ST-DELETE THEN
                EXEC SQL DELETE FROM person
                     WHERE number = :PNUMBER END-EXEC
           END-IF.
* Else rows are UNDEFINED or UNCHANGED. No updates.

* Handle error conditions: if an error occurred, abort the
* transaction. If no rows were updated, inform user and prompt
* for continuation.
          IF SQLCODE < 0 THEN
                EXEC SQL INQUIRE_SQL(:MSGBUF = ERRORTEXT) END-EXEC
                EXEC SQL ROLLBACK WORK END-EXEC
                MOVE 1 TO UPDATE-ERROR
                MOVE 1 TO XACT-ABORTED
                EXEC FRS ENDLOOP END-EXEC
          ELSE IF SQLCODE = NOT-FOUND THEN
                STRING "Person """ PNAME
                    """ not updated. Abort all updates? "
                    DELIMITED BY SIZE INTO MSGBUF
                EXEC FRS PROMPT (:MSGBUF, :RESPBUF) END-EXEC
                IF RESPBUF = "Y" OR RESPBUF = "y" THEN
                      EXEC SQL ROLLBACK WORK END-EXEC
                      MOVE 1 TO XACT-ABORTED
                      EXEC FRS ENDLOOP END-EXEC
                END-IF
          END-IF.
      EXEC FRS END END-EXEC.
      IF XACT-ABORTED = 0 THEN
          EXEC SQL COMMIT END-EXEC.
      EXEC FRS ENDFORMS END-EXEC.
      EXEC SQL DISCONNECT END-EXEC.
      IF UPDATE-ERROR = 1 THEN
            DISPLAY "Your updates were aborted because of error:"
            DISPLAY msgbuf.
      STOP RUN.
END PROGRAM TABLE-EDIT.
IDENTIFICATION DIVISION.
PROGRAM-ID. LOAD-TABLE.
```

```
* This procedure opens a database cursor to load the table field
* with data from the "person" table. The columns "name" and "age"
* will be displayed, and "number" will be hidden. It returns the
* maximum employee number.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
EXEC SQL INCLUDE SQLCA END-EXEC.
* Person information -- declared to preprocessor in main program
01 PERSONREC.
    02 PNAME          PIC X(20).
    02 P-AGE          PIC S99 USAGE COMP.
    02 PNUMBER        PIC S9(6) USAGE COMP.
01  MAXID             PIC S9(6) USAGE COMP.
PROCEDURE DIVISION GIVING MAXID.
BEGIN.
      EXEC SQL DECLARE loadtab CURSOR FOR
         SELECT name, age, number
         FROM person
         END-EXEC.

* Set up error handling for loading procedure
    EXEC SQL WHENEVER SQLERROR GOTO LOAD-END END-EXEC.
    EXEC SQL WHENEVER NOT FOUND GOTO LOAD-END END-EXEC.
    EXEC FRS MESSAGE 'Loading Person Information . . .' END-EXEC.
* Fetch the maximum person id number for later use
    EXEC SQL SELECT MAX(number) INTO :MAXID FROM person END-EXEC.
    EXEC SQL OPEN loadtab END-EXEC.
    PERFORM UNTIL SQLCODE NOT = 0
* Fetch data into record and load table field
          EXEC SQL FETCH loadtab INTO :PERSONREC END-EXEC
          EXEC FRS LOADTABLE personfrm persontbl
             (name = :PNAME, age = :P-AGE, number = :PNUMBER)
              END-EXEC
      END-PERFORM.
LOAD-END.
      EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
      EXEC SQL CLOSE loadtab END-EXEC.
      EXIT PROGRAM.
      END PROGRAM LOAD-TABLE.
```

The Professor–Student Mixed Form Application

This application lets the user browse and update information about graduate students who report to a specific professor. The program is structured in a master/detail fashion, with the professor being the master entry, and the students the detail entries. The application uses two forms—one to contain general professor information and another for detailed student information.

The objects used in this application are shown in the following table:

Object	Description
personnel	The program's database environment.
professor	A database table with two columns: pname (char(25)) pdept (char(10)). See its declare table statement in the program for a full description.
student	A database table with seven columns: sname (char(25)) sage (integer1) sbdate (char(25)) sgpa (float4) dofmp (integer) scomment (varchar(200)) sadvisor (char(25)). See its declare table statement for a full description. The sadvisor column is the join field with the pname column in the Professor table.
masterfrm	The main form has the pname and pdept fields, which correspond to the information in the Professor table, and the studenttbl table field. The pdept field is display-only.
studenttbl	A table field in "masterfrm" with two columns, sname and sage. When initialized, it also has five hidden columns corresponding to information in the Student table.
studentfrm	The detail form, with seven fields, which correspond to information in the Student table. Only the sgpa, scomment, and sadvisor fields are updatable. All other fields are display-only.
grad	A structure whose members correspond in name and type to the columns of the Student database table, the studentfrm form and the studenttbl table field.

The program uses the masterfrm as the general-level master entry, in which data can only be retrieved and browsed, and the studentfrm as the detailed screen, in which specific student information can be updated.

The runtime user enters a name in the pname field and then selects the Students menu operation. The operation fills the studenttbl table field with detailed information of the students reporting to the named professor. This is done by the database cursor "studentcsr" in the LOAD-STUDENTS paragraph. The program assumes that each professor is associated with exactly one department. The user may then browse the table field (in read mode), which displays only the names and ages of the students. More information about a specific student may be requested by selecting the Zoom menu operation. This operation displays the form studentfrm (in update mode). The fields of studentfrm are filled with values stored in the hidden columns of studenttbl. The user may make changes to three fields (sgpa, scomment, and sadvisor). If validated, these changes will be written back to the database table (based on the unique student id), and to the table field's data set. This process can be repeated for different professor names.

Windows **UNIX**

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  STUDENT-ADMINISTRATOR.

       ENVIRONMENT DIVISION.

       DATA DIVISION.
       WORKING-STORAGE SECTION.

       EXEC SQL INCLUDE SQLCA END-EXEC.

*      Graduate student table
       EXEC SQL DECLARE student TABLE
            (sname      char(25),
             sage       integer1,
             sbdate     char(25),
             sgpa       float4,
             sidno      integer,
             scomment   varchar(200),
             sadvisor   char(25))
             END-EXEC.

*      Professor table
       EXEC SQL DECLARE professor TABLE
           (pname      char(25),
            pdept      char(10))
           END-EXEC.

       EXEC SQL BEGIN DECLARE SECTION END-EXEC.
*      Global grad student record maps to database table
       01 GRAD.
           02 SNAME        PIC X(25).
           02 SAGE         PIC S9(4) USAGE COMP.
           02 SBDATE       PIC X(25).
           02 SGPA         PIC S9(10)V9(8) USAGE COMP.
           02 SIDNO        PIC S9(9) USAGE COMP.
           02 SCOMMENT     PIC X(200).
           02 SADVISOR     PIC X(25).

*      Professor info maps to database table
       01 PROF.
           02 PNAME      PIC X(25).
           02 PDEPT      PIC X(10).
```

```
*       Row number of last row in student table field
        01 LASTROW        PIC S9(9) USAGE COMP.

*       Is user on a table field?
        01 ISTABLE          PIC S9 USAGE COMP.

*       Were changes made to data in "studentfrm"?
        01 CHANGED-DATA     PIC S9 USAGE COMP.

        Did user enter a valid advisor name?
        01 VALID-ADVISOR    PIC S9 USAGE COMP.

*       "Studentfrm" loaded?
        01 LOADFORM         PIC S9 USAGE COMP VALUE IS 0.

*       Local utility buffers
        01 MSGBUF           PIC X(200).
        01 RESPBUF          PIC X.
        01 OLD-ADVISOR      PIC X(25).

*       Note: Compiled forms are not yet accepted as
*       EXTERNAL due to restrictions noted in the chapter
*       that describes how to link the RTS with compiled
*       forms.  Consequently, declarations of external
*       form objects and the corresponding ADDFORM
*       statement have been commented out and replaced by
*       a CALL "add_formname" statement.
*       01    masterfrm  PIC S9(9) USAGE COMP-5 IS EXTERNAL.
*       01    studentfrm PIC S9(9) USAGE COMP-5 IS EXTERNAL.

        EXEC SQL END DECLARE SECTION END-EXEC.
**
*       Procedure Division: STUDENT-ADMINISTRATOR
*
*       Start up program, Ingres and the FORMS system and
*       call Master driver.
**
        PROCEDURE DIVISION.
        EXAMPLE SECTION.
        XBEGIN.

        EXEC FRS FORMS END-EXEC.
        EXEC SQL WHENEVER SQLERROR STOP END-EXEC.
        EXEC FRS MESSAGE 'Initializing Student
                           Administrator .  .' END-EXEC.

        EXEC SQL CONNECT personnel END-EXEC.

        PERFORM MASTER THRU END-MASTER.

        EXEC FRS CLEAR SCREEN END-EXEC.
        EXEC FRS ENDFORMS END-EXEC.
        EXEC SQL DISCONNECT END-EXEC.
        STOP RUN.
**
*       Paragraph: MASTER
*
*       Drive the application, by running "masterfrm", and
*       allowing the user to "zoom" into a selected student.
**
        MASTER.
```

```
*      EXEC FRS ADDFORM :masterfrm END-EXEC.
       CALL "add_masterfrm".

*      Initialize "studenttbl" with a data set in READ mode.
*      Declare hidden columns for all the extra fields that the
*      program will display when more information is requested
*      about a student.  Columns "sname" and "sage" are displayed,
*      all other columns are hidden, the student information
*      form.

       EXEC FRS INITTABLE masterfrm studenttbl READ
               (sbdate   = char(25),
                sgpa     = float4,
                sidno    = integer,
                scomment = char(200),
                sadvisor = char(20))
                END-EXEC.
       EXEC FRS DISPLAY masterfrm UPDATE END-EXEC
       EXEC FRS INITIALIZE END-EXEC

       EXEC FRS BEGIN END-EXEC
              EXEC FRS MESSAGE
                         'Enter an Advisor name . . .' END-EXEC.
              EXEC FRS SLEEP 2 END-EXEC.
       EXEC FRS END END-EXEC

       EXEC FRS ACTIVATE MENUITEM
              'Students', FIELD 'pname' END-EXEC
       EXEC FRS BEGIN END-EXEC

*            Load the students of the specified professor
             EXEC FRS GETFORM (:PNAME = pname) END-EXEC

*            If no professor name is given, resume
             IF PNAME = SPACES THEN
                        EXEC FRS RESUME FIELD pname END-EXEC.

*            Verify the professor exists.  Local error handling
*            just prints the message, and continues.  We assume
*            that each professor has exactly one department.

             EXEC SQL WHENEVER SQLERROR CALL SQLPRINT END-EXEC.
             EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
             MOVE SPACES TO PDEPT.
             EXEC SQL SELECT pdept
                      INTO :PDEPT
                      FROM professor
                      WHERE pname = :PNAME
                      END-EXEC.

       IF PDEPT = SPACES THEN
                      STRING "No professor with name """, PNAME,
                            """ [RETURN]" DELIMITED BY SIZE
                            INTO MSGBUF
                       EXEC FRS PROMPT NOECHO (:MSGBUF, :RESPBUF)
                             END-EXEC
                       EXEC FRS CLEAR FIELD ALL END-EXEC
                       EXEC FRS RESUME FIELD pname END-EXEC.

*              Fill the department field and load students
               EXEC FRS PUTFORM (pdept = :PDEPT) END-EXEC.

*              Refresh for query
               EXEC FRS REDISPLAY END-EXEC.

               PERFORM LOAD-STUDENTS THRU END-LOAD.
```

```
                EXEC FRS RESUME FIELD studenttbl END-EXEC.

        EXEC FRS END END-EXEC

        EXEC FRS ACTIVATE MENUITEM 'Zoom' END-EXEC

        EXEC FRS BEGIN END-EXEC

*            Confirm that user is on "studenttbl", and that the
*            table field is not empty.  Collect data from the row
*            and zoom for browsing and updating.
             EXEC FRS INQUIRE_FRS field
                             masterfrm (:ISTABLE = table)
                   END-EXEC.

             IF ISTABLE = 0 THEN
                   EXEC FRS PROMPT NOECHO
                          ('Select from the student
                                         table [RETURN]',
                           :RESPBUF) END-EXEC

             EXEC FRS RESUME FIELD studenttbl END-EXEC.
             EXEC FRS INQUIRE_FRS table masterfrm
                      (:LASTROW = lastrow) END-EXEC.

             IF LASTROW = 0 THEN
                     EXEC FRS PROMPT NOECHO
                              ('There are no students [RETURN]',
                                :RESPBUF) END-EXEC

                     EXEC FRS RESUME FIELD pname END-EXEC.

*            Collect all data on student into global record
             EXEC FRS GETROW masterfrm studenttbl
                           (:SNAME   = sname,
                            :SAGE    = sage,
                            :SBDATE  = sbdate,
                            :SGPA    = sgpa,
                            :SIDNO   = sidno,
                            :SCOMMENT = scomment,
                            :SADVISOR = sadvisor)
                            END-EXEC.

*           Display "studentfrm", and if any changes were made
*           make the updates to the local table field row.  Only
*           updates to the columns corresponding to writable fields
*           in "studentfrm".  If the student changed advisors, then
*           delete this row from the display.

            MOVE SADVISOR TO OLD-ADVISOR.
            PERFORM STUDENT-INFO-CHANGED THRU END-STUDENT.

            IF CHANGED-DATA = 1 THEN
                 IF OLD-ADVISOR NOT = SADVISOR THEN
                            EXEC FRS DELETEROW masterfrm studenttbl
                                 END-EXEC
                 ELSE
                            EXEC FRS PUTROW masterfrm studenttbl
                                  (sgpa = :SGPA,
                                   scomment = :SCOMMENT,
                                   sadvisor = :SADVISOR)
                                  END-EXEC
                 END-IF
        END-IF.

        EXEC FRS END END-EXEC
```

```
        EXEC FRS ACTIVATE MENUITEM 'Exit' END-EXEC
        EXEC FRS BEGIN END-EXEC
               EXEC FRS BREAKDISPLAY END-EXEC.
        EXEC FRS END END-EXEC

        EXEC FRS FINALIZE END-EXEC

        END-MASTER.
              EXIT.

 **
 *      Paragraph: LOAD-STUDENTS
 *
 *      For the current professor name, this paragraph loads into
 *      the "studenttbl" table field all the students whose
 *      advisor is the professor with that name.
 **
        LOAD-STUDENTS.

        EXEC SQL DECLARE studentcsr CURSOR FOR
                 SELECT sname, sage, sbdate, sgpa,
                        sidno, scomment, sadvisor
                 FROM student
                 WHERE sadvisor = :PNAME
                 END-EXEC.

 *      Clear previous contents of table field.  Load the table
 *      field from the database table based on the advisor name.
 *      Columns "sname" and "sage" will be displayed, and all
 *      others will be hidden.

        EXEC FRS MESSAGE 'Retrieving Student Information . . .'
                  END-EXEC.
        EXEC FRS CLEAR FIELD studenttbl END-EXEC.

        EXEC SQL WHENEVER SQLERROR GOTO END-LOAD END-EXEC.
        EXEC SQL WHENEVER NOT FOUND GOTO END-LOAD END-EXEC.

        EXEC SQL OPEN studentcsr END-EXEC.

 *      Before we start the loop, we know that the OPEN was
 *      successful and that NOT FOUND was not set.

        PERFORM UNTIL SQLCODE NOT = 0
               EXEC SQL FETCH studentcsr INTO :GRAD END-EXEC
               EXEC FRS LOADTABLE masterfrm studenttbl
                            (sname    = :SNAME,
                             sage     = :SAGE,
                             sbdate   = :SBDATE,
                             sgpa     = :SGPA,
                             sidno    = :SIDNO,
                             scomment = :SCOMMENT,
                             sadvisor = :SADVISOR)
                             END-EXEC
        END-PERFORM.
        END-LOAD.
 *      Clean up on an error, and close cursors
        EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
        EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
        EXEC SQL CLOSE studentcsr END-EXEC.
 **
 *     Paragraph: STUDENT-INFO-CHANGED
 *
 *     Allow the user to zoom into the details of a selected
 *     student.  Some of the data can be updated by the user.
```

```
*     If any updates were made, then reflect these back into
*     the database table.  The paragraph records whether or not
*     changes were made via the CHANGED-DATA variable.
**
       STUDENT-INFO-CHANGED.

*      Control ADDFORM to only initialize once
       IF LOADFORM = 0 THEN
           EXEC FRS MESSAGE 'Loading Student form . . .' END-EXEC
           EXEC FRS ADDFORM :studentfrm END-EXEC
           CALL "add_studentfrm"
           MOVE 1 TO LOADFORM.

*      Local error handle just prints error and continues
       EXEC SQL WHENEVER SQLERROR CALL SQLPRINT END-EXEC.
       EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
       EXEC FRS DISPLAY studentfrm FILL END-EXEC
       EXEC FRS INITIALIZE
               (sname    = :SNAME,
                sage     = :SAGE,
                sbdate   = :SBDATE,
                sgpa     = :SGPA,
                sidno    = :SIDNO,
                scomment = :SCOMMENT,
                sadvisor = :SADVISOR)
                END-EXEC

        EXEC FRS ACTIVATE MENUITEM 'Write' END-EXEC
        EXEC FRS BEGIN END-EXEC

*            If changes were made, update the database table.
*            Only bother with the fields that are not read-only.

             EXEC FRS INQUIRE_FRS form (:CHANGED-DATA = change)
                    END-EXEC.

             IF CHANGED-DATA = 0 THEN
                    EXEC FRS BREAKDISPLAY END-EXEC.

             EXEC FRS VALIDATE END-EXEC.
             EXEC FRS MESSAGE
                  'Writing changes to database. . .' END-EXEC.
             EXEC FRS GETFORM
                     (:SGPA = sgpa,
                      :SCOMMENT = scomment,
                      :SADVISOR = sadvisor)
                     END-EXEC.

*            Enforce integrity of professor name.
             MOVE 0 TO VALID-ADVISOR.
             EXEC SQL SELECT 1 INTO :VALID-ADVISOR
                      FROM professor
                      WHERE pname = :SADVISOR
                      END-EXEC.
             IF VALID-ADVISOR = 0 THEN
                    EXEC FRS MESSAGE
                        'Not a valid advisor name'
                        END-EXEC
                    EXEC FRS SLEEP 2 END-EXEC
                    EXEC FRS RESUME FIELD sadvisor END-EXEC
             ELSE
                    EXEC SQL UPDATE student SET
                          sgpa     = :SGPA,
                          scomment = :SCOMMENT,
                          sadvisor = :SADVISOR
                          WHERE sidno = :SIDNO
```

```
                         END-EXEC
                   EXEC FRS BREAKDISPLAY END-EXEC
            END-IF.

        EXEC FRS END END-EXEC
        EXEC FRS ACTIVATE MENUITEM 'Quit' END-EXEC

        EXEC FRS BEGIN END-EXEC
*          Quit without submitting changes
           MOVE 0 TO CHANGED-DATA.
           EXEC FRS BREAKDISPLAY END-EXEC.

       EXEC FRS END END-EXEC

       EXEC FRS FINALIZE END-EXEC

       END-STUDENT.
             EXIT.
```

VMS

```
IDENTIFICATION DIVISION.
PROGRAM-ID. STUDENT-ADMINISTRATOR.

ENVIRONMENT DIVISION.

DATA DIVISION.
WORKING-STORAGE SECTION.

EXEC SQL INCLUDE SQLCA END-EXEC.

Graduate student table
EXEC SQL DECLARE student TABLE
    (sname     char(25),
     sage      integer1,
     sbdate    char(25),
     sgpa      float4,
     sidno     integer,
     scomment  □archars(200),
     sadvisor  char(25))
     END-EXEC.

Professor table
EXEC SQL DECLARE professor TABLE
    (pname    char(25),
     pdept    char(10))
     END-EXEC.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.

Global grad student record maps to database table
 GRAD.
     02 SNAME      PIC X(25).
     02 SAGE       PIC S9(4) USAGE COMP.
     02 SBDATE     PIC X(25).
     02 SGPA       USAGE COMP-1.
     02 SIDNO      PIC S9(9) USAGE COMP.
     02 SCOMMENT   PIC X(200).
     02 SADVISOR   PIC X(25).

Professor info maps to database table
 PROF.
     02 PNAME   PIC X(25).
     02 PDEPT   PIC X(10).

Row number of last row in student table field
```

```
 01  LASTROW     PIC S9(9) USAGE COMP.

Is user on a table field?
 01  ISTABLE   PIC S9 USAGE COMP.

Were changes made to data in “studentfrm”?
 01  CHANGED    PIC S9 USAGE COMP.

Did user enter a valid advisor name?
 01 VALID-ADVISOR PIC S9 USAGE COMP.
 02
“Studentfrm” loaded?
 01  LOADFORM    PIC S9 USAGE COMP VALUE IS 0.

Local utility buffers
 01 MSGBUF        PIC X(200).
 01 RESPBUF       PIC X.
 01 OLD-ADVISOR   PIC X(25).

Externally compiled forms
 01 MASTERF   PIC S9(9) USAGE COMP VALUE EXTERNAL Masterfrm.
 01 STUDENTF  PIC S9(9) USAGE COMP VALUE EXTERNAL Studentfrm.
EXEC SQL END DECLARE SECTION END-EXEC.

PROCEDURE DIVISION.
BEGIN.

Start program and call Master driver. First, start Ingres and
the FORMS system.

      EXEC FRS FORMS END-EXEC.

      EXEC SQL WHENEVER SQLERROR STOP END-EXEC.
      EXEC FRS MESSAGE ‘Initializing Student Administrator . . .’
                         END-EXEC.

      EXEC SQL CONNECT personnel END-EXEC.

      PERFORM MASTER THRU END-MASTER.

      EXEC FRS CLEAR SCREEN END-EXEC.
      EXEC FRS ENDFORMS END-EXEC.
      EXEC SQL DISCONNECT END-EXEC.
      STOP RUN.

MASTER.

This paragraph drives the application. It runs “masterfrm” and
allows the user to “zoom” in on a selected student.

      EXEC FRS ADDFORM :MASTERF END-EXEC.
Initialize “studenttbl” with a data set in READ mode. Declare
hidden columns for all the extra fields that the program will
display when more information is requested about a student.
Columns “sname” ad “sage” are displayed. All other columns are
hidden, to be used in the student information form.

      EXEC FRS INITTABLE masterfrm studenttbl READ
          (sbdate   = char(25),
           sgpa     = float4,
           sidno    = integer,
           scomment = char(200),
           sadvisor = char(20))
           END-EXEC.

      EXEC FRS DISPLAY masterfrm UPDATE END-EXEC
```

```
        EXEC FRS INITIALIZE END-EXEC
        EXEC FRS BEGIN END-EXEC

              EXEC FRS MESSAGE 'Enter an Advisor name . . .'
                          END-EXEC.
              EXEC FRS SLEEP 2 END-EXEC.

        EXEC FRS END END-EXEC

        EXEC FRS ACTIVATE MENUITEM 'Students', FIELD 'pname'
                          END-EXEC
        EXEC FRS BEGIN END-EXEC

*      Load the students of the specified professor
              EXEC FRS GETFORM (:PNAME = pname) END-EXEC.

*      If no professor name is given, resume
              IF PNAME = " " THEN
                  EXEC FRS RESUME FIELD pname END-EXEC.

*     Verify that the professor exists. Local error handling just
*     prints the message and continues. Assume that each professor
*     has exactly one department.

              EXEC SQL WHENEVER SQLERROR CALL SQLPRINT END-EXEC.
              EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
              MOVE SPACES TO PDEPT.
              EXEC SQL SELECT pdept
                  INTO :PDEPT
                  FROM professor
                  WHERE pname = :PNAME
                  END-EXEC.

              IF PDEPT = " " THEN
                  STRING "No professor with name """ PNAME
                        """ [RETURN]" DELIMITED BY SIZE INTO MSGBUF
                  EXEC FRS PROMPT NOECHO (:MSGBUF, :RESPBUF)
                          END-EXEC
                  EXEC FRS CLEAR FIELD ALL END-EXEC
                  EXEC FRS RESUME FIELD pname END-EXEC.

*     Fill the department field and load students
*
              EXEC FRS PUTFORM (pdept = :PDEPT) END-EXEC.

* Refresh for query
              EXEC FRS REDISPLAY END-EXEC.
              PERFORM LOAD-STUDENTS THRU END-LOAD.

              EXEC FRS RESUME FIELD studenttbl END-EXEC.

        EXEC FRS END END-EXEC

        EXEC FRS ACTIVATE MENUITEM 'Zoom' END-EXEC
        EXEC FRS BEGIN END-EXEC
```

```
* Confirm that user is in "studenttbl" and that the table field
* is not empty. Collect data from the row and zoom for browsing
* and updating.

            EXEC FRS INQUIRE_FRS field masterfrm
                        (:ISTABLE = table)
                 END-EXEC.

            IF ISTABLE = 0 THEN
                  EXEC FRS PROMPT NOECHO
                        ('Select from the student table [RETURN]',
                         :RESPBUF) END-EXEC
                  EXEC FRS RESUME FIELD studenttbl END-EXEC.

            EXEC FRS INQUIRE_FRS table masterfrm
                 (:LASTROW = lastrow) END-EXEC.
            IF LASTROW = 0 THEN
                  EXEC FRS PROMPT NOECHO
                       ('There are no students [RETURN]',
                       :RESPBUF) END-EXEC
                  EXEC FRS RESUME FIELD pname END-EXEC.

* Collect all data on student into global record

            EXEC FRS GETROW masterfrm studenttbl
                (:SNAME    = sname,
                 :SAGE     = sage,
                 :SBDATE   = sbdate,
                 :SGPA     = sgpa,
                 :SIDNO    = sidno,
                 :SCOMMENT = scomment,
                 :SADVISOR = sadvisor)
                 END-EXEC.

* Display "studentfrm," and if any changes were made, make the
* update to the local table field row. Only make updates to the
* columns corresponding to writable fields in "studentfrm." If
* the student changed advisors delete this row from the display.

            MOVE SADVISOR TO OLD-ADVISOR.
            PERFORM STUDENT-INFO-CHANGED THRU END-STUDENT.

            IF CHANGED = 1 THEN
                  IF OLD-ADVISOR NOT = SADVISOR THEN
                        EXEC FRS DELETEROW masterfrm studenttbl
                             END-EXEC
                  ELSE
                        EXEC FRS PUTROW masterfrm studenttbl
                            (sgpa     = :SGPA,
                             scomment = :SCOMMENT,
                             sadvisor = :SADVISOR)
                            END-EXEC
                  END-IF
            END-IF.

      EXEC FRS END END-EXEC
      EXEC FRS ACTIVATE MENUITEM 'Exit' END-EXEC
      EXEC FRS BEGIN END-EXEC
            EXEC FRS BREAKDISPLAY END-EXEC.
      EXEC FRS END END-EXEC

      EXEC FRS FINALIZE END-EXEC


END-MASTER.
```

```
LOAD-STUDENTS.

* For the current professor name, this paragraph loads into the
* “studenttbl” table field all the students whose advisor is the
* professor with that name.

      EXEC SQL DECLARE studentcsr CURSOR FOR
          SELECT sname, sage, sbdate, sgpa,
                 sidno, scomment, sadvisor
          FROM student
          WHERE sadvisor = :PNAME
          END-EXEC.

* Clear previous contents of table field. Load the table field
* from the database table based on the advisor name. Columns
* "sname" and “sage” will be displayed, and all others will be
* hidden.

      EXEC FRS MESSAGE 'Retrieving Student Information . . '
                  END-EXEC.

      EXEC FRS CLEAR FIELD studenttbl END-EXEC.

      EXEC SQL WHENEVER SQLERROR GOTO END-LOAD END-EXEC.
      EXEC SQL WHENEVER NOT FOUND GOTO END-LOAD END-EXEC.

      EXEC SQL OPEN studentcsr END-EXEC.

* Before we start the loop, we know that the OPEN was
* successful and that NOT FOUND was not set.

      PERFORM UNTIL SQLCODE NOT = 0
            EXEC SQL FETCH studentcsr INTO :GRAD END-EXEC

            EXEC FRS LOADTABLE masterfrm studenttbl
                 (sname    = :SNAME,
                  sage     = :SAGE,
                  sbdate   = :SBDATE,
                  sgpa     = :SGPA,
                  sidno    = :SIDNO,
                  scomment = :SCOMMENT,
                  sadvisor = :SADVISOR)
                  END-EXEC
      END-PERFORM.

END-LOAD.

* Clean up on an error, and close cursors
      EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
      EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
      EXEC SQL CLOSE studentcsr END-EXEC.

STUDENT-INFO-CHANGED.

* This paragraph allows the user to zoom in on the details of a
* selected student. Some of the data can be updated by the
* user. If any updates were made, they are reflected back into
* the database table. The paragraph records whether or not
* changes were made via the CHANGED variable.

* Control ADDFORM to only initialize once

      IF LOADFORM = 0 THEN
           EXEC FRS MESSAGE 'Loading Student form . . .' END-EXEC
           EXEC FRS ADDFORM :STUDENTF END-EXEC
           MOVE 1 TO LOADFORM.
```

```
* Local error handle just prints error and continues
      EXEC SQL WHENEVER SQLERROR CALL SQLPRINT END-EXEC.
      EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.

      EXEC FRS DISPLAY studentfrm FILL END-EXEC
      EXEC FRS INITIALIZE
           (sname    = :SNAME,
            sage     = :SAGE,
            sbdate   = :SBDATE,
            sgpa     = :SGPA,
            sidno    = :SIDNO,
            scomment = :SCOMMENT,
            sadvisor = :SADVISOR)
            END-EXEC
      EXEC FRS ACTIVATE MENUITEM 'Write' END-EXEC
      EXEC FRS BEGIN END-EXEC

* If changes were made, update the database table. Only bother
* with the fields that are not read-only.

            EXEC FRS INQUIRE_FRS form (:CHANGED = change) END-EXEC.

            IF CHANGED = 0 THEN
                   EXEC FRS BREAKDISPLAY END-EXEC.

            EXEC FRS VALIDATE END-EXEC.

            EXEC FRS MESSAGE
                   'Writing changes to database. . .'
                   END-EXEC.

            EXEC FRS GETFORM
                  (:SGPA     = sgpa,
                   :SCOMMENT = scomment,
                   :SADVISOR = sadvisor)
                   END-EXEC.

* Enforce integrity of professor name.

            MOVE 0 TO VALID-ADVISOR.
            EXEC SQL SELECT 1 INTO :VALID-ADVISOR
                  FROM professor
                  WHERE pname = :SADVISOR
                  END-EXEC.
            IF VALID-ADVISOR = 0 THEN
                  EXEC FRS MESSAGE 'Not a valid advisor name'
                        END-EXEC
                  EXEC FRS SLEEP 2 END-EXEC
                  EXEC FRS RESUME FIELD sadvisor END-EXEC
            ELSE
                  EXEC SQL UPDATE student SET
                       sgpa        = :SGPA,
                       scomment    = :SCOMMENT,
                       sadvisor    = :SADVISOR
                       WHERE sidno = :SIDNO
                       END-EXEC
                  EXEC FRS BREAKDISPLAY END-EXEC
            END-IF.

      EXEC FRS END END-EXEC

      EXEC FRS ACTIVATE MENUITEM 'Quit' END-EXEC
      EXEC FRS BEGIN END-EXEC

* Quit without submitting changes
```

```
              MOVE 0 TO CHANGED.
              EXEC FRS BREAKDISPLAY END-EXEC.

        EXEC FRS END END-EXEC

        EXEC FRS FINALIZE END-EXEC

END-STUDENT.
        EXIT.
```

The SQL Terminal Monitor Application

This application executes SQL statements that are read in from the terminal. The application reads statements from input and writes results to output. Dynamic SQL is used to process and execute the statements.

When the application starts, the user is prompted for the database name. The user is then prompted for an SQL statement. SQL comments and statement delimiters are not accepted. The SQL statement is processed using Dynamic SQL and results and SQL errors are written to output. At the end of the results, an indicator of the number of rows affected is displayed. The loop is then continued and the user is prompted for another SQL statement. When end-of-file is typed in the application rolls back any pending updates and disconnects from the database.

The user's SQL statement is prepared using prepare and describe. If the SQL statement is not a select statement, then it is run using execute and the number of rows affected is printed. If the SQL statement is a select statement, a Dynamic SQL cursor is opened, and all the rows are fetched and printed. The sections of code that print the results do not try to tabulate the results. A row of column names is printed, followed by each row of the results.

Keyboard interrupts are not handled. Fatal errors, such as allocation errors, and boundary condition violations are handled by rolling back pending updates and disconnecting from the database session.

Windows **UNIX**

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  SQL-MONITOR.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.

*      Include SQL Communications and Descriptor Areas
       EXEC SQL INCLUDE SQLCA END-EXEC.
       EXEC SQL INCLUDE SQLDA END-EXEC.
*      Dynamic SQL statement name (documentary only)
       EXEC SQL DECLARE stmt STATEMENT END-EXEC.
*      Cursor declaration for dynamic statement
       EXEC SQL DECLARE csr CURSOR FOR stmt END-EXEC.
       EXEC SQL BEGIN DECLARE SECTION END-EXEC.
*      Database name
       01 DB-NAME         PIC X(30).
*      Dynamic SQL statement buffer
       01 STMT-BUF        PIC X(1000).
```

```
*      SQL error message buffer
       01 ERROR-BUF       PIC X(1024).
       EXEC SQL END DECLARE SECTION END-EXEC.
*      SQL statement number
       01 STMT-NUM        PIC 999.
*      Reading state
       01 READING-STMT    PIC S9(4) USAGE COMP.
          88 DONE-READING    VALUE 0.
          88 STILL-READING  VALUE 1.
*      Number of rows affected by last SQL statement
       01 STMT-ROWS           PIC ZZZZZ9.
*      Number of rows retrieved by last SELECT statement
       01 SELECT-ROWS         PIC S9(8) USAGE COMP.
*      Dynamic SELECT statement set up state
       01 SELECT-SETUP        PIC S9(4) USAGE COMP.
           88 SETUP-FAIL      VALUE 0.
           88 SETUP-OK        VALUE 1.
*      Index into SQLVAR table
       01 COLN                PIC 999.
*      Base data type of SQLVAR item without nullability
       01 BASE-TYPE            PIC S9(4) USAGE COMP.
*      Is a result column type nullable
       01 IS-NULLABLE          PIC S9(4) USAGE COMP.
           88 NOT-NULLABLE    VALUE 0.
           88 NULLABLE        VALUE 1.


*      Global result data storage.  This pool of data
*      includes the maximum number of result data
*      items needed to execute a Dynamic SELECT
*      statement.  There is a table of 1024 integers,
*      decimal and null indicator data items, and a
*      large character string buffer.
*      The display data picture formats may be
*      modified if more numeric precision is
*      required.  Note: floating-point and
*      money types are stored in decimal variables.
       01 RESULT-DATA.
            02 NUMERIC-DATA OCCURS IISQ-MAX-COLS TIMES.
                03 INT-DATA         PIC S9(9) USAGE COMP-5 SYNC.
                03 IND-DATA         PIC S9(4) USAGE COMP-5 SYNC.
            02 DECIMAL-DATA         OCCURS IISQ-MAX-COLS TIMES.
                03 DEC-DATA         PIC S9(10)V9(8) USAGE COMP-3.
            02 STRING-DATA.
                03 CHAR-LEN         PIC S9(4) USAGE COMP.
                03 CHAR-DATA        PIC X(2500).
            02 DISPLAY-DATA.
                03 DISP-INT         PIC +Z(6)99.
                03 DISP-DEC         PIC +Z(8)99.99(8).
*      Current lengths of local character data.
       01 CUR-LEN                   PIC S9(4) USAGE COMP.
**
* Procedure Division: SQL-MONITOR
*
*      Main entry of SQL Monitor application.  Prompt for
*      database name and connect to the database.  Run
*      the monitor and disconnect from the database.
*      Before disconnecting, roll back any pending updates.
**
       PROCEDURE DIVISION.
       EXAMPLE SECTION.
       XBEGIN.
```

```
*      Execute a dummy ACCEPT statement from the CONSOLE prior
*      to using the ACCEPT statement to read in input.  This
*      introductory ACCEPT statement (which is documented to
*      read from COMMAND-LINE)may not be necessary on all systems.
       ACCEPT DB-NAME FROM CONSOLE.
*     Prompt for database name.
      MOVE SPACES TO DB-NAME.
      DISPLAY "SQL Database: " WITH NO ADVANCING.
      ACCEPT DB-NAME FROM CONSOLE.
      IF (DB-NAME = SPACES) THEN
           DISPLAY "**************************"
           STOP RUN.
      DISPLAY " -- SQL Terminal Monitor -- ".
*     Treat connection errors as fatal.
      EXEC SQL WHENEVER SQLERROR STOP END-EXEC.
      EXEC SQL CONNECT :DB-NAME END-EXEC.
*     Run the Terminal Monitor
      PERFORM RUN-MONITOR.
      EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
      DISPLAY "SQL: Exiting monitor program.".
      EXEC SQL ROLLBACK END-EXEC.
      EXEC SQL DISCONNECT END-EXEC.
      STOP RUN.
**
* Paragraph: RUN-MONITOR
*
*     Run the SQL monitor. Initialize the global
*     SQLDA with the number of SQLVAR elements. Loop
*     while prompting the user for input; if
*     end-of-file is detected then return to the
*     calling paragraph (the main program). If the
*     user inputs a statement, execute it (using
*     paragraph EXECUTE-STATEMENT).
**
       RUN-MONITOR.
*      Initialize the SQLN (the number of SQLVAR
*      elements is set by default to IISQ-MAX-COLS)
*      Now we are setup for input. Initialize
*      statement number and reading state.
       MOVE 0 TO STMT-NUM.
       SET STILL-READING TO TRUE.
*      Loop while prompting, reading and processing
*      the SQL statement.
       PERFORM UNTIL DONE-READING
            ADD 1 TO STMT-NUM
            PERFORM READ-STATEMENT
            IF (STILL-READING) THEN
                  PERFORM EXECUTE-STATEMENT THRU END-EXECUTE
            END-IF
      END-PERFORM.
**
* Paragraph: EXECUTE-STATEMENT
*
*     Using the PREPARE and DESCRIBE facilities determine if
*     the input statement is a SELECT statement or not. If
*     the statement is not a SELECT statement then EXECUTE it,
*     otherwise open a cursor and
*     process a dynamic SELECT statement (using paragraph
*     EXECUTE-SELECT). After processing the statement, print
*     the number of rows affected by the statement and any SQL
*     errors.
**
       EXECUTE-STATEMENT.
       EXEC SQL WHENEVER SQLERROR GO TO END-EXECUTE END-EXEC.
```

```
*      PREPARE and DESCRIBE the statement. Inspect the
*      contents of the SQLDA and determine if it is a SELECT
*      statement or not.
       EXEC SQL PREPARE stmt FROM :STMT-BUF END-EXEC.
       EXEC SQL DESCRIBE stmt INTO :SQLDA END-EXEC.
*      IF SQLD = 0 then this is not a SELECT.
       IF (SQLD = 0) THEN
               EXEC SQL EXECUTE stmt END-EXEC
               MOVE SQLERRD(3) TO STMT-ROWS

*      Otherwise this is a SELECT. Verify that there are enough
*      SQLVAR result variables. If there are too few print an
*      error and continue, otherwise call EXECUTE-SELECT.
       ELSE IF (SQLD > SQLN) THEN
                 DISPLAY "SQL Error: SQLDA requires more than "
                     "1024 result variables."
                 MOVE 0 TO STMT-ROWS
       ELSE
                 PERFORM EXECUTE-SELECT THRU END-SELECT
                 MOVE SELECT-ROWS TO STMT-ROWS
       END-IF.
*      Print the number of rows processed.
       DISPLAY "[" STMT-ROWS " row(s)]".
*      Only print the error message if we arrived at this label
*      because of an SQL error.
       END-EXECUTE.
            EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
            IF (SQLCODE < 0) THEN
                  PERFORM PRINT-ERROR.
**
* Paragraph: EXECUTE-SELECT
*      Execute a Dynamic SELECT statement. The SQLDA has already
*      been described, so print the table header column names,
*      open a dynamic cursor, and retrieve and print the results.
*      Accumulate the number of rows processed in SELECT-ROWS.
**
       EXECUTE-SELECT.
*      So far no rows.
       MOVE 0 TO SELECT-ROWS.
*      Set up the result types and data items, and print result
*      column names. SETUP-ROW will set SETUP-FAIL/OK if it
*      fails/succeeds.
       PERFORM SETUP-ROW.
       IF (SETUP-FAIL) THEN
             GO TO END-SELECT.
       EXEC SQL WHENEVER SQLERROR GO TO SELECT-ERR END-EXEC.
*      Open the dynamic cursor.
       EXEC SQL OPEN csr FOR READONLY END-EXEC.
*      Fetch and print each row. Accumulate the number of
*      rows fetched.
       PERFORM UNTIL SQLCODE NOT = 0
               EXEC SQL FETCH csr USING DESCRIPTOR :SQLDA END-EXEC
                 IF (SQLCODE = 0) THEN
                         ADD 1 TO SELECT-ROWS
                         PERFORM PRINT-ROW
                 END-IF
       END-PERFORM.
       SELECT-ERR.
            EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
```

```
*       Only print the error message if we arrived at this
*       label because of an SQL error.
        IF (SQLCODE < 0) THEN
             PERFORM PRINT-ERROR.
        EXEC SQL CLOSE csr END-EXEC.
 END-SELECT.
        EXIT.
**
* Paragraph: SETUP-ROW
*
*     A statement has just been described, so set up the
*     SQLDA for result processing. Print all the column
*     names and allocate result data items for retrieving
*     data using paragraph SETUP-COLUMN.
*     This paragraph sets SETUP-OK if it succeeds, and
*     SETUP-FAIL if there was some sort of initialization
*     error(in SETUP-COLUMN).
**

       SETUP-ROW.

*      Initialize column setup. No character data used yet.

       SET SETUP-OK TO TRUE.
       MOVE 1 TO CHAR-LEN.

*      Process each column.

       PERFORM SETUP-COLUMN
               VARYING COLN FROM 1 BY 1
               UNTIL (COLN > SQLD) OR (SETUP-FAIL).

*      At this point we've processed all columns for
*      data type information.
*      End the line of column names.

       DISPLAY SPACE.
       DISPLAY "-----------------------------".
**
* Paragraph: SETUP-COLUMN
*
*     When setting up for a SELECT statement column names are
*     printed, and result data items (for retrieving data)
*     are chosen out of a pool of variables (integers,
*     decimals, a large character string space and null
*     indicators). The SQLDATA and SQLIND fields are pointed
*     at the addresses of the result data items and
*     indicators. Paragraph sets SETUP-FAIL if it fails.
**

       SETUP-COLUMN.


*      For each column print the number and name of the column,
*      e.g.: [001] sal [002] name [003] age
       DISPLAY "[" COLN "] " WITH NO ADVANCING.
       DISPLAY SQLNAMEC(COLN)(1:SQLNAMEL(COLN)) WITH NO ADVANCING.
       IF (COLN < SQLD) THEN
             DISPLAY SPACE WITH NO ADVANCING.
```

```
*     Determine the data type of the column and to where SQLDATA
*     and SQLIND must point in order to retrieve data-compatible
*     results. Use the global numeric table and the large
*     character string buffer from which pieces can be allocated.

*     First find the base type of the current column.

*     Note: Normally you should clear the SQLIND pointer if it
*     is not being used using the SET TO NULL statement. At the
*     time of this writing, however, SET pointer-item TO NULL
*     was not accepted. The pointer will be ignored by
*     Ingres if the SQLTYPE is positive.

      IF (SQLTYPE(COLN) > 0) THEN
           MOVE SQLTYPE(COLN) TO BASE-TYPE
           SET NOT-NULLABLE TO TRUE
*          SET SQLIND(COLN) TO NULL
      ELSE
           COMPUTE BASE-TYPE = 0 - SQLTYPE(COLN)
           SET NULLABLE TO TRUE
           SET SQLIND(COLN) TO ADDRESS OF IND-DATA(COLN)
      END-IF.

*     Collapse all different types into one of
*     integer, decimal or character.

*     Integer data uses 4-byte COMP.

      IF (BASE-TYPE = IISQ-INT-TYPE) THEN

               MOVE IISQ-INT-TYPE TO SQLTYPE(COLN)
               MOVE 4 TO SQLLEN(COLN)
               SET SQLDATA(COLN) TO ADDRESS OF INT-DATA(COLN)

*      Money and floating-point data or decimal data use COMP-3
*
*      Note: You must encode precision and length when settin
*      SQLLEN for a decimal data type. Use the formula: SQLLEN =
*      (256 * p+s) where p is the Ingres precision and s l
*      is scale of the decimal host variable. DEC-DATA is
*      defined as PIC S9(10)V9(8), so p = 10 + 8 (Ingres
*      precision is the total number of digits.) and s = 8.
*      Therefore, SQLLEN = (256 * 18 + 8) = 4616.

       ELSE IF (BASE-TYPE = IISQ-MNY-TYPE) OR
                (BASE-TYPE = IISQ-DEC-TYPE) OR
                (BASE-TYPE = IISQ-FLT-TYPE) THEN

                MOVE IISQ-DEC-TYPE TO SQLTYPE(COLN)
                MOVE 4616          TO SQLLEN(COLN)
                SET SQLDATA(COLN) TO ADDRESS OF DEC-DATA(COLN)

*      Dates, fixed and varying-length character
*      strings use character data.

       ELSE IF (BASE-TYPE = IISQ-DTE-TYPE)
             OR (BASE-TYPE = IISQ-CHA-TYPE)
             OR (BASE-TYPE = IISQ-VCH-TYPE)
             OR (BASE-TYPE = IISQ-LVCH-TYPE) THEN
```

```
*       Fix up the lengths of dates and determine the length
*       of the sub-string required from the large character
*       string buffer.

             IF (BASE-TYPE = IISQ-DTE-TYPE) THEN
                  MOVE IISQ-DTE-LEN TO SQLLEN(COLN)
             END-IF
             IF (BASE-TYPE = IISQ-LVCH-TYPE) THEN
*      Maximize the length of a large object to 100
*      for this example.

                    MOVE 100 TO SQLLEN(COLN)
             END-IF

             MOVE IISQ-CHA-TYPE TO SQLTYPE(COLN)
             MOVE SQLLEN(COLN) TO CUR-LEN

*      If we do not have enough character space left
*      print an error.

             IF ((CHAR-LEN + CUR-LEN) > 2500) THEN
                    DISPLAY "SQL Error: Character result "
                            "data overflow."
                    SET SETUP-FAIL TO TRUE
             ELSE

*      There is enough space so point at the start of the
*      corresponding sub-string. Allocate space out of
*      character buffer and accumulate the currently used
*      character space.

                   SET SQLDATA(COLN) TO ADDRESS OF
                          CHAR-DATA(CHAR-LEN:)
                   ADD CUR-LEN TO CHAR-LEN
              END-IF
       END-IF.

*      If nullable negate the data type
       IF (NULLABLE) THEN
            COMPUTE SQLTYPE(COLN) = 0 - SQLTYPE(COLN)
       END-IF.

**
*      Paragraph: PRINT-ROW
*
*      For each result column inside the SQLDA, print the
*      value. Print its column number too in order to
*      identify it with a column name printed earlier in
*      SETUP-ROW. If the value is NULL print "N/A".The
*      details of the printing are done in PRINT-COLUMN.
**
       PRINT-ROW.


*      Reset the character counter to the first byte.
       MOVE 1 TO CHAR-LEN.
*      Process each column.

       PERFORM PRINT-COLUMN
              VARYING COLN FROM 1 BY 1
              UNTIL (COLN > SQLD).

*      End each line of column data.

       DISPLAY SPACE.
```

```
**
* Paragraph: PRINT-COLUMN
*
*      Detailed printing of PRINT-ROW. This paragraph does
*      not attempt to tabulate the results in a tabular
*      format. The display formats used can be modified if
*      more precision is required.
**

       PRINT-COLUMN.

*      For each column print the number and value of the column.
*      NULL columns are printed as "N/A".

       DISPLAY "[" COLN "] " WITH NO ADVANCING.

*      Find the base type of the current column.

       IF (SQLTYPE(COLN) > 0) THEN
             MOVE SQLTYPE(COLN) TO BASE-TYPE
             SET NOT-NULLABLE TO TRUE
       ELSE
             COMPUTE BASE-TYPE = 0 - SQLTYPE(COLN)
             SET NULLABLE TO TRUE
       END-IF.

*      Different types have been collapsed into one of
*      integer, decimal or character. If the data is NULL
*      then just print "N/A".

       IF (NULLABLE AND (IND-DATA(COLN) = -1)) THEN

       DISPLAY "N/A" WITH NO ADVANCING

*      Integer data.

       ELSE IF (BASE-TYPE = IISQ-INT-TYPE) THEN
               MOVE INT-DATA(COLN) TO DISP-INT
               DISPLAY DISP-INT WITH NO ADVANCING

*      Decimal, money and float column data will also
*      be printed  here.

      ELSE IF (BASE-TYPE = IISQ-DEC-TYPE) THEN

              MOVE DEC-DATA(COLN) TO DISP-DEC
              DISPLAY DISP-DEC WITH NO ADVANCING
*      Character data. Print only the relevant substring.

       ELSE IF (BASE-TYPE = IISQ-CHA-TYPE) THEN

             MOVE SQLLEN(COLN) TO CUR-LEN
             DISPLAY CHAR-DATA(CHAR-LEN:CUR-LEN)
                   WITH NO ADVANCING
             ADD CUR-LEN TO CHAR-LEN

       END-IF.

*      Add trailing space after each value.
       IF (COLN < SQLD) THEN
            DISPLAY SPACE WITH NO ADVANCING.
```

```
**
*  Paragraph: PRINT-ERROR
*
*      SQLCA error detected. Retrieve the error message and
*      print it.
**

       PRINT-ERROR.

       EXEC SQL INQUIRE_SQL (:ERROR-BUF = ERRORTEXT) END-EXEC.
       DISPLAY "SQL Error:".
       DISPLAY ERROR-BUF.

**
* Paragraph: READ-STATEMENT
*
*      Prompt user and read input SQL statement. This paragraph
*      can be expanded to scan and process an SQL statement
*      string searching
*      for delimiters (such as quotes and the semicolon).
*      Currently the user is allowed to input only one SQL e
*      statement on on line without any terminators. Blank or
*      empty lines will causthe normal termination of this
*      program.
**
       READ-STATEMENT.

       DISPLAY STMT-NUM ">" WITH NO ADVANCING.
       ACCEPT STMT-BUF FROM CONSOLE.
       IF (STMT-BUF = SPACES) THEN
            DISPLAY "***************************"
            SET DONE-READING TO TRUE.
```

VMS

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQL-MONITOR.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

* Include SQL Communications and Descriptor Areas
    EXEC SQL INCLUDE SQLCA END-EXEC.
    EXEC SQL INCLUDE SQLDA END-EXEC.
* Dynamic SQL statement name (documentary only)
    EXEC SQL DECLARE stmt STATEMENT END-EXEC.
* Cursor declaration for dynamic statement
    EXEC SQL DECLARE csr CURSOR FOR stmt END-EXEC.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
* Database name
     01 DB-NAME              PIC X(30).
* Dynamic SQL statement buffer
     01 STMT-BUF             PIC X(1000).
* SQL error message buffer
     01 ERROR-BUF            PIC X(1024).
EXEC SQL END DECLARE SECTION END-EXEC.
* SQL statement number
 01 STMT-NUM             PIC 999.
* Reading state
 01 READING-STMT PIC S9(4) USAGE COMP.
     88 DONE-READING     VALUE 0.
     88 STILL-READING    VALUE 1.
```

```
* Number of rows affected by last SQL statement
 01 STMT-ROWS            PIC ZZZZZ9.
* Number of rows retrieved by last SELECT statement
 01 SELECT-ROWS          PIC S9(8) USAGE COMP.
* Dynamic SELECT statement set up state
 01 SELECT-SETUP         PIC S9(4) USAGE COMP.
     88 SETUP-FAIL       VALUE 0.
     88 SETUP-OK         VALUE 1.
* Index into SQLVAR table
 01 COL                  PIC 999.
* Base data type of SQLVAR item without nullability
 01 BASE-TYPE            PIC S9(4) USAGE COMP.
* Is a result column type nullable
 01 IS-NULLABLE            PIC S9(4) USAGE COMP.
     88 NOT-NULLABLE       VALUE 0.
     88 NULLABLE           VALUE 1.

* Global result data storage. This pool of data includes the maximum
* number of result data items needed to execute a Dynamic SELECT
* statement. There is a table of 1024 integers, decimal, large object
* handlers, and null indicator data items, and a large character
* string buffer. Note: Floating-point and money types are stored in
* decimal variables.
 01 RESULT-DATA.
    02 INTEGER-DATA OCCURS 1024 TIMES.
       03 INT-DATA         PIC S9(9) USAGE COMP.
       03 IND-DATA         PIC S9(4) USAGE COMP.
    02 DECIMAL-DATA OCCURS 1024 TIMES.
       03 DEC-DATA         PIC S9(10)V9(8) USAGE COMP-3.
    02 STRING-DATA.
       03 CHAR-LEN         PIC S9(4) USAGE COMP.
       03 CHAR-DATA        PIC X(2500).
   02 BLOB-DATA  OCCURS 1024 TIMES.
       03 BLOB-ARG        USAGE POINTER.
       03 BLOB-HDLR       PIC S9(9) USAGE COMP.

* User defined handler for large objects
 01 UsrDatHdlr        PIC S9(9) USAGE COMP VALUE EXTERNAL UsrDataHdlr
* Limit the size of a large object
 01 BLOB-MAX          PIC S9(4) USAGE COMP IS EXTERNAL.
* Current lengths of local character data.
 01 CUR-LEN           PIC S9(4) USAGE COMP.
**
* Procedure Division: SQL-MONITOR
*
* Main entry of SQL Monitor application. Prompt for database name
* and connect to the database. Run the monitor and disconnect from
* the database. Before disconnecting roll back any pending updates.
**
PROCEDURE DIVISION.
SBEGIN.
* Prompt for database name.
      DISPLAY "SQL Database: " WITH NO ADVANCING.
      ACCEPT DB-NAME AT END STOP RUN.
      DISPLAY " -- SQL Terminal Monitor -- ".
* Treat connection errors as fatal.
      EXEC SQL WHENEVER SQLERROR STOP END-EXEC.
      EXEC SQL CONNECT :DB-NAME END-EXEC.
* Run the Terminal Monitor
      PERFORM RUN-MONITOR.
      EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
      DISPLAY "SQL: Exiting monitor program.".
      EXEC SQL ROLLBACK END-EXEC.
      EXEC SQL DISCONNECT END-EXEC.
      STOP RUN.
```

```
**
* Paragraph: RUN-MONITOR
*
* Run the SQL monitor. Initialize the global SQLDA with the number
* of SQLVAR elements. Loop while prompting the user for input;
* if end-of-file is detected then return to the calling paragraph
* (the main program). If the user inputs a statement, execute it
* (using paragraph EXECUTE-STATEMENT).
**
RUN-MONITOR.
* Initialize the SQLN (set the number of SQLVAR elements)
        MOVE 1024 TO SQLN.
* If you increase BLOB-MAX then increase BLOB_DATA in the datahandler
        MOVE 50 TO BLOB-MAX.
* Now we are setup for input. Initialize statement number and
* reading state.
        MOVE 0 TO STMT-NUM.
        SET STILL-READING TO TRUE.
* Loop while prompting, reading and processing the SQL statement.
        PERFORM UNTIL DONE-READING
                ADD 1 TO STMT-NUM
                PERFORM READ-STATEMENT
                IF (STILL-READING) THEN
                        PERFORM EXECUTE-STATEMENT THRU END-EXECUTE
                END-IF
        END-PERFORM.
**
* Paragraph: EXECUTE-STATEMENT
*
* Using the PREPARE and DESCRIBE facilities determine if the input
* statement is a SELECT statement or not. If the statement is not
* a SELECT statement then EXECUTE it, otherwise open a cursor and
* process a dynamic SELECT statement (using paragraph EXECUTE-SELECT).
* After processing the statement, print the number of rows affected
* by the statement and any SQL errors.
**
EXECUTE-STATEMENT.
        EXEC SQL WHENEVER SQLERROR GO TO END-EXECUTE END-EXEC.
* PREPARE and DESCRIBE the statement. Inspect the contents of the
* SQLDA and determine if it is a SELECT statement or not.
        EXEC SQL PREPARE stmt FROM :STMT-BUF END-EXEC.
        EXEC SQL DESCRIBE stmt INTO :SQLDA END-EXEC.
* If SQLD = 0 then this is not a SELECT.
            IF (SQLD = 0) THEN
                EXEC SQL EXECUTE stmt END-EXEC
                MOVE SQLERRD(3) TO STMT-ROWS

* Otherwise this is a SELECT. Verify that there are enough SQLVAR
* result variables. If there are too few print an error and continue,
* otherwise call EXECUTE-SELECT.
        ELSE
                IF (SQLD > SQLN) THEN
                        DISPLAY "SQL Error: SQLDA requires more than "
                              "1024 result variables."
                        MOVE 0 TO STMT-ROWS
                ELSE
                        PERFORM EXECUTE-SELECT THRU END-SELECT
                        MOVE SELECT-ROWS TO STMT-ROWS
                END-IF
       END-IF.
```

```
* Print the number of rows processed.
        DISPLAY "[" STMT-ROWS " row(s)]".
* Only print the error message if we arrived at this label because
* of an SQL error.
END-EXECUTE.
        EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
        IF (SQLCODE < 0) THEN
                PERFORM PRINT-ERROR.
**
* Paragraph: EXECUTE-SELECT
*
* Execute a Dynamic SELECT statement. The SQLDA has already been
* described, so print the table header column names, open a
* dynamic cursor, and retrieve and print the results. Accumulate
* the number of rows processed in SELECT-ROWS.
**
EXECUTE-SELECT.
* So far no rows.
        MOVE 0 TO SELECT-ROWS.
* Set up the result types and data items, and print result column
* names SETUP-ROW will set SETUP-FAIL/OK if it fails/succeeds.
        PERFORM SETUP-ROW.
        IF (SETUP-FAIL) THEN
                GO TO END-SELECT.
        EXEC SQL WHENEVER SQLERROR GO TO SELECT-ERR END-EXEC.
* Open the dynamic cursor.
        EXEC SQL OPEN csr FOR READONLY END-EXEC.
* Fetch and print each row. Accumulate the number of rows fetched.
        PERFORM UNTIL SQLCODE NOT = 0
                EXEC SQL FETCH csr USING DESCRIPTOR :SQLDA END-EXEC
                IF (SQLCODE = 0) THEN
                       ADD 1 TO SELECT-ROWS
                       PERFORM PRINT-ROW
                END-IF
        END-PERFORM.
SELECT-ERR.
        EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.

* Only print the error message if we arrived at this label because
* of an SQL error.
        IF (SQLCODE < 0) THEN
               PERFORM PRINT-ERROR.
        EXEC SQL CLOSE csr END-EXEC.
END-SELECT.
       EXIT.
**
* Paragraph: SETUP-ROW
*
* A statement has just been described so set up the SQLDA for result
* processing. Print all the column names and allocate result data
* items for retrieving data using paragraph SETUP-COLUMN.
*
* This paragraph sets SETUP-OK if it succeeds, and SETUP-FAIL if
* there was some sort of initialization error (in SETUP-COLUMN).
**
SETUP-ROW.
* Initialize column setup. No character data used yet.
          SET SETUP-OK TO TRUE.
          MOVE 1 TO CHAR-LEN.
* Process each column.
          PERFORM SETUP-COLUMN
                   VARYING COL FROM 1 BY 1
                   UNTIL (COL > SQLD) OR (SETUP-FAIL).
```

```
* At this point we've processed all columns for data type
* information. End the line of column names.
          DISPLAY SPACE.
          DISPLAY "----------------------------".
**
* Paragraph: SETUP-COLUMN
*
* When setting up for a SELECT statement column names are printed,
* and result data items (for retrieving data) are chosen out of a
* pool of variables (integers, floating-points, a large character
* string space, and null indicators). The SQLDATA and SQLIND fields
* are pointed at the addresses of the result data items and
* indicators. Paragraph sets SETUP-FAIL if it fails.
**
SETUP-COLUMN.
* For each column print the number and name of the column, e.g.:
*        [001] sal [002] name [003] age
        DISPLAY "[" COL "] " WITH NO ADVANCING.
        DISPLAY SQLNAMEC(COL)(1:SQLNAMEL(COL)) WITH NO ADVANCING.
        IF (COL < SQLD) THEN
              DISPLAY SPACE WITH NO ADVANCING.
* Determine the data type of the column and to where SQLDATA and
* SQLIND must point in order to retrieve data-compatible results. Use
* the global numeric table and the large character string buffer from
* which pieces can be allocated.

* First find the base type of the current column.
        IF (SQLTYPE(COL) > 0) THEN
            MOVE SQLTYPE(COL) TO BASE-TYPE
            SET NOT-NULLABLE TO TRUE
            MOVE 0 TO SQLIND(COL)
        ELSE
            COMPUTE BASE-TYPE = 0 - SQLTYPE(COL)
            SET NULLABLE TO TRUE
            SET SQLIND(COL) TO REFERENCE IND-DATA(COL)
        END-IF.
* Collapse all different types into one of integer, float or
* character.
* Integer data uses 4-byte COMP.
         IF (BASE-TYPE = 30) THEN
                IF (NOT-NULLABLE) THEN
                        MOVE 30 TO SQLTYPE(COL)
                ELSE
                        MOVE -30 TO SQLTYPE(COL)
                END-IF
                MOVE 4 TO SQLLEN(COL)
                SET SQLDATA(COL) TO REFERENCE INT-DATA(COL)
* Money, decimal and floating-point data use COMP-3.
*
* Note: You must encode precision and length when setting SQLLEN
* for a decimal data type. Use the formula: SQLLEN = (256 * p+s)
* where p is the Ingres precision and s is scale of the Decimal
* host variable. DEC-DATA is defined as PIC S9(10)V9(8), so
* p = 10+8 (Ingres precision is the total number of digits)
* and s= 8. Therefore, SQLLEN = (256 * 18+8) = 4616.
        ELSE IF (BASE-TYPE = 5)
             OR (BASE-TYPE = 10)
             OR (BASE-TYPE = 31) THEN
                IF (NOT-NULLABLE) THEN
                         MOVE 10 TO SQLTYPE(COL)
                ELSE
                         MOVE -10 TO SQLTYPE(COL)
                END-IF
                MOVE 4616 TO SQLLEN(COL)
                SET SQLDATA(COL) TO REFERENCE DEC-DATA(COL)
```

```
* Dates, fixed and varying-length character strings use
* character data.
        ELSE IF (BASE-TYPE = 3) OR (BASE-TYPE = 20)
             OR (BASE-TYPE = 21) THEN
* Fix up the lengths of dates and determine the length of the
* sub-string required from the large character string buffer.
                  IF (BASE-TYPE = 3) THEN
                            MOVE 25 TO SQLLEN(COL)
                  END-IF
                  IF (NOT-NULLABLE) THEN
                            MOVE 20 TO SQLTYPE(COL)
                  ELSE
                            MOVE -20 TO SQLTYPE(COL)
                  END-IF
                  MOVE SQLLEN(COL) TO CUR-LEN

* If we do not have enough character space left print an error.
                  IF ((CHAR-LEN + CUR-LEN) > 2500) THEN
                     DISPLAY "SQL Error: Character result "
                             "data overflow."
                     SET SETUP-FAIL TO TRUE
                  ELSE
* There is enough space so point at the start of the corresponding
* sub-string. Allocate space out of character buffer and accumulate
* the currently used character space.
                     SET SQLDATA(COL) TO REFERENCE CHAR-DATA(CHAR-LEN:)
                     ADD CUR-LEN TO CHAR-LEN
                  END-IF
* For Long Varchar use Datahandler

       ELSE IF (BASE-TYPE = 22) THEN
                 IF (NOT-NULLABLE) THEN
                      MOVE 46 TO SQLTYPE(COL)
                 ELSE
                      MOVE -46 TO SQLTYPE(COL)
                 END-IF

                 SET SQLDATA(COL) TO REFERENCE BLOB-DATA(COL)
                 MOVE UsrDataHdlr to BLOB-HDLR(COL)
                 MOVE BLOB-MAX TO SQLLEN(COL)
                 MOVE SQLLEN(COL) TO CUR-LEN
* If we do not have enough character space left print an error.
                 IF ((CHAR-LEN + CUR-LEN) > 2500) THEN
                          DISPLAY “SQL Error: Large object result “
                                   “data overflow.”
                          SET SETUP-FAIL TO TRUE
                 ELSE
* There is enough space so point at the start of the corresponding
* sub-string. Allocate space out of character buffer and accumulate
* the currently used character space.

                 SET BLOB-ARG(COL) TO REFERENCE CHAR-DATA(CHAR-LEN:)
                     ADD CUR-LEN TO CHAR-LEN
                 END-IF
          END-IF.
```

```
**
* Paragraph: PRINT-ROW
*
* For each result column inside the SQLDA, print the value. Print
* its column number too in order to identify it with a column name
* printed earlier in SETUP-ROW. If the value is NULL print "N/A".
* The details of the printing are done in PRINT-COLUMN.
**
PRINT-ROW.
* Reset the character counter to the first byte.
        MOVE 1 TO CHAR-LEN.
* Process each column.
        PERFORM PRINT-COLUMN
                VARYING COL FROM 1 BY 1
                UNTIL (COL > SQLD).
* End each line of column data.
        DISPLAY SPACE.
**
* Paragraph: PRINT-COLUMN
*
* Detailed printing of PRINT-ROW. This paragraph does not attempt
* to tabulate the results in a tabular format. Default formats are
* used (using WITH CONVERSION clause).
**
PRINT-COLUMN.
* For each column print the number and value of the column.
* NULL columns are printed as "N/A".
          DISPLAY "[" COL "] " WITH NO ADVANCING.

* Find the base type of the current column.
          IF (SQLTYPE(COL) > 0) THEN
                MOVE SQLTYPE(COL) TO BASE-TYPE
                SET NOT-NULLABLE TO TRUE
          ELSE
                COMPUTE BASE-TYPE = 0 - SQLTYPE(COL)
                SET NULLABLE TO TRUE
          END-IF.
* Different types have been collapsed into one of integer, float or
* character. If the data is NULL then just print "N/A".
          IF (NULLABLE AND (IND-DATA(COL) = -1)) THEN
                DISPLAY "N/A" WITH NO ADVANCING
* Integer data.
          ELSE IF (BASE-TYPE = 30) THEN
               DISPLAY INT-DATA(COL) WITH CONVERSION WITH NO ADVANCING
* Decimal data.
          ELSE IF (BASE-TYPE = 10) THEN
               DISPLAY DEC-DATA(COL) WITH CONVERSION WITH NO ADVANCING
* Character and large object data. Print only the relevant substring.
          ELSE IF (BASE-TYPE = 20)
               OR (BASE-TYPE = 46) THEN
                  MOVE SQLLEN(COL) TO CUR-LEN
                  DISPLAY CHAR-DATA(CHAR-LEN:CUR-LEN) WITH NO ADVANCING
                  ADD CUR-LEN TO CHAR-LEN
          END-IF.
* Add trailing space after each value.
         IF (COL < SQLD) THEN
                DISPLAY SPACE WITH NO ADVANCING.
**
* Paragraph: PRINT-ERROR
*
* SQLCA error detected. Retrieve the error message and print it.
**
PRINT-ERROR.
        EXEC SQL INQUIRE_SQL (:ERROR-BUF = ERRORTEXT) END-EXEC.
        DISPLAY "SQL Error:".
        DISPLAY ERROR-BUF.
```

```
**
* Paragraph: READ-STATEMENT
*
* Prompt user and read input SQL statement. This paragraph can be
* expanded to scan and process an SQL statement string searching
* for delimiters (such as quotes and the semicolon). Currently
* the user is allowed to input only one SQL statement on one
* line without any terminators. Blank lines or Control Z
* will cause normal termination of the program.
**
READ-STATEMENT.
        DISPLAY STMT-NUM "> " WITH NO ADVANCING.
        ACCEPT STMT-BUF AT END SET DONE-READING TO TRUE.

        IF (STMT-BUF = SPACES) THEN
                 SET DONE-READING TO TRUE.
END PROGRAM SQL-MONITOR.
*****************************************************
IDENTIFICATION DIVISION.
PROGRAM-ID. UsrDataHdlr.
ENVIRONMENT DIVISION.
DATA DIVISION.

WORKING-STORAGE SECTION.

* Include SQL Communications and Descriptor Areas
EXEC SQL INCLUDE SQLCA END-EXEC.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  SEG-BUG           PIC X(100).
 01  SEG-LEN           PIC S9(6) USAGE COMP.
 01  DATA-END          PIC S9(6) USAGE COMP.
 01  MAX-LEN           PIC S9(6) USAGE COMP.
 01  TOTAL-LEN         PIC S9(6) USAGE COMP.
EXEC SQL END DECLARE SECTION END-EXEC.

* Limit the size of a large object.
 01 BLOB-MAX        PIC S9(4) USAGE COMP IS EXTERNAL.
 01 P               PIC S9(6) USAGE COMP.

LINKAGE-SECTION.
 01 BLOB-DATA       PIC X(50).
PROCEDURE DIVISION USING BLOB-DATA.
BEGIN.

       EXEC SQL WHENEVER SQLERROR CALL SQLPRINT END-EXEC.

       MOVE BLOB-MAX TO MAX-LEN.
       MOVE 0 TO DATA-END.
       MOVE 0 TO TOTAL-LEN.
       PERFORM UNTIL DATA-END = 1
                    OR TOTAL-LEN NOT < BLOB-MAX

                EXEC SQL GET DATA (:SEG-BUF = SEGMENT,
                                    :SEG-LEN = SEGMENTLENGTH,
                                      :DATA-END = DATAEND
                               WITH MAXLENGTH = :MAX-LEN
                               END-EXEC
```

```
                ADD TOTAL-LEN 1 GIVING P
                STRING SEG-BUG DELIMITED BY SIZE INTO BLOB-DATA WITH
                          POINTER P

                 ADD SEG-LEN TO TOTAL-LEN

        END-PERFORM.

        IF DATA-END = 0 THEN
                 EXEC SQL ENDDATA END-EXEC.

END PROGRAM UsrDataHdlr.
```

A Dynamic SQL/Forms Database Browser

This program lets the user browse data from and insert data into any table in any database, using a dynamically defined form. The program uses Dynamic SQL and Dynamic FRS statements to process the interactive data. You should already have used VIFRED to create a Default Form based on the database table that you want to browse. VIFRED will build a form with fields that have the same names and data types as the columns of the specified database table.

When run, the program prompts the user for the name of the database, the table and the form. The form is profiled using the describe form statement, and the field name, data type and length information is processed. From this information, the program fills in the SQLDA data and null indicator areas, and builds two Dynamic SQL statement strings to select data from and insert data into the database.

The Browse menu item retrieves the data from the database using an SQL cursor associated with the dynamic select statement, and displays that data using the dynamic putform statement. A submenu allows the user to continue with the next row or return to the main menu. The Insert menu item retrieves the data from the form using the dynamic getform statement, and adds the data to the database table using a prepared insert statement. The Save menu item commits the user's changes and, because prepared statements are discarded, reprepares the select and insert statements. When the Quit menu item is selected, all pending changes are rolled back and the program is terminated.

For readability, all Embedded SQL words are in uppercase.

Windows **UNIX**

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  DYNAMIC-FRS.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.

*      Include SQL Communications and Descriptor Areas
       EXEC SQL INCLUDE SQLCA END-EXEC.
       EXEC SQL INCLUDE SQLDA END-EXEC.
```

```
*      Dynamic SQL SELECT and INSERT statements (documentary only)
       EXEC SQL DECLARE sel_stmt STATEMENT END-EXEC.
       EXEC SQL DECLARE ins_stmt STATEMENT END-EXEC.

*      Cursor declaration for dynamic statement
       EXEC SQL DECLARE csr CURSOR FOR sel_stmt END-EXEC.

       EXEC SQL BEGIN DECLARE SECTION END-EXEC.

*      Database, form and table names
       01 DB-NAME        PIC X(40).
       01 FORM-NAME      PIC X(40).
       01 TABLE-NAME     PIC X(40).

*      Dynamic SQL SELECT and INSERT statement buffers
       01 SEL-BUF        PIC X(1000).
       01 INS-BUF        PIC X(1000).

*      Error status and prompt error return buffer
       01 ERR            PIC S9(8) USAGE COMP.
       01 RET            PIC X.

       EXEC SQL END DECLARE SECTION END-EXEC.

*      DESCRIBE-FORM (form profiler) return state
       01 DESCRIBED          PIC S9(4) USAGE COMP.
          88 DESCRIBE-FAIL   VALUE 0.
          88 DESCRIBE-OK     VALUE 1.

*      Index into SQLVAR table
       01 COLN               PIC S9(4) USAGE COMP.

*      Base data type of SQLVAR item without nullability
       01 BASE-TYPE          PIC S9(4) USAGE COMP.

*      Is a result column type nullable
       01 IS-NULLABLE        PIC S9(4) USAGE COMP.
          88 NOT-NULLABLE   VALUE 0.
          88 NULLABLE       VALUE 1.

*      Global result data storage.  This pool of data includes the
*      maximum number of data items needed to execute a dynamic
*      retrieval or insertion.  There is a table of 1024 integer,
*      decimal and null indicator data items, and a large
*      character string buffer from which sub-strings are
*      allocated.  Note: Floating-point and money types are stored
*      in decimal variables.
       01 RESULT-DATA.
          02 ARRAY-STORAGE OCCURS IISQ-MAX-COLS TIMES.
              03 INTEGERS          PIC S9(9) USAGE COMP-5 SYNC.
              03 DECIMALS          PIC S9(10)V9(8) USAGE COMP-3.
              03 INDICATORS        PIC S9(4) USAGE COMP-5 SYNC.
          02 CHARS                PIC X(3000).
*      Total used length of data buffer
          02 CHAR-CNT             PIC S9(4) USAGE COMP VALUE 1.
*      Current length required from character data buffer
          02 CHAR-CUR             PIC S9(4) USAGE COMP.

*      Buffer for building Dynamic SQL statement string names
       01 NAMES                    PIC X(1000) VALUE SPACES.
       01 NAME-CNT                 PIC S9(4) USAGE COMP VALUE 1.

*      Buffer for collecting Dynamic SQL place holders
       01 MARKS                    PIC X(1000) VALUE SPACES.
       01 MARK-CNT                 PIC S9(4) USAGE COMP VALUE 1.
```

```
**
* Procedure Division: DYNAMIC-FRS
*
*     Main body of Dynamic SQL forms application.  Prompt for
*     database, form and table name.  Perform DESCRIBE-FORM
*     to obtain a profile of the form and set up the SQL
*     statements.  Then allow the user to interactively browse
*     the database table and append new data.
**

       PROCEDURE DIVISION.
       EXAMPLE SECTION.
       XBEGIN.

*      Turn on forms system

       EXEC FRS FORMS END-EXEC.

*      Prompt for database name - will abort on errors

       EXEC SQL WHENEVER SQLERROR STOP END-EXEC.
       EXEC FRS PROMPT ('Database name: ', :DB-NAME) END-EXEC.
       EXEC SQL CONNECT :DB-NAME END-EXEC.

       EXEC SQL WHENEVER SQLERROR CALL SQLPRINT END-EXEC.

*      Prompt for table name - later a Dynamic SQL SELECT
*      statement will be built from it.

       EXEC FRS PROMPT ('Table name: ', :TABLE-NAME) END-EXEC.


*      Prompt for form name.  Check forms errors through
*      INQUIRE_FRS.

       EXEC FRS PROMPT ('Form name: ', :FORM-NAME) END-EXEC.
       EXEC FRS MESSAGE 'Loading form ...' END-EXEC.
       EXEC FRS FORMINIT :FORM-NAME END-EXEC.
       EXEC FRS INQUIRE_FRS FRS (:ERR = ERRORNO) END-EXEC.
       IF (ERR > 0) THEN
            EXEC FRS MESSAGE 'Could not load form.
                   Exiting.' END-EXEC
            EXEC FRS ENDFORMS END-EXEC
            EXEC SQL DISCONNECT END-EXEC
            STOP RUN.
*      Commit any work done so far - access of forms catalogs

       EXEC SQL COMMIT END-EXEC.

*      Describe the form and build the SQL statement strings

       PERFORM DESCRIBE-FORM THROUGH END-DESCRIBE.
       IF (DESCRIBE-FAIL) THEN
           EXEC FRS MESSAGE 'Could not describe form.  Exiting.'
               END-EXEC
           EXEC FRS ENDFORMS END-EXEC
           EXEC SQL DISCONNECT END-EXEC
           STOP RUN.

*      PREPARE the SELECT and INSERT statements that correspond
*      to the menu items Browse and Insert.  If the Save menu item
*      is chosen the statements are reprepared.
```

```
              EXEC SQL PREPARE sel_stmt FROM :SEL-BUF END-EXEC.
              MOVE SQLCODE TO ERR.
              EXEC SQL PREPARE ins_stmt FROM :INS-BUF END-EXEC.
              IF (ERR < 0) OR (SQLCODE < 0) THEN
                  EXEC FRS MESSAGE
                       'Could not prepare SQL statements.
                                  Exiting.' END-EXEC
                  EXEC FRS ENDFORMS END-EXEC
                  EXEC SQL DISCONNECT END-EXEC
                  STOP RUN.
       *      Display the form and interact with user, allowing browsing
       *      and the inserting of new data.

              EXEC FRS DISPLAY :FORM-NAME FILL END-EXEC
              EXEC FRS INITIALIZE END-EXEC

              EXEC FRS ACTIVATE MENUITEM 'Browse' END-EXEC
              EXEC FRS BEGIN END-EXEC

       *      Retrieve data and display the first row on the form,
       *      allowing the user to browse through successive rows.  If
       *      data types from the database table are not consistent with
       *      data descriptions obtained from the form, a retrieval
       *      error will occur.  Inform the user of this or other errors.
       *
       *      Note that the data will return sorted by the first field
       *      that was described, as the SELECT statement, sel_stmt,
       *      included an ORDER BY clause.

                    EXEC SQL OPEN csr FOR READONLY END-EXEC.

       *      Fetch and display each row

              FETCH-NEXT-ROW.
                    IF (SQLCODE NOT= 0) THEN
                         GO TO END-FETCH-NEXT.

                    EXEC SQL FETCH csr USING DESCRIPTOR :SQLDA END-EXEC.

                    IF (SQLCODE NOT= 0) THEN
                         EXEC SQL CLOSE csr END-EXEC
                         EXEC FRS PROMPT NOECHO ('No more rows :', :RET)
                              END-EXEC
                         EXEC FRS CLEAR FIELD ALL END-EXEC
                         EXEC FRS RESUME END-EXEC.

                    EXEC FRS PUTFORM :FORM-NAME USING DESCRIPTOR :SQLDA
                        END-EXEC.
                    EXEC FRS INQUIRE_FRS FRS (:ERR = ERRORNO) END-EXEC.
                    IF (ERR > 0) THEN
                         EXEC SQL CLOSE csr END-EXEC
                         EXEC FRS RESUME END-EXEC.

       *      Display data before prompting user with submenu

                    EXEC FRS REDISPLAY END-EXEC.

                    EXEC FRS SUBMENU END-EXEC
                    EXEC FRS ACTIVATE MENUITEM 'Next', FRSKEY4 END-EXEC
                    EXEC FRS BEGIN END-EXEC
       *      Continue with cursor loop

                         EXEC FRS MESSAGE 'Next row ...' END-EXEC.
                         EXEC FRS CLEAR FIELD ALL END-EXEC.
```

```
            EXEC FRS END END-EXEC

            EXEC FRS ACTIVATE MENUITEM 'End', FRSKEY3 END-EXEC
            EXEC FRS BEGIN END-EXEC

                EXEC SQL CLOSE csr END-EXEC.
                EXEC FRS CLEAR FIELD ALL END-EXEC.
                EXEC FRS RESUME END-EXEC.

            EXEC FRS END END-EXEC

*     Fetch next row

            GO TO FETCH-NEXT-ROW.
*     End of row processing

      END-FETCH-NEXT.
            CONTINUE.

      EXEC FRS END END-EXEC

      EXEC FRS ACTIVATE MENUITEM 'Insert' END-EXEC
      EXEC FRS BEGIN END-EXEC

          EXEC FRS GETFORM :FORM-NAME USING DESCRIPTOR :SQLDA
               END-EXEC.
          EXEC FRS INQUIRE_FRS FRS (:ERR = ERRORNO) END-EXEC.
          IF (ERR > 0) THEN
               EXEC FRS CLEAR FIELD ALL END-EXEC
               EXEC FRS RESUME END-EXEC.


               EXEC SQL EXECUTE ins_stmt USING DESCRIPTOR :SQLDA
                  END-EXEC.
               IF (SQLCODE < 0) OR (SQLERRD(3) = 0) THEN
               EXEC FRS PROMPT NOECHO
                    ('No rows inserted :', :RET) END-EXEC
          ELSE
               EXEC FRS PROMPT NOECHO
                   ('One row inserted :', :ret) END-EXEC.

       EXEC FRS END END-EXEC

       EXEC FRS ACTIVATE MENUITEM 'Save' END-EXEC
       EXEC FRS BEGIN END-EXEC
*       COMMIT any changes and then re-PREPARE the SELECT and
*       INSERT statements as the COMMIT statements discards them.

              EXEC SQL COMMIT END-EXEC.
              EXEC SQL PREPARE sel_stmt FROM :SEL-BUF END-EXEC.
              MOVE SQLCODE TO ERR.
              EXEC SQL PREPARE ins_stmt FROM :INS-BUF END-EXEC.
              IF (ERR < 0) OR (SQLCODE < 0) THEN
                  EXEC FRS PROMPT NOECHO
                       ('Could not reprepare SQL
                                   statements :', :RET)
                        END-EXEC
                  EXEC FRS BREAKDISPLAY END-EXEC.
      EXEC FRS END END-EXEC

      EXEC FRS ACTIVATE MENUITEM 'Clear' END-EXEC
      EXEC FRS BEGIN END-EXEC

              EXEC FRS CLEAR FIELD ALL END-EXEC.

      EXEC FRS END END-EXEC
```

```
       EXEC FRS ACTIVATE MENUITEM 'Quit', FRSKEY2 END-EXEC
       EXEC FRS BEGIN END-EXEC

             EXEC SQL ROLLBACK END-EXEC.
             EXEC FRS BREAKDISPLAY END-EXEC.
       EXEC FRS END END-EXEC
       EXEC FRS FINALIZE END-EXEC.

       EXEC FRS ENDFORMS END-EXEC.
       EXEC SQL DISCONNECT END-EXEC.

       STOP RUN.

**
* Paragraph: DESCRIBE-FORM
*
*      Profile the specified form for name and data type
*      information.  Using the DESCRIBE FORM statement, the SQLDA
*      is loaded with field information from the form.  This h
*      paragraph (together with the DESCRIBE-COLUMN paragraph) n
*      processes the form informatio to allocate result storage,
*      point at storage for dynamic FRS
*      data retrieval and assignment, and build SQL statements
*      strings for subsequent dynamic SELECT and INSERT
*      statements.  For example, assume the form (and table) 'emp'
*      has the following fields:
*
*                Field Name    Type     Nullable?
*                ----------    ----     ---------
*                name          char(10)  No
*                age           integer4  Yes
*                salary        money     Yes
*
*        Based on 'emp', this paragraph will construct the SQLDA.
*        The paragraph allocates variables from a result variable
*        pool (integers, floats and a large character string
*        space).  The SQLDATA and SQLIND fields are pointed at the
*        addresses of the result variables in the pool.  The
*        following SQLDA is built:
*
*                SQLVAR(1)
*                         SQLTYPE  =  CHAR TYPE
*                         SQLLEN   =  10
*                         SQLDATA  =  pointer into CHARS buffer
*                         SQLIND   =  null
*                         SQLNAME  =  'name'
*                SQLVAR(2)
*                         SQLTYPE   = - INTEGER TYPE
*                         SQLLEN    = 4
*                         SQLDATA   = address of INTEGERS(2)
*                         SQLIND    = address of INDICATORS(2)
*                         SQLNAME   = 'age'
*                SQLVAR(3)
*                         SQLTYPE   = - DECIMAL TYPE
*                         SQLLEN    = 4616 (see below)
*                         SQLDATA   = address of DECIMALS(3)
*                         SQLIND    = address of INDICATORS(3)
*                         SQLNAME   = 'salary'
*
*      This paragraph also builds two dynamic SQL statements
*      strings.
*      Note that the paragraph should be extended to verify that
*      the statement strings do fit into the statement buffers
*      (this was not done in this example).  The above example
*      would construct the following statement strings:
```

```
*
*         'SELECT name, age, salary FROM emp ORDER BY name'
*         'INSERT INTO emp (name, age, salary) VALUES (?, ?, ?)'
*

*       This paragraph sets DESCRIBE-OK if it succeeds, and
*       DESCRIBE-FAIL if there was some sort of initialization
*       error.
**
        DESCRIBE-FORM.

*       Initialize the SQLDA and DESCRIBE the form.  If we cannot
*       fully describe the form (our SQLDA is too small) then
*       report an error and return.

        SET DESCRIBE-OK TO TRUE.

        MOVE IISQ-MAX-COLS TO SQLN.
        EXEC FRS DESCRIBE FORM :FORM-NAME ALL INTO
                             :SQLDA END-EXEC.
        EXEC FRS INQUIRE_FRS FRS (:ERR = ERRORNO) END-EXEC.
        IF (ERR > 0) THEN
            SET DESCRIBE-FAIL TO TRUE
            GO TO END-DESCRIBE.
        IF (SQLD > SQLN) THEN
            EXEC FRS PROMPT NOECHO
                  ('SQLDA is too small for form :', :RET) END-EXEC
            SET DESCRIBE-FAIL TO TRUE
             GO TO END-DESCRIBE.
        IF (SQLD = 0) THEN
             EXEC FRS PROMPT NOECHO
             ('There are no fields in the form :', :RET) END-EXEC
             SET DESCRIBE-FAIL TO TRUE
             GO TO END-DESCRIBE.
*       For each field determine the size and type of the
*       result data area.   This is done by DESCRIBE-COLUMN.
*
*       If a table field type is returned then issue an error.
*
*       Also, for each field add the field name to the 'NAMES'
*       buffer and the SQL place holders '?' to the 'MARKS'
*       buffer, which will be used to build the final SELECT and
*       INSERT statements.

        PERFORM DESCRIBE-COLUMN
              VARYING COLN FROM 1 BY 1
              UNTIL (COLN > SQLD) OR (DESCRIBE-FAIL).

*       At this point we've processed all columns for data type
*       information.
*       Create final SELECT and INSERT statements.  For the SELECT
*       statement ORDER BY the first field.
        STRING "SELECT " NAMES(1: NAME-CNT) " FROM "
              TABLE-NAME " ORDER BY "
              SQLNAMEC(1)(1: SQLNAMEL(1))
              DELIMITED BY SIZE INTO SEL-BUF.
        STRING "INSERT INTO " TABLE-NAME "("
              NAMES(1: NAME-CNT) ") VALUES ("
              MARKS(1: MARK-CNT) ")"
              DELIMITED BY SIZE INTO INS-BUF.

        END-DESCRIBE.
              EXIT.
```

```
**
*       Paragraph: DESCRIBE-COLUMN
*
*       When setting up data for the SQLDA result data items are
*       chosen out of a pool of variables.  The SQLDATA and SQLIND
*       fields are pointed at the addresses of the result data
*       items and indicators as described in paragraph
*       DESCRIBE-FORM.
*
*       Field names are collected for the building of the Dynamic
*       SQL statement strings as described for paragraph
*       DESCRIBE-FORM.
*
*       Paragraph sets DESCRIBE-FAIL if it fails.
**

        DESCRIBE-COLUMN.

*       Determine the data type of the field and to where SQLDATA
*       and SQLIND must point in order to retrieve type-compatible
*       results.

*       First find the base type of the current column.

*       Note: Normally you should clear the SQLIND pointer if it
*       is not being used using the SET TO NULL statement.  At the
*       time of this writing, however, SET pointer-item TO NULL
*       was not accepted.  The pointer will be ignored by
*       Ingres if the SQLTYPE is positive.

        IF (SQLTYPE(COLN) > 0) THEN
            MOVE SQLTYPE(COLN) TO BASE-TYPE
            SET NOT-NULLABLE TO TRUE
            SET SQLIND(COLN) TO NULL
        ELSE
            COMPUTE BASE-TYPE = 0 - SQLTYPE(COLN)
            SET NULLABLE TO TRUE
            SET SQLIND(COLN) TO ADDRESS OF INDICATORS(COLN)
        END-IF.

*       Collapse all different types into one of integer,
*       float or character.

*       Integer data uses 4-byte COMP.

        IF (BASE-TYPE = IISQ-INT-TYPE) THEN

            MOVE IISQ-INT-TYPE TO SQLTYPE(COLN)
            MOVE 4 TO SQLLEN(COLN)
            SET SQLDATA(COLN) TO ADDRESS OF INTEGERS(COLN)

*       Money and floating-point or decimal use COMP-3.
*

*       Note: You must encode precision and length when setting
*       SQLLEN for a decimal data type.  Use the formula: SQLLEN =
*       (256 * p+s) where p is the Ingres precision and s
*       is scale of the decimal host variable.DEC-DATA is defined
*       as PIC S9(10)V9(8), so p = 10+8 (Ingres precision
*       is the total number of digits) and s= 8.  Therefore, SQLLEN
*       - (256 * 18 + 8) = 4616.

        ELSE IF (BASE-TYPE = IISQ-MNY-TYPE)
             OR (BASE-TYPE = IISQ-DEC-TYPE)
             OR (BASE-TYPE = IISQ-FLT-TYPE) THEN
```

```
                 MOVE IISQ-DEC-TYPE TO SQLTYPE(COLN)
                 MOVE 4616 TO SQLLEN(COLN)
                 SET SQLDATA(COLN) TO ADDRESS OF DECIMALS(COLN)

      *      Dates, fixed and varying-length character strings use
      *      character data.

             ELSE IF (BASE-TYPE = IISQ-DTE-TYPE)
                   OR (BASE-TYPE = IISQ-CHA-TYPE)
                   OR (BASE-TYPE = IISQ-VCH-TYPE) THEN

      *      Fix up the lengths of dates and determine the length of
      *      the sub-string required from the large character string
      *      buffer.

                   IF (BASE-TYPE = IISQ-DTE-TYPE) THEN
                        MOVE IISQ-DTE-LEN TO SQLLEN(COLN)
                   END-IF
                   MOVE IISQ-CHA-TYPE TO SQLTYPE(COLN)
                   MOVE SQLLEN(COLN) TO CHAR-CUR

      *      If we do not have enough character space left display an
      *      error.

                   IF ((CHAR-CNT + CHAR-CUR) > 3000) THEN
                        EXEC FRS PROMPT NOECHO
                         ('Character pool buffer overflow :', :RET) END-EXEC
                        SET DESCRIBE-FAIL TO TRUE
                   ELSE
      *      There is enough space so point at the start of the
      *      corresponding sub-string.  Allocate space out of character
      *       buffer and accumulate the currently used character space.

                      SET SQLDATA(COLN) TO ADDRESS OF CHARS(CHAR-CNT:)
                      ADD CHAR-CUR TO CHAR-CNT
                 END-IF

      *      Table fields are not allowed

             ELSE IF (BASE-TYPE = IISQ-TBL-TYPE) THEN
                 EXEC FRS PROMPT NOECHO
                     ('Table field found in form :', :RET) END-EXEC
                 SET DESCRIBE-FAIL TO TRUE
      *      Unknown data type

             ELSE

                 EXEC FRS PROMPT NOECHO
                         ('Invalid field type :', :RET) END-EXEC
                 SET DESCRIBE-FAIL TO TRUE

             END-IF.

      *      If nullable negate the data type
             IF (NULLABLE) THEN
                  COMPUTE SQLTYPE(COLN) = 0 - SQLTYPE(COLN)
             END-IF.

      *      Store field names and place holders (separated by commas)
      *      for the SQL statements.
```

```
IF (COLN > 1) THEN
    MOVE "," TO NAMES(NAME-CNT:1)
    ADD 1 TO NAME-CNT
    MOVE "," TO MARKS(MARK-CNT:1)
    ADD 1 TO MARK-CNT.
END-IF.

MOVE SQLNAMEC(COLN)(1:SQLNAMEL(COLN)) TO
            NAMES(NAME-CNT:SQLNAMEL(COLN)).
ADD SQLNAMEL(COLN) TO NAME-CNT.
MOVE "?" TO MARKS(MARK-CNT:1).
ADD 1 TO MARK-CNT.
```

VMS

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DYNAMIC-FRS.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

* Include SQL Communications and Descriptor Areas
  EXEC SQL INCLUDE SQLCA END-EXEC.
  EXEC SQL INCLUDE SQLDA END-EXEC.
* Dynamic SQL SELECT and INSERT statements (documentary only)
  EXEC SQL DECLARE sel_stmt STATEMENT END-EXEC.
  EXEC SQL DECLARE ins_stmt STATEMENT END-EXEC.
* Cursor declaration for dynamic statement
 EXEC SQL DECLARE csr CURSOR FOR sel_stmt END-EXEC.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
* Database, form and table names
    01 DB-NAME                PIC X(40).
    01 FORM-NAME              PIC X(40).
    01 TABLE-NAME             PIC X(40).
* Dynamic SQL SELECT and INSERT statement buffers
    01 SEL-BUF                PIC X(1000).
    01 INS-BUF                PIC X(1000).
* Error status and prompt error return buffer
    01 ERR                    PIC S9(8) USAGE COMP.
    01 RET                    PIC X.
EXEC SQL END DECLARE SECTION END-EXEC.
* DESCRIBE-FORM (form profiler) return state
  01 DESCRIBED             PIC S9(4) USAGE COMP.
      88 DESCRIBE-FAIL      VALUE 0.
      88 DESCRIBE-OK        VALUE 1.
* Index into SQLVAR table
  01 COL                    PIC S9(4) USAGE COMP.
* Base data type of SQLVAR item without nullability
  01 BASE-TYPE              PIC S9(4) USAGE COMP.
* Is a result column type nullable
  01 IS-NULLABLE            PIC S9(4) USAGE COMP.
      88 NOT-NULLABLE        VALUE 0.
      88 NULLABLE            VALUE 1.

* Global result data storage. This pool of data includes the maximum
* number of data items needed to execute a dynamic retrieval or
* insertion. There is a table of 1024 integer, floating-point and
* null indicator data items, and a large character string buffer
* from which sub-strings are allocated.
  01 RESULT-DATA.
     02 INTEGERS        PIC S9(9) USAGE COMP OCCURS 1024 TIMES.
     02 DECIMALS        PIC S9(10)V9(8) USAGE COMP-3 OCCURS 1024 TIMES.
     02 INDICATORS      PIC S9(4) USAGE COMP OCCURS 1024 TIMES.
     02 CHARS           PIC X(3000).
* Total used length of data buffer
```

```
  01 CHAR-CNT           PIC S9(4) USAGE COMP VALUE 1.

* Current length required from character data buffer
  01 CHAR-CUR           PIC S9(4) USAGE COMP.
* Buffer for building Dynamic SQL statement string names
  01 NAMES              PIC X(1000) VALUE SPACES.
  01 NAME-CNT           PIC S9(4) USAGE COMP VALUE 1.
* Buffer for collecting Dynamic SQL place holders
  01 MARKS              PIC X(1000) VALUE SPACES.
  01 MARK-CNT           PIC S9(4) USAGE COMP VALUE 1.
**
* Procedure Division: DYNAMIC-FRS
*
* Main body of Dynamic SQL forms application. Prompt for database,
* form and table name. Perform DESCRIBE-FORM to obtain a profile
* of the form and set up the SQL statements. Then allow the user
* to interactively browse the database table and append new data.
**
PROCEDURE DIVISION.
SBEGIN.
* Turn on forms system
      EXEC FRS FORMS END-EXEC.
* Prompt for database name - will abort on errors
      EXEC SQL WHENEVER SQLERROR STOP END-EXEC.
      EXEC FRS PROMPT ('Database name: ', :DB-NAME) END-EXEC.
      EXEC SQL CONNECT :DB-NAME END-EXEC.
      EXEC SQL WHENEVER SQLERROR CALL SQLPRINT END-EXEC.
* Prompt for table name - later a Dynamic SQL SELECT statement
* will be built from it.
      EXEC FRS PROMPT ('Table name: ', :TABLE-NAME) END-EXEC.
* Prompt for form name. Check forms errors reported through
* INQUIRE_FRS.
      EXEC FRS PROMPT ('Form name: ', :FORM-NAME) END-EXEC.
      EXEC FRS MESSAGE 'Loading form ...' END-EXEC.
      EXEC FRS FORMINIT :FORM-NAME END-EXEC.
      EXEC FRS INQUIRE_FRS FRS (:ERR = ERRORNO) END-EXEC.
      IF (ERR > 0) THEN
             EXEC FRS MESSAGE 'Could not load form. Exiting.' END-EXEC
             EXEC FRS ENDFORMS END-EXEC
             EXEC SQL DISCONNECT END-EXEC
             STOP RUN.
* Commit any work done so far - access of forms catalogs
      EXEC SQL COMMIT END-EXEC.
* Describe the form and build the SQL statement strings
      PERFORM DESCRIBE-FORM THROUGH END-DESCRIBE.
      IF (DESCRIBE-FAIL) THEN
           EXEC FRS MESSAGE 'Could not describe form. Exiting.'
               END-EXEC
           EXEC FRS ENDFORMS END-EXEC
           EXEC SQL DISCONNECT END-EXEC
          STOP RUN.

* PREPARE the SELECT and INSERT statements that correspond to the
* menu items Browse and Insert. If the Save menu item is chosen
* the statements are reprepared.
      EXEC SQL PREPARE sel_stmt FROM :SEL-BUF END-EXEC.
      MOVE SQLCODE TO ERR.
      EXEC SQL PREPARE ins_stmt FROM :INS-BUF END-EXEC.
      IF (ERR < 0) OR (SQLCODE < 0) THEN
           EXEC FRS MESSAGE
                 'Could not prepare SQL statements. Exiting.' END-EXEC
           EXEC FRS ENDFORMS END-EXEC
           EXEC SQL DISCONNECT END-EXEC
           STOP RUN.
* Display the form and interact with user, allowing browsing
* and the inserting of new data.
```

```
      EXEC FRS DISPLAY :FORM-NAME FILL END-EXEC
      EXEC FRS INITIALIZE END-EXEC
      EXEC FRS ACTIVATE MENUITEM 'Browse' END-EXEC
      EXEC FRS BEGIN END-EXEC
* Retrieve data and display the first row on the form, allowing
* the user to browse through successive rows. If data types
* from the database table are not consistent with data descriptions
* obtained from the form, a retrieval error will occur. Inform
* the user of this or other errors.
*
* Note that the data will return sorted by the first field that
* was described, as the SELECT statement, sel_stmt, included an
* ORDER BY clause.
      EXEC SQL OPEN csr FOR READONLY END-EXEC.
* Fetch and display each row
FETCH-NEXT-ROW.
            IF (SQLCODE NOT= 0) THEN
                GO TO END-FETCH-NEXT.
            EXEC SQL FETCH csr USING DESCRIPTOR :SQLDA END-EXEC.
            IF (SQLCODE NOT= 0) THEN
                EXEC SQL CLOSE csr END-EXEC
                EXEC FRS PROMPT NOECHO ('No more rows :', :RET)
                   END-EXEC
                EXEC FRS CLEAR FIELD ALL END-EXEC
                EXEC FRS RESUME END-EXEC.
            EXEC FRS PUTFORM :FORM-NAME USING DESCRIPTOR :SQLDA
                END-EXEC.
            EXEC FRS INQUIRE_FRS FRS (:ERR = ERRORNO) END-EXEC.
            IF (ERR > 0) THEN
                 EXEC SQL CLOSE csr END-EXEC
                 EXEC FRS RESUME END-EXEC.
* Display data before prompting user with submenu
            EXEC FRS REDISPLAY END-EXEC.
            EXEC FRS SUBMENU END-EXEC
            EXEC FRS ACTIVATE MENUITEM 'Next', FRSKEY4 END-EXEC
            EXEC FRS BEGIN END-EXEC

* Continue with cursor loop
                  EXEC FRS MESSAGE 'Next row ...' END-EXEC.
                  EXEC FRS CLEAR FIELD ALL END-EXEC.
            EXEC FRS END END-EXEC
            EXEC FRS ACTIVATE MENUITEM 'End', FRSKEY3 END-EXEC
            EXEC FRS BEGIN END-EXEC
                  EXEC SQL CLOSE csr END-EXEC.
                  EXEC FRS CLEAR FIELD ALL END-EXEC.
                  EXEC FRS RESUME END-EXEC.
            EXEC FRS END END-EXEC
* Fetch next row
            GO TO FETCH-NEXT-ROW.
* End of row processing
END-FETCH-NEXT.
            CONTINUE.
      EXEC FRS END END-EXEC
      EXEC FRS ACTIVATE MENUITEM 'Insert' END-EXEC
      EXEC FRS BEGIN END-EXEC
         EXEC FRS GETFORM :FORM-NAME USING DESCRIPTOR :SQLDA END-EXEC.
         EXEC FRS INQUIRE_FRS FRS (:ERR = ERRORNO) END-EXEC.
         IF (ERR > 0) THEN
              EXEC FRS CLEAR FIELD ALL END-EXEC
              EXEC FRS RESUME END-EXEC.
         EXEC SQL EXECUTE ins_stmt USING DESCRIPTOR :SQLDA END-EXEC.
         IF (SQLCODE < 0) OR (SQLERRD(3) = 0) THEN
               EXEC FRS PROMPT NOECHO ('No rows inserted :', :RET)
                 END-EXEC
```

```
         ELSE
         EXEC FRS PROMPT NOECHO ('One row inserted :', :ret)
               END-EXEC.
      EXEC FRS END END-EXEC
      EXEC FRS ACTIVATE MENUITEM 'Save' END-EXEC
      EXEC FRS BEGIN END-EXEC
* COMMIT any changes and then re-PREPARE the SELECT and INSERT
* statements as the COMMIT statements discards them.
              EXEC SQL COMMIT END-EXEC.
              EXEC SQL PREPARE sel_stmt FROM :SEL-BUF END-EXEC.
              MOVE SQLCODE TO ERR.
              EXEC SQL PREPARE ins_stmt FROM :INS-BUF END-EXEC.
              IF (ERR < 0) OR (SQLCODE < 0) THEN
                    EXEC FRS PROMPT NOECHO
                        ('Could not reprepare SQL statements :', :RET)
                        END-EXEC
                    EXEC FRS BREAKDISPLAY END-EXEC.
      EXEC FRS END END-EXEC
      EXEC FRS ACTIVATE MENUITEM 'Clear' END-EXEC
      EXEC FRS BEGIN END-EXEC
           EXEC FRS CLEAR FIELD ALL END-EXEC.
      EXEC FRS END END-EXEC

      EXEC FRS ACTIVATE MENUITEM 'Quit', FRSKEY2 END-EXEC
      EXEC FRS BEGIN END-EXEC
            EXEC SQL ROLLBACK END-EXEC.
            EXEC FRS BREAKDISPLAY END-EXEC.

      EXEC FRS END END-EXEC
      EXEC FRS FINALIZE END-EXEC.
      EXEC FRS ENDFORMS END-EXEC.
      EXEC SQL DISCONNECT END-EXEC.
      STOP RUN.
**
* Paragraph: DESCRIBE-FORM
*
* Profile the specified form for name and data type information.
* Using the DESCRIBE FORM statement, the SQLDA is loaded with
* field information from the form. This paragraph (together with
* the DESCRIBE-COLUMN paragraph) processes the form information
* to allocate result storage, point at storage for dynamic FRS
* data retrieval and assignment, and build SQL statements strings
* for subsequent dynamic SELECT and INSERT statements. For example,
* assume the form (and table) 'emp' has the following fields:
*
* Field Name Type    Nullable?
* ---------- ----    ---------
* name       char(10)  No
* age        integer4  Yes
* salary     money     Yes
*
* Based on 'emp', this paragraph will construct the SQLDA.
* The paragraph allocates variables from a result variable
* pool (integers, decimals and a large character string space).
* The SQLDATA and SQLIND fields are pointed at the addresses
* of the result variables in the pool. The following SQLDA
* is built:
*
```

```
*       SQLVAR(1)
*           SQLTYPE = CHAR TYPE
*           SQLLEN  = 10
*           SQLDATA = pointer into CHARS buffer
*           SQLIND  = null
*           SQLNAME = 'name'
*       SQLVAR(2)
*           SQLTYPE = - INTEGER TYPE
*           SQLLEN  = 4
*           SQLDATA = address of INTEGERS(2)
*           SQLIND  = address of INDICATORS(2)
*           SQLNAME = 'age'
*       SQLVAR(3)
*           SQLTYPE = - DECIMAL TYPE
*           SQLLEN  = 4616 (see below)
*           SQLDATA = address of DECIMALS(3)
*           SQLIND  = address of INDICATORS(3)
*           SQLNAME = 'salary'
*

* This paragraph also builds two dynamic SQL statements strings.
* Note that the paragraph should be extended to verify that the
* statement strings do fit into the statement buffers (this was
* not done in this example). The above example would construct
* the following statement strings:
*
* 'SELECT name, age, salary FROM emp ORDER BY name'
* 'INSERT INTO emp (name, age, salary) VALUES (?, ?, ?)'
*
* This paragraph sets DESCRIBE-OK if it succeeds, and
* DESCRIBE-FAIL if there was some sort of initialization error.
**
DESCRIBE-FORM.
* Initialize the SQLDA and DESCRIBE the form. If we cannot fully
* describe the form (our SQLDA is too small) then report an error
* and return.
          SET DESCRIBE-OK TO TRUE.
          MOVE 1024 TO SQLN.
          EXEC FRS DESCRIBE FORM :FORM-NAME ALL INTO :SQLDA END-EXEC.
          EXEC FRS INQUIRE_FRS FRS (:ERR = ERRORNO) END-EXEC.
          IF (ERR > 0 ) THEN
                SET DESCRIBE-FAIL TO TRUE
                GO TO END-DESCRIBE.
          IF (SQLD > SQLN) THEN
                EXEC FRS PROMPT NOECHO
                     ('SQLDA is too small for form :', :RET) END-EXEC
                SET DESCRIBE-FAIL TO TRUE
                GO TO END-DESCRIBE.
          IF (SQLD = 0) THEN
                EXEC FRS PROMPT NOECHO
                  ('There are no fields in the form :', :RET) END-EXEC
                SET DESCRIBE-FAIL TO TRUE
                GO TO END-DESCRIBE.
```

```
* For each field determine the size and type of the result data area.
* This is done by DESCRIBE-COLUMN.
*
* If a table field type is returned then issue an error.
*
* Also, for each field add the field name to the 'NAMES' buffer
* and the SQL place holders '?' to the 'MARKS' buffer, which
* will be used to build the final SELECT and INSERT statements.
      PERFORM DESCRIBE-COLUMN
                VARYING COL FROM 1 BY 1
                UNTIL (COL > SQLD) OR (DESCRIBE-FAIL).
* At this point we've processed all columns for data type
* information. Create final SELECT and INSERT statements. For the
* SELECT statement ORDER BY the first field.
     STRING "SELECT " NAMES(1: NAME-CNT) " FROM " TABLE-NAME
               " ORDER BY " SQLNAMEC(1)(1: SQLNAMEL(1))
               DELIMITED BY SIZE INTO SEL-BUF.
     STRING "INSERT INTO " TABLE-NAME "(" NAMES(1: NAME-CNT)
               ") VALUES (" MARKS(1: MARK-CNT) ")"
              DELIMITED BY SIZE INTO INS-BUF.
END-DESCRIBE.
      EXIT.
**
* Paragraph: DESCRIBE-COLUMN
*
* When setting up data for the SQLDA result data items are chosen
* out of a pool of variables. The SQLDATA and SQLIND fields are
* pointed at the addresses of the result data items and indicators
* as described in paragraph DESCRIBE-FORM.
*
* Field names are collected for the building of the Dynamic SQL
* statement strings as described for paragraph DESCRIBE-FORM.
*
* Paragraph sets DESCRIBE-FAIL if it fails.
**
DESCRIBE-COLUMN.
* Determine the data type of the filed and to where SQLDATA and
* SQLIND must point in order to retrieve type-compatible results.
* First find the base type of the current column.
            IF (SQLTYPE(COL) > 0) THEN
            MOVE SQLTYPE(COL) TO BASE-TYPE
            SET NOT-NULLABLE TO TRUE
            MOVE 0 TO SQLIND(COL)
      ELSE
            COMPUTE BASE-TYPE = 0 - SQLTYPE(COL)
            SET NULLABLE TO TRUE
            SET SQLIND(COL) TO REFERENCE INDICATORS(COL).
* Collapse all different types into one of integer, float
* or character.
* Integer data uses 4-byte COMP.
      IF (BASE-TYPE = 30) THEN
            IF (NOT-NULLABLE) THEN
                  MOVE 30 TO SQLTYPE(COL)
            ELSE
                  MOVE -30 TO SQLTYPE(COL)
            END-IF
            MOVE 4 TO SQLLEN(COL)
            SET SQLDATA(COL) TO REFERENCE INTEGERS(COL)
* Money and floating-point or decimal data use COMP-3.
* Note: You must encode precision and length when setting SQLLEN
* for a decimal data type. Use the formula: SQLLEN = (256 *p+s)
* where p is the Ingres precision and s is scale of the decimal
* host variable. DEC-DATA is defined as PIC S9(10)V9(8), so
* p = 10 + 8 (Ingres precision is the total number of digits)
* and s= 8. Therefore, SQLLEN = (256 * 18+8) = 4616.
```

```
      ELSE IF (BASE-TYPE = 5)
           OR (BASE-TYPE = 10)
           OR (BASE-TYPE = 31) THEN
          IF (NOT-NULLABLE) THEN
                MOVE 10 TO SQLTYPE(COL)
          ELSE
                MOVE -10 TO SQLTYPE(COL)
          END-IF
          MOVE 4616 TO SQLLEN(COL)
          SET SQLDATA(COL) TO REFERENCE DECIMALS(COL)

* Dates, fixed and varying-length character strings use
* character data.
      ELSE IF (BASE-TYPE = 3)
          OR  (BASE-TYPE = 20)
          OR  (BASE-TYPE = 21) THEN
* Fix up the lengths of dates and determine the length of the
* sub-string required from the large character string buffer.
          IF (BASE-TYPE = 3) THEN
              MOVE 25 TO SQLLEN(COL)
          END-IF
          IF (NOT-NULLABLE) THEN
              MOVE 20 TO SQLTYPE(COL)
          ELSE
              MOVE -20 TO SQLTYPE(COL)
          END-IF
          MOVE SQLLEN(COL) TO CHAR-CUR
* If we do not have enough character space left display an error.
          IF ((CHAR-CNT + CHAR-CUR) > 3000) THEN
              EXEC FRS PROMPT NOECHO
                  ('Character pool buffer overflow :', :RET) END-EXEC
              SET DESCRIBE-FAIL TO TRUE
          ELSE
* There is enough space so point at the start of the corresponding
* sub-string. Allocate space out of character buffer and accumulate
* the currently used character space.
              SET SQLDATA(COL) TO REFERENCE CHARS(CHAR-CNT:)
              ADD CHAR-CUR TO CHAR-CNT
          END-IF
* Table fields are not allowed
      ELSE IF (BASE-TYPE = 52) THEN
            EXEC FRS PROMPT NOECHO
                  ('Table field found in form :', :RET) END-EXEC
            SET DESCRIBE-FAIL TO TRUE
* Unknown data type
      ELSE
            EXEC FRS PROMPT NOECHO ('Invalid field type :', :RET)
                END-EXEC
            SET DESCRIBE-FAIL TO TRUE
      END-IF.
* Store field names and place holders (separated by commas)
* for the SQL statements.
     IF (COL > 1) THEN
          MOVE "," TO NAMES(NAME-CNT:1)
          ADD 1 TO NAME-CNT
          MOVE "," TO MARKS(MARK-CNT:1)
          ADD 1 TO MARK-CNT.
     END-IF.
     MOVE SQLNAMEC(COL)(1:SQLNAMEL(COL)) TO
              NAMES(NAME-CNT:SQLNAMEL(COL)).
     ADD SQLNAMEL(COL) TO NAME-CNT.
     MOVE "?" TO MARKS(MARK-CNT:1).
     ADD 1 TO MARK-CNT.
```

Chapter 4: Embedded SQL for Fortran

This chapter describes the use of Embedded SQL with the Fortran programming language.

Embedded SQL Statement Syntax for Fortran

This section describes the language-specific issues inherent in embedding SQL database and forms statements in a Fortran program. An Embedded SQL database statement has the following general syntax:

[*margin*] **exec sql** *SQL_statement*

The syntax of an Embedded SQL/FORMS statement is almost identical:

[*margin*] **exec frs** *SQL/FORMS_statement*

For information on SQL statements, see the *SQL Reference Guide*. For information on SQL/FORMS statements, see the *Forms-based Application Development Tools User Guide*.

The following sections describe the various syntactical elements of these statements as implemented in Fortran.

The preprocessor generates tab format code. As a result, tab characters instead of single spaces delimit various syntactical elements, such as labels.

Margin

In general, Embedded SQL statements in Fortran require no special margins. The **exec** keyword can begin anywhere on the source line as long as it is preceded only by blank space. Host declarations can also begin on any column. In the case, however, of an Embedded SQL statement continuation, the continuation indicator must follow the rules for line continuation. For more information, see Line Continuation in this chapter. For more information on tab format for source input, see the Preprocessor Operation in this chapter.

For portability to other implementations of SQL, you should code your SQL statements between columns 7 and 72.

Terminator

Unlike other Embedded SQL languages, there is no terminator for Fortran. For example, a **delete** statement embedded in a Fortran program looks like:

```
exec sql delete from employee where eno = :numvar
```

Do not follow an Embedded SQL statement on the same line with another Embedded SQL statement or with a Fortran statement. This causes preprocessor compile time syntax errors on the second statement. Only use white space (blanks and tabs) after the end of the statement to the end of the line.

The preprocessor allows, but does not require, a semicolon (;) to be a statement terminator for Embedded SQL statements. It does not write the semicolon to the output file it creates. The terminating semicolon can be convenient when entering source code directly from the terminal, using the **-s** flag on the preprocessor command line to test the syntax of a particular statement. For further details, see Preprocessor Operation in this chapter.

Labels

Like Fortran statements, Embedded SQL statements can have a label prefix. An Embedded SQL label is a Fortran statement number specified between columns 1 and 5. For example:

```
100 exec sql close cursor1
```

The label can appear anywhere a Fortran label can appear. However, labels cause the preprocessor to generate Fortran **continue** statements. Therefore, you should precede only *executable* SQL statements with labels. Although the preprocessor accepts the label in front of any **exec sql** or **exec frs** prefix, it may not be appropriate to code the label on some lines. For example, the following label, although acceptable to the preprocessor, later causes a compiler error:

```
101 exec sql include sqlca
```

The preprocessor reserves statement numbers 7000 through 12000.

Line Continuation

You can continue embedded SQL statements over multiple lines. The line continuation rules are the same as those for Fortran statements.

A line continuation indicator is:

- For UNIX, an ampersand (&) in the first column or any character in column 6, except a blank or zero

- For VMS, any digit, except zero, following the first tab
- For Windows, any character except zero or blank in column 6

The preprocessor considers the characters after the continuation indicator to be the first characters of the line. For example, the following **select** statement continues over four lines:

```
 exec sql select ename
1 into :namvar
2 from employee
3 where eno = :numvar
```

You can put blank lines between Embedded SQL statement lines. Blank lines do not require a continuation indicator. If a line continuation indicator is missing from an Embedded SQL statement that spans more than one line, the preprocessor generates the following error message: "Syntax error on terminator or missing Fortran continuation indicator".

You must use the continuation indicator to continue Embedded SQL/Fortran declarations over multiple lines. Comments (except comments that use the SQL comment delimiters—see Comments) cannot continue over multiple lines. In VMS, you cannot continue variable initialization clauses over multiple lines.

Comments

You can use the following in column 1 to indicate that a line is a comment:

- The letter "C"
- The asterisk (*)
- The lower case letter "c" (VMS and Windows)
- The exclamation point (!) (VMS and Windows)

The following example illustrates the correct use of the "C" comment delimiter:

```
       exec sql select ename
      1 into :namvar
      2 from employee
C Confirm that "eno" is the same as the current
C value chosen
      3 where eno = :curval
```

The VMS exclamation point can also be used anywhere on the statement line to mark a comment that extends to the end of the line as in the following example. However, this type of comment line cannot be continued over multiple lines:

```
exec sql delete from employee !Delete all employees
```

A comment line can appear anywhere in an Embedded SQL program that a blank line is allowed, with the following exceptions:

- In string constants. The preprocessor interprets such a comment as part of the string constant.
- Between component lines of Embedded SQL/FORMS block-type statements. All block-type statements (such as **activate** and **unloadtable**) are compound statements that include a statement section delimited by **begin** and **end**. Comment lines must not appear between the statement and its section. The preprocessor interprets such comments as Fortran host code, causing preprocessor syntax errors. For example, the following statement causes a syntax error on the first comment:

```
      exec frs unloadtable empform
                            employee (:namvar = ename)
C Illegal comment before statement body.
     exec frs begin
C Comment legal here
           exec frs message :namvar
     exec frs end
```

In VMS, you could also use an exclamation point on the following line with the C comment. For example:

```
C Comment legal here
     exec frs message :namvar     !And legal here too
```

- An example of a compound statement is the **display** statement, which typically consists of the **display** clause, an **initialize** section, **activate** sections, and a **finalize** section. The preprocessor translates these comments as host code, which causes syntax errors on subsequent statement components.
- In parts of statements that are dynamically defined. For example, a comment in a string variable specifying a form name is interpreted as part of the form name.

The SQL comment delimiter (--) indicates that the remainder of the line is a comment. In-line comments are not propagated to the host language file.

String Literals

Use single quotes to delimit Embedded SQL string literals. To embed a single quote in a string literal, use two single quotes, for example:

```
 exec sql update employee
1   set comments ='Doesn''t seem to relax'
2   where eno    = :numvar
```

You can continue string literals over multiple lines. Following Fortran rules, if the continued line ends without a closing quotation mark, the continuation line must follow the rules for continuation markers. The first character after the continuation marker is considered part of the string literal. For example:

```
 exec sql update employee
1   set comments = 'Completed all projects on time.
2   Recommended for promotion.'
3   where ename  = 'Jones'
```

Do not place comment lines between string literal continuation lines.

String Literals and Statement Strings

The Dynamic SQL statements **prepare** and **execute immediate** both use statement strings, which specify an SQL statement. The statement string can be specified by a string literal or character string variable, for example:

```
exec sql execute immediate 'drop employee'
str = 'drop employee'
exec sql execute immediate :str
```

As with regular Embedded SQL string literals, the statement string delimiter is the single quote. However, quotes embedded in statement strings must conform to the SQL *runtime* rules when the statement is executed. Notice the doubling of the single quote in the following dynamic **insert** statement:

```
exec sql prepare s1 from
1 'Insert into t1 values (''single''''double"'')'
```

The runtime evaluation of the previous statement string is:

```
Insert into t1 values ('single''double"')
```

which inserts the single'double" value into t1.

The Create Procedure Statement

As described in the *SQL Reference Guide,* the **create procedure** statement has language-specific syntax rules for line continuation, string literal continuation, comments, and the final terminator. These syntax rules follow the rules discussed in this section. For example, there is no final terminator. Regardless of the number of statements in the procedure's body, the preprocessor treats the **create procedure** statement as a single statement and, as an Embedded SQL/Fortran statement, it has no final terminator. However, all statements *within* the body of the procedure must end with a semicolon.

The following example shows a **create procedure** statement that follows the Embedded SQL/Fortran syntax rules:

```
     exec sql
     1 create procedure proc (parm integer) as
     2 declare
     3      var integer;
     4 begin
C Use Fortran comment line
     5       if (parm > 10) then
     6           message 'Fortran strings can continue
     7 over lines';
     8           insert into tab values (:parm);
     9       endif;
     1 end
```

Database procedures tend to be quite long, requiring a Fortran continuation indicator on each line. There is no limit over how many lines the **create procedure** statement can continue, even though the Fortran compiler may have a limit for host Fortran statements.

Fortran Variables and Data Types

This section describes how to declare and use Fortran program variables in Embedded SQL.

Variable and Type Declarations

The following sections describe variable and type declarations.

Embedded SQL Variable Declaration Sections

Embedded SQL statements use Fortran variables to transfer data between the database, or a form, and the program. You can also use Fortran constants in those SQL statements transferring data from the program into the database. You must declare Fortran variables, constants and structure definitions to SQL before using them in any Embedded SQL statements. The preprocessor does not allow the declaration of Fortran variables by implication. Fortran variables are declared to SQL in a *declaration section*. This section has the following syntax:

> **exec sql begin declare section**
> *Fortran variable declarations*
> **exec sql end declare section**

Embedded SQL variable declarations are global to the program file from the point of declaration onward. Multiple declaration sections can be incorporated into a single file, as is the case when a few different Fortran subprograms issue embedded statements using local variables. Each subprogram can have its own declaration section. For more information on the declaration of variables and types that are local to Fortran subprograms see The Scope of Variables in this chapter.

Reserved Words in Declarations

Fortran keywords are reserved, therefore you cannot declare types or variables with the same name as these keywords:

byte	**double**	**map**	**real**
character	**integer**	**parameter**	**record**
complex	**logical**	**precision**	**structure**

The Embedded SQL preprocessor does not distinguish between uppercase and lowercase in keywords. In generating Fortran code, it converts any uppercase letters in keywords to lowercase.

Typed Data Declarations

The preprocessor recognizes numeric variables declared with the following format:

data_type [******default_type_len*]
    var_name [******type_len*] [**(***array_spec***)**] [/*init_clause*/]
    {**,** *var_name* [******type_len*] [**(***array_spec***)**] [/*init_clause*/]}

The preprocessor recognizes character variables declared with the following format:

data_type [******default_type_len*[**,**]]
    var_name [**(***array_spec***)**] [******type_len*] [/*init_clause*/]
    {, *var_name* [**(***array_spec***)**] [******type_len*] [/*init_clause*/]}

Syntax Notes:

- A variable or type name must begin with an alphabetic character, which can be followed by alphanumeric characters. In VMS and Windows, it can also be followed by underscores.
- For information on the allowable *data_types*, see Data Types in this chapter.
- The *default_type_len* specifies the size of the variable being declared. For variables of numeric type, it must be represented by an integer literal of an acceptable length for the particular data type. For variables of **character** type, it can be represented by an integer literal or a parenthesized expression followed optionally by a comma. The preprocessor does not parse the length field for variables of type **character**. Note the default type lengths in the declarations shown below:

```
C Declares "eage" a 2-byte integer
    integer*2 eage
C Declares "stat" a 4-byte integer
    integer*4 stat
C Declares "ename" a character string
    character*(4+len) ename
```

- The *type_len* allows you to declare a variable with a length different from *default_type_len*. Again, you can use a parenthesized expression only to declare the length of character variable declarations. The type length for a numeric variable must be an integer literal representing an acceptable numeric size. For example:

```
C Default-sized integer and 2-byte integer
    integer length
    integer*2 height
    character*15 name, socsec*(numlen)
```

  Some UNIX Fortran compilers do not permit the length of a character variable to be redeclared.

- The data type and variable names must be legal Fortran identifiers beginning with an alphabetic character. In VMS, it can also begin with an underscore.

- The *array_spec* should conform to Fortran syntax rules. The preprocessor simply notes that the declared variable is an array but does not parse the *array_spec* clause. Note that if you specify both an array and a type length, the order of those two clauses differs depending on whether the variable being declared is of character or numeric type. The following are examples of array declarations:

```
C Array specification first
     character*16 enames(100), edepts(15)*10
C Type length first
     real*4 saltab(5,12), real*8 yrtots(12)
```

- The preprocessor allows you to initialize a variable or array in the declaration statement by means of the *init_clause*. The preprocessor accepts, but does not examine, any initial data. The Fortran compiler, however, will later detect any errors in the initial data. For example:

```
     real*8 initcash /512.56/
     character*4 baseyear /'1950'/
     character*4 year /1950/
C Acceptable to preprocessor but not to compiler
```

Do not continue initial data over multiple lines. If an initialization value is too long for the line, as could be the case with a string constant, instead use the Fortran **data** statement. For UNIX, *init_clause* is an extension to the F77 standard.

Constant Declarations

UNIX

You can declare constants to the preprocessor using the Fortran **parameter** statement using the following syntax:

> **parameter (***const_name* **=** *value* {**,** *const_name* **=** *value*}**)**

Syntax Notes:

- The preprocessor derives the data type of *const_name* from the data type of *value*. The F77 compiler uses implicit data typing; it derives the data type of *value* from the first letter of *const_name*. Be sure that the type of the specified value is the same as the implicit type derived from *const_name*.
- The *value* can be a **real**, **integer** or **character** literal. It cannot be an expression or a symbolic name.

The following example declaration illustrates the **parameter** statement:

```
C real constant
      parameter (pi = 3.14159 )
C integer and real
      parameter (bigint = 2147483648, bignum = 999999.99
```

VMS

You can declare constants to the preprocessor using the Fortran **parameter** statement using the following syntax:

> **parameter** *const_name* **=** *value* {**,** *const_name* **=** *value*}

Syntax Notes:

- The preprocessor and the compiler derive the data type of *const_name* from the data type of *value*. Neither the preprocessor nor the compiler make use of implicit data typing. Explicit data type declarations are not allowed in **parameter** statements.
- The *value* can be a **real**, **integer** or **character** literal. It cannot be an expression or a symbolic name.

The following example declarations illustrate the **parameter** statement:

```
parameter (pi = 3.14159 )    real constant

parameter (bigint = 2147483648,
                bignum = 999999.99) !integer and real
```

Windows

You can declare constants to the preprocessor using the Fortran **parameter** statement using the following syntax:

> **parameter** [(]*const_name* **=** *value* {**,** *const_name* **=** *value*}[)]

Syntax Notes:

- The preprocessor and the compiler derive the data type of *const_name* by an explicit type declaration statement in the same scoping unit or by the implicit typing rules in effect for the scoping unit. If the named constant is implicitly typed, it can appear in a subsequent type declaration only if that declaration confirms the implicit typing.
- The *value* can be an expression of any data type.

The following example declarations illustrate the **parameter** statement:

```
C real constant
```

```
    parameter pi = 3.14159
C integer and real
    parameter (bigint = 2147483648, bignum = 999999.99)
```

Data Types

The Embedded SQL preprocessor accepts the following elementary Fortran data types and maps them to corresponding Ingres data types. For a description of exact type mapping, see Data Type Conversion in this chapter.

Fortran Data Types	Ingres Data Types
integer**N* where *N* = 2 or 4	integer
logical	integer
logical**N* where *N* = 1, 2 or 4	integer
byte	integer
real	float
real**N* where *N* = 4 or 8	float
double precision	float
character**N* where *N* 0	character
real*8	decimal
integer	integer

The Integer Data Type

The Fortran compiler allows the default size of **integer** variables to be either two or four bytes in length, depending on whether the **-i2** compiler flag (UNIX), the **noi4** flag (VMS), or the /integer_size:16 or /4I2 (Windows) is set.

This feature is also supported in Embedded SQL/Fortran by means of the preprocessor flag **-i2**. This flag allows you to change the default size of **integer** variables to two from the normal default size of four bytes. For detailed information on this flag, see Preprocessor Operation in this chapter.

You can explicitly override the default size when declaring the Fortran variable to the preprocessor. To do so, you must specify a size indicator (*2 or *4) following the **integer** keyword, as these examples illustrate:

```
integer*2 smlint
integer*4 bigint
```

These declarations create Embedded SQL **integer** variables of two and four bytes, respectively, regardless of the default setting.

The preprocessor treats **byte** and **logical** data type as **integer** data types. A **logical** variable has a default size of either 2 or 4 bytes according to whether the **-i2** flag has been set. You can override this default size by using a size indicator of 1, 2, or 4. For example:

```
logical logl*1, log2*2, log4*4
```

The **byte** data type has a size of one byte. You cannot override this size.

You can use an **integer** or **byte** variable with any numeric-valued object to assign or receive numeric data. For example, you can use such a variable to set a field in a form or to select a column from a database table. It can also specify simple numeric objects, such as table field row numbers. You can use a **logical** variable to assign or receive integer data, although your program should restrict its value to 1 and 0, which map respectively to the Fortran logical values **.TRUE.** and **.FALSE.**.

The Real Data Type

The preprocessor accepts **real** and **double precision** as legal real data types. The preprocessor accepts both 4-byte and 8-byte **real** variables. It makes no distinction between an 8-byte **real** variable and a **double precision** variable. The default size of a **real** variable is four bytes. However, you can override this size if you use a size indicator (*8) after the **real** keyword. For example:

```
C 4-byte real variable
            real salary
C 8-byte real variable
            real*8 yrtoda
C 8-byte real variable
            double precision saltot
```

You can only use a **real** variable to assign or receive numeric data (both real, decimal, and integer). You cannot use it to specify numeric objects, such as table field row numbers.

VMS

The preprocessor expects the internal format of **real** and **double** precision variables to be the standard VAX format. For this reason, you should not compile your program with the **g_floating** qualifier.

The Character Data Type

Variables of type **character** are compatible with all Ingres character string objects. The preprocessor does not need to know the declared length of a character string variable to use it at runtime. Therefore, it does not check the validity of any expression or symbolic name used to declare the length of the string variable. You should ensure that your string variables are long enough to accommodate any possible runtime values. For example:

```
character*7         first
character*10        last
character*1         init
```

```
character*(bufsiz)  msgbuf
```

For information on the interaction between character string variables and Ingres data at runtime, see Runtime Character and Varchar Type Conversion in this chapter.

Character strings containing embedded single quotes are legal in SQL, for example:

```
mary's
```

User variables may contain embedded single quotes and need no special handling unless the variable represents the entire search condition of a where clause:

```
where :variable
```

In this case you must escape the single quote by reconstructing the *:variable* string so that any embedded single quotes are modified to double single quotes, as in:

```
mary''s
```

Otherwise, a runtime error will occur. For more information on escaping single quotes, see String Literals in this chapter.

Indicator Variables

An *indicator variable* is a 2-byte integer variable. There are three possible ways to use them in an application:

- In a statement that retrieves data from Ingres, you can use an indicator variable to determine if its associated host variable was assigned a **null**.
- In a statement that sends data to Ingres, you can use an indicator variable to assign a null to the database column, form field, or table field column.
- In a statement that retrieves character data from Ingres, you can use the indicator variable as a check that the associated host variable was large enough to hold the full length of the returned character string. You can use **SQLSTATE** to do this. Although you can use **SQLCODE** as well, it is preferable to use **SQLSTATE** because **SQLCODE** is a deprecated feature.

The following statements illustrate how to set an indicator variable:

```
C Indicator variable
      integer*2   ind
C Array of indicators
      integer*2   indarr(10)
```

When using an indicator array with a host structure, as described in the *SQL Reference Guide*, you must declare the indicator array as an array of 2-byte integers. In the above example, the "indarr" variable can be used as an indicator array with a structure assignment.

Structure and Record Declarations Syntax

The Embedded SQL preprocessor supports the declaration and use of user-defined structure variables. In UNIX, the structure variables are an extension to the F77 standard and may not be available on all Fortran compilers.

The syntax of a structure definition is:

> **structure** [/*structdef_name*/] [*field_namelist*]
> *field_declaration*
> {*field_declaration*}
> **end structure**

Syntax Notes:

- The *structdef_name* is optional only for a nested structure definition.
- The *field_namelist* is allowed only with a nested structure definition. Each name in the *field_namelist* constitutes a field in the enclosing structure.
- The *field_declaration* can be a typed data declaration, a nested structure declaration, a **union** declaration, a **record** declaration, or a **fill** item.

The syntax of a **union** declaration is:

> **union**
> *map_declaration*
> *map_declaration*
> {*map_declaration*}
> **end union**

where *map_declaration* is:

> **map**
> *field_declaration*
> {*field_declaration*}
> **end map**

To use a structure with Embedded SQL statements, you must both define the structure and declare the structure's record in the Embedded SQL declaration section of your program. The syntax of the **record** declaration is:

> **record** /*structdef_name*/ *structurename* {**,**[/*structdef_name*/]
> *structurename*}

Syntax Note:

The *structdef_name* must have been previously defined in a **structure** statement.

For information on the use of structure variables in Embedded SQL statements, see Structure Variables in this chapter.

The following example includes a structure definition and a record declaration:

```
structure /name_map/
        union
            map
                character*30 fullname
            end map
            map
                character*10 firstnm
                character*2  init
                character*18 lastnm
            end map
        end union
end structure

record /name_map/ empname
```

The next example shows the definition of a structure containing an array of nested structures:

```
structure /class_struct/
           character*10   subject
           integer*2      year
           structure      student(100)
C No structure definition name needed
                   character*12 name
                   byte       grade
         end structure
end structure

record /class_struct/ classrec
```

The DCLGEN Utility

DCLGEN (Declaration Generator) is a utility that maps the columns of a database table into a Fortran structure that can be included in a declaration section. The following command invokes DCLGEN from the operating system level:

dclgen *language dbname tablename filename structurename*

where:

- *language* is the Embedded SQL host language, in this case, "fortran"
- *dbname* is the name of the database containing the table
- *tablename* is the name of the database table
- *filename* is the output file into which the structure declaration is placed
- *structurename* is the name of the Fortran structure variable that the command generates. The command generates a structure definition named *structurename* followed by an underscore character (_). It also generates a **record** statement for the structure variable of *structurename*.

The DCLGEN utility creates the declaration file *filename*, containing a structure or a series of Fortran variables, if the **-f77** flag is used, corresponding to the database table. The file also includes a **declare table** statement that serves as a comment and identifies the database table and columns from which the variables were generated.

UNIX

DCLGEN has the option to map the columns of a database table into a series of Fortran variables rather than into a Fortran structure. This is useful if your Fortran compiler does not support structures. Specify the **-f77** flag to indicate this DCLGEN option as follows:

dclgen *-f77 language dbname tablename filename prefixname*

prefixname is required when *-f77* is used. This prefix is appended to the column names of the table to produce the Fortran variables.

After the file is generated, you can use an Embedded SQL **include** statement to incorporate it into the variable declaration section. The following example demonstrates how to use DCLGEN in a Fortran program.

Assume the "employee" table was created in the "personnel" database as:

```
exec sql create table employee
    (eno      smallint not null,
    ename     char(20) not null,
    age       integer1,
    job       smallint,
    sal       decimal(14,2) not null,
    dept      smallint)
```

When the DCLGEN system-level command is:

```
dclgen fortran personnel employee employee.dcl emprec
```

The "employee.dcl" file created by this command contains a comment, and three statements. The first statement is the **declare table** description of "employee," which serves as a comment. The second statement is a declaration of the Fortran structure "emprec_". The last statement is a **record** statement for "emprec". The contents of the "employee.dcl" file are:

```
c       Description of table employee from database personnel
         exec sql declare employee table
         1 (eno    smallint not null,
         1  ename  char(20) not null,
         1  age    integer1,
         1  job    smallint,
         1  sal    decimal(14,2) not null,
         1  dept   smallint)
          structure /emprec_/
                         integer*2      eno
                         character*20   ename
                         integer*2      age
                         integer*2      job
                         real*8         sal
                         integer*2      dept
          end structure
          record /emprec_/ emprec
```

UNIX

For this example the DCLGEN system-level command is:

```
dclgen -f77 fortran personnel employee employee.dcl emp
```

The "employee.dcl" file created by this command contains a comment, a **declare table** statement and the variable declarations. The **declare table** statement describes the employee table and serves as a comment. The exact contents of the "employee.dcl" file are:

```
C   Description of table employee from database personnel
        exec sql declare employee table
         1 (eno          smallint not null,
         1 ename         char(20) not null,
         1 age           integer1,
         1 job           smallint,
         1 sal           decimal(14,2) not null,
         1 dept          smallint)

             integer*2    empeno
             character*20 empename
             integer*2    empage
             integer*2    empjob
             real*8       empsal
             integer*2    empdept
```

The Ingres **integer1** data type is mapped to the Fortran **integer*2** data type, rather than to **byte.**

Include this file, by means of the Embedded SQL **include** statement, in an Embedded SQL declaration section:

```
exec sql begin declare section
      exec sql include 'employee.dcl'
exec sql end declare section
```

You can then use the variables in data manipulation statements.

The field names of the structure that DCLGEN generates are identical to the column names in the specified table. Therefore, if the column names in the table contain any characters that are illegal for host language variable names you must modify the name of the field before attempting to use the variable in an application.

DCLGEN and Large Objects

When a table contains a large object column, DCLGEN will issue a warning message and map the column to a zero length character string variable. You must modify the length of the generated variable before attempting to use the variable in an application.

For example, assume that the "job_description" table was created in the "personnel" database as:

```
create table job_description
```

```
      (job smallint, description long varchar))
```

and the DCLGEN system-level command is:

```
dclgen fortran personnel job_description jobs.dcl jobs_rec
```

The contents of the “jobs.dcl” file would be:

```
C Description of table job_description from database
C personnel
      exec sql declare job_description table
     1    (job          smallint not null,
                        description long varchar)

      structure /jobs_rec_/
                     integer*2     job
                     character*0    description
      end structure
      record /jobs_rec/ blobs_rec
```

The table definition when used with the **-f77** flag (assuming the prefix of “b_” was specified) results in the following DCLGEN generated output in “jobs.dcl”:

```
  exec sql declare job_description table
1        (job              smallint,
           description long varchar)

             character*0    b_description
```

Indicator Variables

An *indicator variable* is a 2-byte integer variable. You can use an indicator variable in an application in three ways:

- In a statement that retrieves data from Ingres, you can use an indicator variable to determine if its associated host variable was assigned a **null**.
- In a statement that sets data to Ingres, you can use an indicator variable to assign a null to the database column, form field, or table field column.
- In a statement that retrieves character (or byte) data from Ingres, you can use the indicator variable as a check that the associated host variable was large enough to hold the full length of the returned string. However, the preferred method is to use **SQLSTATE**.

The base type for a null indicator variable must be the integer type **integer*2**. For example:

```
C Indicator variable
      integer*2   indvar

C Array of indicator variables
      integer*2 indarr(10)
```

The word **indicator** is reserved.

When using an indicator array with a host structure (see Using Indicator Variables in this chapter), you must declare the indicator array as an array of integer*2 variables. In the above example, you can use the variable "indarr" as an indicator array with a structure assignment.

Declaring External Compiled Forms

You can precompile your forms in the Visual Forms Editor (VIFRED). This saves the time otherwise required at runtime to extract the form's definition from the database forms catalogs.

In UNIX, when you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in the C language. VIFRED prompts you for the name of the file. After creating the file, you can compile it into a linkable object module. For an outline of steps, see Linking Precompiled Forms in this chapter.

In Windows, when you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in the C language. VIFRED prompts you for the name of the file. After creating the file, you can compile it into a linkable object module. For an outline of steps, see Linking Precompiled Forms in this chapter.

In VMS, when you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in the MACRO language. VIFRED prompts you for the name of the file with the MACRO description. When the file is created, you can assemble it into a linkable object module with the VMS command that produces an object file containing a global symbol with the same name as your form:

macro *filename*

In UNIX, Windows, and VMS, before the Embedded SQL/FORMS statement **addform** can refer to this object, the object must be declared in an Embedded SQL declaration section, with the following syntax:

integer *formname*

Next, in order for the program to access the external form definition, you must declare the *formname* as an external symbol:

external *formname*

This second declaration must take place *outside* the Embedded SQL declaration section. (Note, however, that the previous declaration of *formname* as an integer must occur inside the declaration section, so that you can use *formname* with the **addform** statement.)

Syntax Notes:

- The *formname* is the actual name of the form. VIFRED gives this name to the address of the global object. The *formname* is also used as the title of the form in other Embedded SQL/FORMS statements.
- The **external** statement associates the object with the external form definition.

The following example shows a typical form declaration and illustrates the difference between using the form's object definition and the form's name:

```
      exec sql begin declare section
                integer empfrm
                ...

      exec sql end declare section

          external empfrm
                ...

C The global object
      exec frs addform :empfrm
C The name of the form
      exec frs display empfrm
          ...
```

Concluding Example

The following example demonstrate some simple Embedded SQL/Fortran declarations:

```
C Include error handling
            exec sql include sqlca
            exec sql begin declare section

C Variables of each data type
                byte              dbyte
                logical*1         dlog1
                logical*2         dlog2
                logical*4         dlog4
                logical           dlog
                integer*2         dint2
                integer*4         dint4
                integer           dint
                real*4            dreal4
                real*8            dreal8
                real              dreal
                double precision  ddoub

                parameter (max = 1000)

                character*12 dbname
                character*12 fname, tname, cname
```

```
C Structure with a union
                structure /person/
                     byte      age
                     integer   flags
                     union
                         map
                                 character*30  fullnm
                         end map
                         map
                                 character*12  first
                                 character*18  last
                         end map
                      end union
             end structure

             record /person/ person, ptable(MAX)

C From DCLGEN
             exec sql include 'employee.dcl'

C Compiled forms
             integer  empfrm, dptfrm

        exec sql end declare section
             external  empfrm, dptfrm
```

The Scope of Variables

You can reference all variables declared in an Embedded SQL declaration section and the preprocessor accepts them from the point of declaration to the end of the file. This may not be true for the Fortran compiler, which allows references to variables only in the scope of the program unit in which they were declared. If you have two unrelated subprograms in the same file, each of which contains a variable with the same name to be used by Embedded SQL, do *not* redeclare the variable to Embedded SQL. The preprocessor uses the data type information supplied the first time you declared the variable.

In the following program fragment, the variable *dbname* is passed as a parameter between two subroutines. In the first subroutine, the variable is a local variable. In the second subroutine, the variable is a formal parameter passed as a string to be used with the **connect** statement. The declaration of *dbname* in the second subroutine must not occur in an Embedded SQL declaration section. In both subroutines, the preprocessor uses the data attributes from the variable’s declaration in the first subroutine:

```
            subroutine Scopes

            exec sql include sqlca
            exec sql begin declare section
                                character*20 dbname
            exec sql end declare section

C Prompt for and read database name
            type *, 'Database: '
            accept *, dbname
            call OpenDb(dbname)
                                ...

            end
```

```
            subroutine OpenDb(dbname)

            exec sql include sqlca

            character*(*) dbname

            exec sql whenever sqlerror stop
C Declared to SQL in first subroutine
            exec sql connect :dbname
               ...

    end
```

Take special care when using variables in a **declare cursor** statement. The variables used in such a statement must also be valid in the scope of the **open** statement for that same cursor. The preprocessor actually generates the code for the **declare** at the point that the **open** is issued and, at that time, evaluates any associated variables. For example, in the following program fragment, even though the variable “number” is valid to the preprocessor at the point of both the **declare cursor** and **open** statements, it is not an explicitly declared variable name for the Fortran compiler at the point that the **open** is issued, possibly resulting in a runtime error. Because Fortran allows implicit variable declarations (although Embedded SQL does not), the compiler itself does not generate an error message. For example:

```
C This example contains an error
              subroutine IniCsr

              exec sql include sqlca

              exec sql begin declare section
C A local variable
                                  integer number
              exec sql end declare section

              exec sql declare cursor1 cursor for
           1     select ename, age
           2     from employee
           3     where eno = :number
C Initialize "number" to a particular value
                          ...

              end

              subroutine PrcCsr

              exec sql include sqlca

              exec sql begin declare section
                          character*16 ename
                          integer      eage
              exec sql end declare section

C Illegal evaluation of "number"

              exec sql open cursor1

              exec sql fetch cursor1 into :ename, :eage
                        ...

              end
```

You must issue the **include sqlca** statement in each subprogram that contains Embedded SQL statements.

Variable Usage

Fortran variables declared in an Embedded SQL declaration section can substitute for most elements of Embedded SQL statements that are not keywords. Of course, the variable and its data type must make sense in the context of the element. When you use a Fortran variable in an Embedded SQL statement, you must precede it with a colon (:). You must further verify that the statement using the variable is in the scope of the variable's declaration. As an example, the following **select** statement uses the variables "namevar" and "numvar" to receive data and the variable "idno" as an expression in the **where** clause:

```
 exec sql select ename, eno
1   into :namevar, :numvar
2   from employee
3   where eno = :idno
```

Various rules and restrictions apply to the use of Fortran variables in Embedded SQL statements. The following sections describe the usage syntax of different categories of variables and provide examples of such use.

Simple Variables

The following syntax refers to a simple scalar-valued variable (integer, real or character string):

:*simplename*

Syntax Notes:

- If you use the variable to send values to Ingres, it can be any scalar-valued variable.
- If you use the variable to receive values from Ingres, it must be a scalar-valued variable.

The following program fragment demonstrates a typical error handling routine, which can be called either directly or by a **whenever** statement. The variables "buffer" and "buflen" are scalar-valued variables:

```
subroutine ErrHnd

exec sql include sqlca

exec sql begin declare section
          parameter (buflen = 100)
          character*(buflen) buffer
exec sql end declare section
```

```
exec sql whenever sqlerror continue
exec sql inquire_sql (:buffer= errrortext)
print *, 'the following error occurred aborting session.'
print *, buffer
exec sql abort
        ...
end
```

Array Variables

The following syntax refers to an array variable:

:*arrayname* **(***subscripts***)**

Syntax Notes:

- You must subscript the variable because only scalar-valued elements (integers, reals and character strings) are legal SQL values.
- When you declare the array, the Embedded SQL preprocessor does not parse the array bounds specification. Consequently, the preprocessor accepts illegal bounds values. Also, when you reference an array, the preprocessor does not parse the subscript. The preprocessor confirms only that an array subscript is used with an array variable. You must ensure that the subscript is legal and that the correct number of indices is used.
- The preprocessor does not accept substring references for character variables.
- Arrays of indicator variables used with structure assignments must not include subscripts when referenced.

The following example uses the "i" variable as a subscript. It does not need to be declared in the declaration section because it is not parsed:

```
      exec sql begin declare section
                character*8 frmnam(3)
      exec sql end declare section

      integer i
                  frmnam(1) = "empfrm"
                  frmnam(2) = "dptfrm"
                  frmnam(3) = "hlpfrm"
      do 100 i=1,3
100      exec frs forminit:formname
```

Structure Variables

A structure variable can be used in two different ways if your Fortran compiler supports structures. First, the structure can be used as a simple variable, implying that all its members are used. This would be appropriate in the Embedded SQL **select**, **fetch**, and **insert** statements. Second, a member of a structure may be used to refer to a single element. This member must be a scalar value (integer, real or character string).

Using a Structure as a Collection of Variables

The syntax for referring to a complete structure is the same as referring to a simple variable:

:*structurename*

Syntax Notes:

- The *structurename* can refer to a main or nested structure. It can be an element of an array of structures. Any variable reference that denotes a structure is acceptable. For example:

```
C A simple structure
        :emprec
C An element of an array of structures
        :struct_array(i)
 C A nested structure at level 3
        :struct.minor2.minor3
```

- In order to be used as a collection of variables, the final structure in the reference must have no nested structures or arrays. All the members of the structure will be enumerated by the preprocessor and must have scalar values. The preprocessor generates code as though the program had listed each structure member in the order in which it was declared.
- You must not use a structure containing a **union** declaration when the structure is being used as a collection of variables. The preprocessor generates references to all components of the structure and ignores the **map** groupings. Using a **union** declaration results in either a “wrong number of errors” preprocessor error or a runtime error.

The following example uses the” employee.dcl” file that DCLGEN generates to retrieve values into a structure. This example is not applicable if DCLGEN was run with the **-f77** flag:

```
    exec sql begin declare section
             exec sql include 'employee.dcl'
    exec sql end declare section

    exec sql select *
1    into :emprec
2    from employee
3    where eno = 123
```

The example above generates code as though the following statement had been issued instead:

```
exec sql select *
1   into :emprec.eno, :emprec.ename, :emprec.age,
2       :emprec.job, :emprec.sal, :emprec.dept
3   from employee
4   where eno = 123
```

The example below fetches the values associated with all the columns of a cursor into a record:

```
exec sql begin declare section
```

```
            exec sql include 'employee.dcl'
exec sql end declare section

exec sql declare empcsr cursor for
1   select *
2   from employee
3   order by ename
             ...

exec sql open empcsr
        exec sql fetch empcsr into :emprec
exec sql close empcsr
```

The following example inserts values by looping through a locally declared array of structures whose elements have been initialized:

```
exec sql begin declare section
         exec sql declare person table
         1    (pname     char(30),
         2     page      integer1,
         3     paddr     varchar(50)

         structure /person_/
                        character*30  name
                        integer*2     age
                        character*50  addr
         end structure

         record /person_/ person(10)
         integer*2 I

exec sql end declare section
            ...

do i=1,10
            exec sql insert into person
            1   values (:person(i))
end do
```

The **insert** statement in the example above generates code as though the following statement had been issued instead:

```
exec sql insert into person
1 values (:person(i).name, :person(i).age,:person(i).addr)
```

Using a Structure Member

The syntax Embedded SQL uses to refer to a structure member is the same as in Fortran:

> **:***structure***.***member*{**.***member*}

Syntax Notes:

- The structure member denoted by the above reference must be a scalar value (integer, real or character string). There can be any combination of arrays and structures, but the last object referenced must be a scalar value. Thus, the following references are all legal:

```
C Member of a structure
       :employee.sal
C Member of an element of an array
       :person(3).name
C Deeply nested member
       :struct1.mem2.mem3.age
```

- Any array elements referred to *within* the structure reference, and not at the very end of the reference, are not checked by the preprocessor. Consequently, both of the following references are accepted, even though one must be wrong, depending on whether "person" is an array:

```
:person(1).age
:person.age
```

- The preprocessor expects unambiguous and fully qualified structure member references.

The following example uses the "emprec" structure, similar to the structure generated by DCLGEN, to put values into the "empform" form:

```
        exec sql begin declare section
                        structure /emprec_/
                                integer*2       eno
                                character*2     ename
                                integer*2       age
                                integer*2       job
                                real*8          sal
                                integer*2       dept
                        end structure

    record /emprec_/ emprec

exec sql end declare section
    ...

exec frs putform empform
1    (eno = :emprec.eno, ename = :emprec.ename,
2     age = :emprec.age, job = :emprec.job,
3     sal = :emprec.sal, dept = :emprec.dept)
```

Using Indicator Variables

The syntax for referring to an indicator variable is the same as for a simple variable, except that an indicator is always associated with a host variable:

:host_variable:indicator_variable

or

:host_variable **indicator** *:indicator_variable*

Syntax Notes:

- The indicator variable can be a simple variable or an array element that yields a 2-byte integer. For example:

```
integer*2 indvar, indarr(5)

:var_1:indvar
:var_2:indarr(2)
```

- If the host variable associated with the indicator variable is a structure, the indicator variable should be an array of 2-byte integers. In this case the array should *not* be dereferenced with a subscript.
- When an indicator array is used, the first element of the array corresponds to the first member of the structure, the second element with the second member, and so on. Array elements begin at subscript 1.

The following example uses the “employee.dcl” file that DCLGEN generates to retrieve non-null values into a structure and null values into the “empind” array:

```
exec sql begin declare section
         exec sql include 'employee.dcl'
         integer*2 empind(10)
exec sql end declare section
exec sql select *
1    into :emprec:empind
2    from employee
```

The previous example generates code as though the following statement had been issued:

```
exec sql select *
1   into :emprec.eno:empind(1), :emprec.ename:empind(2),
2        :emprec.age:empind(3), :emprec.job:empind(4),
3        :emprec.sal:empind(5), :emprec.dept:empind(6),
4   from employee
```

Data Type Conversion

A Fortran variable declaration must be compatible with the Ingres value it represents. Numeric Ingres values can be set by and retrieved into numeric variables, and Ingres character values can be set by and retrieved into character variables.

Data type conversion occurs automatically for different numeric types, such as from floating-point Ingres database column values into integer Fortran variables, and for character strings, such as from varying-length Ingres character fields into fixed-length Fortran character string buffers.

Ingres does not automatically convert between numeric and character types. You must use the Ingres type conversion operators, the Ingres **ascii** function, or a Fortran conversion routine for this purpose.

The following table shows the default type compatibility for each Ingres data type.

Ingres and Fortran Data Type Compatibility

Ingres Type	Fortran Type
char(*N*)	character**N* < 2000
varchar(*N*)	character**N* < 2000
integer1	integer*2
integer2	integer*2
smallint	integer*2
integer4	integer*4
integer	integer*4
bigint	integer*8
float4	real*4
float8	real*8
date	character*25
money	real*8
table_key	character*8
object_key	character*16
decimal	real* 8
long varchar	character**N* > 2000

Runtime Numeric Type Conversion

The Ingres runtime system provides automatic data type conversion between numeric-type values in the database and the forms system and numeric Fortran variables. The standard type conversion rules in UNIX are followed according to standard Fortran rules. In VMS, the standard VAX rules are followed. For example, if you assign a **real** variable to an integer-valued field, the digits after the decimal point of the variable's value are truncated. Runtime errors are generated for overflow on conversion.

The default size of integers in Embedded SQL/Fortran is four bytes. You can change the default size to two bytes by means of the **-i2** preprocessor flag. If you use this flag, you should also compile the program with the **-i2** compiler flag (UNIX), the **noi4** qualifier (VMS), or the /integer_size:16 or /4I2 (Windows).

The Ingres **money** type is represented as **real*8**, an 8-byte real value.

Runtime Character and Varchar Type Conversion

Automatic conversion occurs between Ingres character string values and Fortran fixed-length character variables. String-valued Ingres objects that can interact with character string variables are:

- Ingres names, such as form and column names
- Database columns of type **character**
- Database columns of type **varchar**
- Form fields of type **character**
- Database columns of type **long varchar**

Several considerations apply when dealing with character string conversions, both to and from Ingres.

The conversion of Fortran character variables used to represent Ingres names is simple: trailing blanks are truncated from the variables, because the blanks make no sense in that context. For example, the string constants "empform " and "empform" refer to the same form.

The conversion of other Ingres objects is a bit more complex. First, the storage of character data in Ingres differs according to whether the medium of storage is a database column of type **character**, a database column of type **varchar**, or a **character** form field. Ingres pads columns of type **character** with blanks to their declared length. Conversely, it does not add blanks to the data in columns of type **varchar** or **long varchar**, or in form fields.

Second, the Fortran convention is to blank-pad fixed-length character strings. For example, the character string "abc" is stored in a Fortran **character*5** variable as the string "abc " followed by two blanks.

When character data is retrieved from a database column or form field into a Fortran character variable and the variable is longer than the value being retrieved, the variable is padded with blanks. If the variable is shorter than the value being retrieved, the value is truncated. You should always ensure that the variable is at least as long as the column or field, in order to avoid truncation of data.

When inserting character data into an Ingres database column or form field from a Fortran variable, note the following conventions:

- When you insert data from a Fortran variable into a database column of type **character** and the column is longer than the variable, the column is padded with blanks. If the column is shorter than the variable, the data is truncated to the length of the column.

- When you insert data from a Fortran variable into a database column of type **long varchar** or **varchar** and the column is longer than the variable, no padding of the column takes place. Furthermore, by default, all trailing blanks in the data are truncated before the data is inserted into the **varchar** column. For example, when a string "abc" stored in a Fortran **character*5** variable as "abc " (see above) is inserted into the **varchar** column, the two trailing blanks are removed and only the string "abc" is stored in the database column. To retain such trailing blanks, you can use the Ingres **notrim** function. It has the following syntax:

  notrim(:*charvar***)**

  where *charvar* is a character string variable. An example demonstrating this feature follows later. If the **varchar** column is shorter than the variable, the data is truncated to the length of the column.

- When you insert data from a Fortran variable into a **character** form field and the field is longer than the variable, no padding of the field takes place. In addition, all trailing blanks in the data are truncated before the data is inserted into the field. If the field is shorter than the data (even after all trailing blanks have been truncated), the data is truncated to the length of the field.

When comparing character data in a database column with character data in a Fortran variable, note the following. When comparing data in **character** or **varchar** database columns with data in a character variable, trailing blanks are ignored. Initial and embedded blanks are significant. To retain the significance of the trailing blanks in the comparison, you can use the **notrim** function, as shown in the following example.

Caution: As just described, the conversion of character string data between Ingres objects and Fortran variables often involves the trimming or padding of trailing blanks, with resultant change to the data. If trailing blanks have significance in your application, give careful consideration to the effect of any data conversion.

The Ingres **date** data type is represented as a 25-byte character string.

The following program fragment demonstrates the **notrim** function and the truncation rules previously explained:

```
      exec sql include sqlca
      exec sql begin declare section
           exec sql declare varychar table
     1    (row integer,
C Note the vchar type
     2     data vchar(10))

           integer*2   row
           character*7 data
      exec sql end declare section

C The variable data holds "abc" followed by 4 blanks
      data = 'abc    '
```

```
C The following INSERT adds the string "abc"
C (blanks truncated)
      exec sql insert into varychar (row, data)
     1   values (1, :data)

C
C This statement adds the string "abc ", with 4 trailing
C blanks left intact by using the NOTRIM function
C
      exec sql insert into varychar (row, data)
     1   values (2, notrim(:data))

C
C This SELECT will retrieve the second row, because the
C NOTRIM function leaves trailing blanks in the "data"
C variable for the comparison with Ingres
C vchar data.
      exec sql select row
     1   into :row
     2   from varychar
     3   where data = notrim(:data)
      print *, 'row found = ', row
```

The SQL Communications Area

The Include SQLCA Statement

You should issue the **include sqlca** statement in your main program module and in each subprogram of your Fortran file that includes Embedded SQL statements. If the file is composed of one main program and a few subprograms **include sqlca** should be the first Embedded SQL statement in each of the program units. For example:

```
program EmpPrc
            exec sql include sqlca
            ...
end

subroutine EmpSub

            exec sql include sqlca
            ...
end

integer function EmpFun

            exec sql include sqlca
            ...
end
```

The **include sqlca** statement instructs the preprocessor to generate code to call Ingres runtime libraries. It generates a Fortran **include** statement to make all the generated calls acceptable to the compiler.

Regardless of whether you intend to use the SQLCA for error handling, you *must* issue an **include sqlca** statement in each program unit containing Embedded SQL statements; if you do not, the Fortran compiler can complain about undeclared functions. Furthermore, the program aborts at runtime because program memory is overwritten. This occurs because, without explicit declaration of the SQLCA by means of the **include sqlca** statement, the Fortran compiler implicitly declares all references (including preprocessor-generated references) to the SQLCA as type **real**. Therefore, to help detect runtime errors due to missing **include sqlca** statements, you may want either to include the Fortran **implicit undefined** statement (UNIX) or **implicit none** statement (VMS and Windows) in each program unit, or to use the **-u** flag (UNIX), qualifier **warnings=declarations** (VMS), or **/warn:declarations** or **/4Yd** (Windows) with the compiler command. By doing so, you can ensure that the compiler generates a warning upon encountering a reference to an undeclared SQLCA.

Contents of the SQLCA

One of the results of issuing the **include sqlca** statement is the declaration of the SQLCA (SQL Communications Area), which you can use for error handling in the context of database statements. As mentioned above, you should issue the statement in your main program and in each subprogram that contains Embedded SQL statements. The declaration for the SQLCA is:

```
     character*8    sqlcai
     integer*4      sqlcab
     integer*4      sqlcod
     integer*2      sqltxl
     character*70   sqltxt
     character*8    sqlerp
     integer*4      sqlerr(6)
     character*1    sqlwrn(0:7)
     character*8    sqlext
 common /sqlca/ sqlcai, sqlcab, sqlcod, sqltxl, sqltxt,
1           sqlerp, sqlerr, sqlwrn, sqlext
```

This definition varies from the more standard definition of some other implementations. Also, because the names of the SQLCA fields conform to the names given in other implementations of Embedded SQL/Fortran, they are different from those mentioned in the *SQL Reference Guide*. The names of the fields most commonly used are **sqlcod** and **sqlerr**. These fields are equivalent to the fields **sqlcode** and **sqlerrd** described in the *SQL Reference Guide*. For a full description of all the SQLCA fields, see that guide.

The SQLCA is initialized at load time. The **sqlcai** and **sqlcab** fields are initialized to the string “SQLCA” and the constant 136, respectively.

The preprocessor is not aware of the SQLCA declaration. Therefore, you cannot use SQLCA fields in an Embedded SQL statement. For example, the following statement, attempting to **insert** the error code **sqlcod** into a table, generates an error:

```
C This statement is illegal
        exec sql insert into employee (eno)
     1 values (:sqlcod)
```

All modules (written in Fortran or other Embedded SQL languages) share the same SQLCA.

Using the SQLCA for Error Handling

User-defined error, message and dbevent handlers offer the most flexibility for handling errors, database procedure messages, and database events. For more information, see Advanced Processing in this chapter.

However, you can do error handling with the SQLCA implicitly by using **whenever** statements or explicitly by checking the contents of the SQLCA fields **sqlcod**, **sqlerr(3)**, and **sqlwrn(0).**

Error Handling with the Whenever Statement

The syntax of the **whenever** statement is as follows:

exec sql whenever *condition action*

where *condition* is **dbevent**, **sqlwarning**, **sqlerror**, **sqlmessage**, or **not found**, and *action* is **continue**, **stop**, **goto** a label, or **call** a Fortran subroutine. For a detailed description of this statement, see the *SQL Reference Guide.*

In Fortran, all subroutine names must be legal Fortran identifiers, beginning with an alphabetic character. In VMS, you can also use an underscore. If the subroutine name is an Embedded SQL reserved word, specify it in quotes. All labels specified in a **whenever goto** action must be legal statement numbers. Note that Embedded SQL reserves statement numbers 7000 through 12000. The label targeted by the **goto** action must be within the scope of all subsequent Embedded SQL statements until another **whenever** statement is encountered for the same action. This is necessary because the preprocessor may generate the Fortran statement:

if (*condition***) goto** *label*

after an Embedded SQL statement. If the label is an invalid statement number, the Fortran compiler generates an error.

The same scope rules apply to subroutine names used with the **call** action as to label numbers used with the **goto** action. However, the reserved subroutine name **sqlprint**, which prints errors or database procedure messages and then continues, is always within the scope of the program.

When a **whenever** statement specifies a **call** as the action, the target subroutine is called, and after its execution, control returns to the statement following the statement that caused the subroutine to be called. Consequently, after handling the **whenever** condition in the called subroutine, you may want to take some action, instead of merely issuing a Fortran **return** statement. The Fortran **return** statement causes the program to continue execution with the statement following the Embedded SQL statement that generated the error.

The following example demonstrates the use of the **whenever** statements in the context of printing some values from the Employee table. The comments do not relate to the program but to the use of error handling.

```
C
C Main error handling program
C

              program DbTest

              exec sql include sqlca

              exec sql begin declare section
                                  integer*2     eno
                                  character*20  ename
                                  integer*1     eage
              exec sql end declare section

              exec sql declare empcsr cursor for
             1   select eno, ename, age
             2   from employee

C An error when opening the "personnel" database will
C cause the error to be printed and the program
C to abort.

              exec sql whenever sqlerror stop
              exec sql connect personnel

C Errors from here on will cause the program to clean up

              exec sql whenever sqlerror call ClnUp
              exec sql open empcsr
              print *, 'Some values from the "employee" table'

C When no more rows are fetched, close the cursor

              exec sql whenever not found goto 200

C The last executable Embedded SQL statement was
C an OPEN, so we know that the value of "sqlcod"
C cannot be SQLERROR or NOT FOUND.

C The following loop is broken by NOT FOUND

exec sql fetch empcsr
```

```
             1   into :eno, :ename, :age

C This "print" does not execute after the previous
C FETCH returns the NOT FOUND condition.

              print *, eno, ename, age
              if (sqlcod .eq. 0) goto 100

C From this point in the file onwards, ignore all
C errors.
C Also turn off the NOT FOUND condition, for
C consistency.

              exec sql whenever sqlerror continue
              exec sql whenever not found continue

    200   exec sql close empcsr

              exec sql disconnect

              end

C
C ClnUp: Error handling subroutine (print error
C and disconnect).
C

              subroutine ClnUp

              exec sql include sqlca

              exec sql begin declare section
                        character*100 errmsg
              exec sql end declare section

              exec sql inquire_sql (:errmsg=ERRORTEXT)
              print *, 'Aborting because of error'
              print *, errmsg

              exec sql disconnect

C Do not return to DbTest

              stop

              end
```

Whenever Goto Action in Embedded SQL Blocks

An Embedded SQL block-structured statement is a statement delimited by the words **begin** and **end**. For example, the **select** loop and **unloadtable** loops are block-structured statements. You can terminate these statements only by the methods specified for their termination in the *SQL Reference Guide.* For example, the **select** loop is terminated either when all the rows in the database result table are processed or by an **endselect** statement. The **unloadtable** loop is terminated either when all the rows in the forms table field are processed or by an **endloop** statement.

Therefore, if you use a **whenever** statement with the **goto** action in an SQL block, you must avoid going to a label outside the block. Such a **goto** causes the block to be terminated without issuing the runtime calls necessary to clean up the information that controls the loop. (For the same reason, you must not issue a Fortran **return** or **goto** statement that causes control to leave or enter the middle of an SQL block.) The target label of the **whenever goto** statement should be a label in the block. If however, it is a label for a block of code that cleanly exits the program, you do not need to take such precautions.

The above information does not apply to error handling for database statements issued outside an SQL block or to explicit hard-coded error handling. For an example of hard-coded error handling, see The Table Editor Table Field Application in this chapter.

Explicit Error Handling

The program can also handle errors by inspecting values of the SQLCA at various points. For further details, see the *SQL Reference Guide*.

The following example is functionally the same as the previous example, except that the error handling is hard-coded in Fortran statements:

```
C
C Main error handling program
C

              program DbTest

              exec sql include sqlca

              exec sql begin declare section
                        integer*2       eno
                        character*20    ename
                        integer*1       eage
              exec sql end declare section

              exec sql declare empcsr cursor for
           1   select eno, ename, age
           2   from employee

C Exit if database cannot be opened

              exec sql connect personnel

              if (sqlcod .lt. 0) then
                        print *, 'Cannot access database'
                        stop
              end if

C Error if cannot open cursor

              exec sql open empcsr
              if (sqlcod .lt. 0) call ClnUp('OPEN "empcsr"')

              print *, 'Some values from the "employee" table'
```

```
C The last executable Embedded SQL statement was
C an OPEN, so we know that the value of "sqlcod"
C cannot be SQLERROR or NOT FOUND.

C The following loop is broken by NOT FOUND
C (condition 100) or an
C error

100         exec sql fetch empcsr
            1           into :eno, :ename, :age

                     if (sqlcod .lt. 0) then
                                       call ClnUp('FETCH "empcsr"')

C Do not print the last values twice

                     else if (sqlcod .ne. 100) then
                                       print *, eno, ename, age
                     end if

            if (sqlcod .eq. 0) goto 100

            exec sql close empcsr
            exec sql disconnect

      end

C
C ClnUp: Error handling subroutine
C (print error and disconnect).
C

            subroutine ClnUp(reason)

            exec sql include sqlca

            exec sql begin declare section
                    character*(50) reason
                    character*100 errmsg
            exec sql end declare section

            print *, 'Aborting because of error in ', reason
            exec sql inquire_sql (:errmsg=ERRORTEXT)
            print *, errmsg

            exec sql disconnect

C Do not return to DbTest

            stop

            end
```

Determining the Number of Affected Rows

The SQLCA variable **sqlerr(3)** indicates how many rows were affected by the last row-affecting statement. (Note that in the *SQL Reference Guide,* this field is called **sqlerrd(3)**.) The following program fragment, which deletes all employees whose employee numbers are greater than a given number, demonstrates how to use **sqlerr**:

```
      subroutine DelRow(lbnum)

          exec sql include sqlca

          exec sql begin declare section
                     integer lbnum
      exec sql end declare section

      exec sql delete from employee
     1    where eno > :lbnum

C Print the number of employees deleted

          print *, sqlerr(3), 'row(s) were deleted.'

          end
```

Using the SQLSTATE Variable

You can use the **SQLSTATE** variable in an ESQL/Fortran program to return status information about the last SQL statement that was executed. **SQLSTATE** must be declared in a declaration section and must be in uppercase. Also, it is valid across all sessions, so you only need to declare one **SQLSTATE** per application.

To declare this variable, use:

```
character*5 SQLSTATE
```

or :

```
character*5 SQLSTA
```

Dynamic Programming for Fortran

Ingres provides Dynamic SQL and Dynamic FRS to allow you to write generic programs. Dynamic SQL allows a program to build and execute SQL statements at runtime. For example, an application can include an expert mode in which the runtime user can type in select queries and browse the results at the terminal. Dynamic FRS allows a program to interact with any form at runtime. For example, an application can load in any form, allowing the runtime user to retrieve new data from the form and insert it into the database.

The Dynamic SQL and Dynamic FRS statements are described in the *SQL Reference Guide* and the *Forms-based Application Development Tools User Guide*. This section discusses the Fortran-dependent issues of dynamic programming. For a complete example of using Dynamic SQL to write an SQL Terminal Monitor application, see The SQL Terminal Monitor Application in this chapter. For an example of using both Dynamic SQL and Dynamic FRS to browse and update a database using any form, see A Dynamic SQL/Forms Database Browser in this chapter.

The VMS examples in this section make use of the VMS extensions to the Fortran language. Because the SQLDA is a structure, the UNIX examples in this section apply only to those F77 Fortran compilers that have been extended to include the support of the structures.

The SQLDA Structure

You use the SQLDA (SQL Descriptor Area) to pass type and size information about an SQL statement, an Ingres form, or an Ingres table field, between Ingres and your program.

To use the SQLDA, issue the **include sqlda** statement in each subprogram of your source file that references the SQLDA. The **include sqlda** statement generates a Fortran include directive to a file that defines the SQLDA structure type. The file does *not* declare an SQLDA variable; your program must declare a variable of the specified type. You can also code this structure variable directly instead of using the **include sqlda** statement. You can choose any name for the structure.

The definition of the SQLDA (as specified in the **include** file) is:

UNIX

```
C
C Single element of SQLDA variable
C
          structure /IISQLVAR/
                        integer*2     sqltype
                        integer*2     sqllen
                        integer*4     sqldata
                        integer*4     sqlind
                        structure /IISQLNAME/ sqlname
                                integer*2           sqlname1
                                character*34        sqlnamec
                        end structure
          end structure

C
C Maximum number of columns returned from Ingres
C
          parameter (IISQ_MAX_COLS = 1024)
```

```
C
C IISQLDA - SQLDA with maximum number of entries
C for variables.
C
          structure /IISQLDA/
                        character*8 sqldaid
                        integer*4   sqldabc
                        integer*2   sqln
                        integer*2   sqld
                        record /IISQLVAR/ sqlvar(IISQ_MAX_COLS)
          end structure

    structure /IISQLHDLR/
C     Optional argument to pass through
            integer*4   sqlarg
C     user-defined datahandler function
            integer*4   sqlhdlr
    end structure

C
C Allocation sizes
C
          parameter (IISQDA_HEAD_SIZE = 16,
   1                 IISQDA_VAR_SIZE = 48)
C
C Type and length codes
C
C

          parameter (IISQ_DTE_TYPE = 3,
   1      IISQ_MNY_TYPE  = 5,
   2      IISQ_DEC_TYPE  = 10,
   3      IISQ_CHA_TYPE  = 20,
   4      IISQ_VCH_TYPE  = 21,
   5      IISQ_LVCH_TYPE = 22,
   6      IISQ_INT_TYPE  = 30,
   7      IISQ_FLT_TYPE  = 31,
   8      IISQ_OBJ_TYPE  = 45,
   9      IISQ_HDLR_TYPE = 46,
   1      IISQ_TBL_TYPE  = 52,
   2      IISQ_DTE_LEN   = 25)
    parameter (IISQ_LVCH_TYPE = 22,
   1          IISQ_HDLR_TYPE = 46)
```

VMS

```
    structure /IISQLVAR/ ! Single SQLDA variable
                integer*2 sqltype
                integer*2 sqllen
                integer*4 sqldata ! Address of any type
                integer*4 sqlind  ! Address of 2-byte integer
                structure /IISQLNAME/ sqlname
                      integer*2 sqlname1
                      character*34 sqlnamec
                end structure
    end structure

    parameter IISQ_MAX_COLS = 1024 ! Maximum number of
C                                    columns
```

```
    structure /IISQLDA/
                character*8 sqldaid
                integer*4 sqldabc
                integer*2 sqln
                integer*2 sqld
                record /IISQLVAR/ sqlvar(IISQ_MAX_COLS)
    end structure

    structure /IISQLHDR/
                integer*4   sqlarg  ! Optional argument to pass
                integer*4   sqlhdlr ! User-defined datahandler fn
    end structure

! Type codes
    parameter IISQ_DTE_TYPE = 3, ! Date - Output
   1    IISQ_MNY_TYPE = 5,  ! Money - Output
   2    IISQ_DEC_TYPE = 10, ! Decimal - Output
   3    IISQ_CHA_TYPE = 20, ! Char - Input, Output
   4    IISQ_VCH_TYPE = 21, ! Varchar - Input, Output
   5    IISQ_LVCH_TYPE= 22, ! Long Varchar - Input,Output
   6    IISQ_INT_TYPE = 30, ! Integer - Input, Output
   7    IISQ_FLT_TYPE = 31, ! Float - Input, Output
   8    IISQ_OBJ_TYPE = 45, ! 4GL Object: Output
   9    IISQ_HDLR_TYPE= 46, ! IISQLHDLR: Datahandler
   1    IISQ_TBL_TYPE = 52, ! Table Field - Output
   2    IISQ_DTE_LEN = 25,  ! Date length

! Allocation sizes
    parameter IISQDA_VAR_SIZE = 16,
   1     IISQDA_VAR = 48

    parameter IISQ_LVCH_TYPE = 22,
   1         IISQ_HDLR_TYPE = 46
```

Windows

```
    structure /IISQLVAR/
        integer*2       sqltype
        integer*2       sqllen
        integer*4       sqldata         ! Address of any type
        integer*4       sqlind          ! Address of 2-byte integer
        structure /IISQLNAME/ sqlname
            integer*2      sqlnamel
            character*34  sqlnamec
        end structure
    end structure

C
C IISQ_MAX_COLS - Maximum number of columns returned from INGRES
C
    parameter IISQ_MAX_COLS = 1024

C
C IISQLDA - SQLDA with maximum number of entries for variables.
C
    structure /IISQLDA/
        character*8             sqldaid
        integer*4               sqldabc
        integer*2               sqln
        integer*2               sqld
        record /IISQLVAR/       sqlvar(IISQ_MAX_COLS)
    end structure

C
C IISQLHDLR - Structure type with function pointer and function argument
C         for the DATAHANDLER.
```

```
C
    structure /IISQLHDLR/
        integer*4               sqlarg
        integer*4               sqlhdlr
    end structure


C
C Allocation sizes - When allocating an SQLDA for the size use:
C       IISQDA_HEAD_SIZE + (N * IISQDA_VAR_SIZE)
C
    parameter IISQDA_HEAD_SIZE = 16,
    1     IISQDA_VAR_SIZE  = 48

C
C Type and Length Codes
C
    parameter IISQ_DTE_TYPE = 3, ! Date - Output
    1     IISQ_MNY_TYPE = 5,     ! Money - Output
    2     IISQ_DEC_TYPE = 10,    ! Decimal - Output
    3     IISQ_CHA_TYPE  = 20,   ! Char - Input, Output
    4     IISQ_VCH_TYPE  = 21,   ! Varchar - Input, Output
    5     IISQ_INT_TYPE  = 30,   ! Integer - Input, Output
    6     IISQ_FLT_TYPE = 31,    ! Float - Input, Output
    7     IISQ_TBL_TYPE = 52,    ! Table field - Output
    8     IISQ_DTE_LEN  = 25     ! Date length
    parameter IISQ_LVCH_TYPE = 22        ! Long varchar
    parameter IISQ_LBIT_TYPE = 16        ! Long bit
    parameter IISQ_HDLR_TYPE = 46        ! Datahandler
    parameter IISQ_BYTE_TYPE = 23        ! Byte - Input, Output
    parameter IISQ_VBYTE_TYPE = 24       ! Byte Varying - Input, Output
    parameter IISQ_LBYTE_TYPE = 25       ! Long Byte - Output
    parameter IISQ_OBJ_TYPE = 45         ! Object - Output
```

Structure Definition and Usage Notes:

- The structure type definition of the SQLDA is called IISQLDA. This is done so that an SQLDA variable may be called "SQLDA" without causing a compile-time conflict.
- The **sqlvar** array is an array of IISQ_MAX_COLS (1024) elements. If a variable of type IISQLDA is declared, the program will have a variable of IISQ_MAX_COLS elements.
- The **sqlvar** array begins at subscript 1.
- If your program defines its own SQLDA type, you must confirm that the structure layout is identical to that of the IISQLDA structure type, although you can declare a different number of **sqlvar** elements.
- The nested structure **sqlname** is a varying length character string consisting of a length and data area. The **sqlnamec** field contains the name of a result field or column after the **describe** (or **prepare into**) statement. The length of the name is specified by **sqlnamel**. The characters in the **sqlnamec** field are blank padded. The **sqlname** structure can also be set by a program using Dynamic FRS. The program is not required to pad **sqlname** with blanks. (See Setting SQLNAME for Dynamic FRS in this chapter.)

- The list of type codes represents the types that are returned by the **describe** statement, and the types used by the program when using an SQLDA to retrieve or set data. The type code IISQ_TBL_TYPE indicates a table field and is set by the FRS when describing a form that contains a table field.

Declaring an SQLDA Variable

Once the SQLDA definition has been included (or hard-coded) the program can declare an SQLDA variable. This variable must be declared outside a **declare section**, as the preprocessor does not understand the special meaning of the SQLDA. When the variable is used, the preprocessor will accept any object name and assume that the variable refers to a legal SQLDA.

If a program requires a statically declared SQLDA with the same number of **sqlvar** variables as the IISQLDA type, it can accomplish this as in the following example:

```
            exec sql include SQLDA
            record /IISQLDA/ sqlda
C Set the size
            sqlda.sqln = IISQ_MAX_COLS

. . .

            exec sql describe s1 into :sqlda
```

Recall that you must confirm that the SQLDA object being used is a valid SQLDA.

If a program requires a statically declared SQLDA with a *different* number of variables (not IISQ_MAX_COLS), it can declare its own type. For example:

```
    structure /MYSQLDA/
            character*8       sqldaid
            integer*4         sqldabc
            integer*2         sqln
            integer*2         sqld
            record /IISQLVAR/ sqlvar(10)
end structure
```

In the above declaration, the names of the structure components do not need to be the same as those of the IISQLDA structure.

Using the SQLVAR

The *SQL Reference Guide* discusses the legal values of the **sqlvar** array. The **describe** and **prepare into** statements assign type, length, and name information into the SQLDA. This information refers to the result columns of a prepared **select** statement, the fields of a form, or the columns of a table field. When the program uses the SQLDA to retrieve or set Ingres data, it must assign the type and length information that now refers to the variables being pointed at by the SQLDA.

Fortran Variable Type Codes

The type codes listed below are the types that describe Ingres result fields and columns. For example, the SQL types **date**, **long varchar, money,** and **decimal** do not describe program variables, but rather data types that are compatible with Fortran types **character** and **real*8**. When these types are returned by the describe statement, the type code must be changed to a compatible SQL/Fortran type.

The following table describes the type codes to use with Fortran variables that will be *pointed* at by the **sqldata** pointers.

Embedded SQL/Fortran Type Codes

Embedded SQL/Fortran Type Codes (sqltype)	Length (sqllen)	Fortran Variable Type
IISQ_INT_TYPE	1	**byte**
IISQ_INT_TYPE	2	**integer*2**
IISQ_INT_TYPE	4	**integer*4**
IISQ_FLT_TYPE	4	**real*4**
IISQ_FLT_TYPE	8	**real*8**
IISQ_CHA_TYPE	LEN	**character*LEN**
IISQ_VCH_TYPE	LEN	**character*LEN**
IISQ_HDLR_TYPE	0	**IISQHDLR**

To retrieve a decimal value from the DBMS, you must use a **float** because Fortran does not have decimal variables.

Nullable data types (those variables that are associated with a null indicator) are specified by assigning the negative of the type code to the **sqltype** field. If the type is negative, a null indicator must be pointed at by the **sqlind** field.

Character data and the SQLDA have exactly the same rules as character data in regular Embedded SQL statements.

Pointing at Fortran Variables

In order to fill an element of the **sqlvar** array, you must set the type information and assign a valid address to **sqldata**. The address must be that of a legal variable address.

For example, the following fragment sets the type information of and points at a 4-byte integer variable, an 8-byte nullable floating-point variable, and an **sqllen**-specified character substring. The following example demonstrates how a program can maintain a pool of available variables, such as large arrays of the few different typed variables, and a large string space. The next available spot is chosen from the pool:

UNIX

Note: On UNIX the Fortran function "loc" may be provided. If your UNIX Fortran library does not contain a function for obtaining the address of variables, the Ingres functions "IInum" and "IIsadr" can be used to return the address of number and character strings respectively.

It has the following usage:

sqlda.sqlvar(i).sqldata = IInum(current_integer)
sqlda.sqlvar(i).sqldata = IIsadr (current_string)

```
C Assume sqlda has been declared

           sqlda.sqlvar(1).sqltype = IISQ_INT_TYPE
           sqlda.sqlvar(1).sqllen = 4
           sqlda.sqlvar(1).sqldata
            1             = loc(integer_array(current_integer))
           sqlda.sqlvar(1).sqlind = 0
           current_integer = current_integer + 1

           sqlda.sqlvar(2).sqltype = -IISQ_FLT_TYPE
           sqlda.sqlvar(2).sqllen = 8
           sqlda.sqlvar(2).sqldata
            1                = loc(real_array(current_real))
           sqlda.sqlvar(2).sqlind
            1                = loc(indicator_array(current_ind))
           current_real = current_real + 1
           current_ind = current_ind + 1
C
C SQLLEN has been assigned by DESCRIBE to be the
C length of a specific result column. This length
C is used to pick off
C a substring out of a large string space.
C
           needlen = sqlda.sqlvar(3).sqllen
           sqlda.sqlvar(3).sqltype = IISQ_CHA_TYPE
           sqlda.sqlvar(3).sqldata =
            1            loc(large_string(current_string:needlen))
           sqlda.sqlvar(3).sqlind = 0
           current_string = current_string + needlen
```

VMS

Note: On VMS the Fortran function "%loc" is used to access the address of variables.

```
! Assume sqlda has been declared

    sqlda.sqlvar(1).sqltype = IISQ_INT_TYPE
    sqlda.sqlvar(1).sqllen = 4
    sqlda.sqlvar(1).sqldata
         1             = %loc(integer_array(current_integer))
    sqlda.sqlvar(1).sqlind = 0
    current_integer = current_integer + 1

    sqlda.sqlvar(2).sqltype = -IISQ_FLT_TYPE
    sqlda.sqlvar(2).sqllen = 8
    sqlda.sqlvar(2).sqldata
   1                = %loc(real_array(current_real))
    sqlda.sqlvar(2).sqlind
         1             = %loc(indicator_array(current_ind))
    current_integer = current_real + 1
    current_integer = current_ind + 1
!
! SQLLEN has been assigned by DESCRIBE to be the length
! of a specific result column. This length is used to
! pick off a substring out of a large string space.
!
    needlen = sqlda.sqlvar(3).sqllen
    sqlda.sqlvar(3).sqltype = IISQ_CHA_TYPE
    sqlda.sqlvar(3).sqldata
         1          = %loc(large_string(current_string:needlen))
    sqlda.sqlvar(3).sqlind = 0
    current_string = current_string + needlen
```

Windows

Note: On Windows the "loc" intrinsic function (or the "%loc" built-in function) is used to access the address of variables.

```
C Assume sqlda has been declared

    sqlda.sqlvar(1).sqltype = IISQ_INT_TYPE
    sqlda.sqlvar(1).sqllen = 4
    sqlda.sqlvar(1).sqldata
         1             = %loc(integer_array(current_integer))
    sqlda.sqlvar(1).sqlind = 0
    current_integer = current_integer + 1

    sqlda.sqlvar(2).sqltype = -IISQ_FLT_TYPE
    sqlda.sqlvar(2).sqllen = 8
    sqlda.sqlvar(2).sqldata
   1                = %loc(real_array(current_real))
    sqlda.sqlvar(2).sqlind
         1             = %loc(indicator_array(current_ind))
    current_integer = current_real + 1
    current_integer = current_ind + 1
C
C SQLLEN has been assigned by DESCRIBE to be the length
C of a specific result column. This length is used to
C pick off a substring out of a large string space.
C
    needlen = sqlda.sqlvar(3).sqllen
    sqlda.sqlvar(3).sqltype = IISQ_CHA_TYPE
    sqlda.sqlvar(3).sqldata
         1          = %loc(large_string(current_string:needlen))
    sqlda.sqlvar(3).sqlind = 0
    current_string = current_string + needlen
```

You may also set the SQLVAR to point to a datahandler for large object columns. For details, see Advanced Processing in this chapter.

Setting SQLNAME for Dynamic FRS

A few extra steps are required when you use the **sqlvar** with Dynamic FRS statements. These extra steps relate to the differences between Dynamic FRS and Dynamic SQL and are described in the *SQL Reference Guide* and the *Forms-based Application Development Tools User Guide*.

When using the SQLDA in a forms input or output **using** clause, set the **sqlname** to a valid field or column name. If this name was set by a previous **describe** statement, it must be retained or reset by the program. If the name refers to a hidden table field column, the program must set **sqlname** directly. If your program sets **sqlname** directly, it must also set **sqlnamel** and **sqlnamec**. The name portion does not need to be padded with blanks.

For example, a dynamically named table field has been described, and the application always initializes any table field with a hidden 6-byte character column called "rowid." The code used to retrieve a row from the table field including the hidden column and **_state** variable would have to construct the two named columns:

UNIX

```
...
     character*6 rowid
         integer*4 rowstate

...

     exec frs describe table :formname :tablename
    1     into :sqlda

...

     sqlda.sqld = sqlda.sqld + 1
         col_num = sqlda.sqld

C Set up to retrieve rowid
         sqlda.sqlvar(col_num).sqltype = IISQ_CHA_TYPE
         sqlda.sqlvar(col_num).sqllen = 6
         sqlda.sqlvar(col_num).sqldata = loc(rowid)
         sqlda.sqlvar(col_num).sqlind = 0
         sqlda.sqlvar(col_num).sqlname.sqlnamel = 5
    sqlda.sqlvar(col_num).sqlname.sqlnamec(1:5) = 'rowid'

         sqlda.sqld = sqlda.sqld + 1
         col_num = sqlda.sqld
```

```
C Set up to retrieve _STATE
          sqlda.sqlvar(col_num).sqltype = IISQ_INT_TYPE
          sqlda.sqlvar(col_num).sqllen = 4
          sqlda.sqlvar(col_num).sqldata = loc(rowstate)
          sqlda.sqlvar(col_num).sqlind = 0
          sqlda.sqlvar(col_num).sqlname.sqlnamel = 6
          sqlda.sqlvar(col_num).sqlname.sqlnamec(1:6)
    1                                         = '_state'

...

     exec frs getrow :formname :tablename using
    1       descriptor :sqlda
```

VMS

```
...
     character*6 rowid
     integer*4 rowstate

...

     exec frs describe table :formname :tablename
    1     into :sqlda

...

     sqlda.sqld = sqlda.sqld + 1
     col_num = sqlda.sqld

! Set up to retrieve rowid
     sqlda.sqlvar(col_num).sqltype = IISQ_CHA_TYPE
     sqlda.sqlvar(col_num).sqllen = 6
     sqlda.sqlvar(col_num).sqldata = %loc(rowid)
     sqlda.sqlvar(col_num).sqlind = 0
     sqlda.sqlvar(col_num).sqlname.sqlnamel = 5
     sqlda.sqlvar(col_num).sqlname.sqlnamec(1:5) = 'rowid'
     sqlda.sqld = sqlda.sqld + 1
     col_num = sqlda.sqld

! Set up to retrieve _STATE
     sqlda.sqlvar(col_num).sqltype = IISQ_INT_TYPE
     sqlda.sqlvar(col_num).sqllen = 4
     sqlda.sqlvar(col_num).sqldata = %loc(rowstate)
     sqlda.sqlvar(col_num).sqlind = 0
     sqlda.sqlvar(col_num).sqlname.sqlnamel = 6
     sqlda.sqlvar(col_num).sqlname.sqlnamec(1:6)
    1                                         = '_state'

...

     exec frs getrow :formname :tablename using
    1        descriptor:sqlda
```

Windows

```
...
     character*6 rowid
         integer*4 rowstate

...

     exec frs describe table :formname :tablename
    1     into :sqlda
```

```
...

     sqlda.sqld = sqlda.sqld + 1
         col_num = sqlda.sqld

C Set up to retrieve rowid
         sqlda.sqlvar(col_num).sqltype = IISQ_CHA_TYPE
         sqlda.sqlvar(col_num).sqllen = 6
         sqlda.sqlvar(col_num).sqldata = loc(rowid)
         sqlda.sqlvar(col_num).sqlind = 0
         sqlda.sqlvar(col_num).sqlname.sqlnamel = 5
    sqlda.sqlvar(col_num).sqlname.sqlnamec(1:5) = 'rowid'

         sqlda.sqld = sqlda.sqld + 1
         col_num = sqlda.sqld

C Set up to retrieve _STATE
         sqlda.sqlvar(col_num).sqltype = IISQ_INT_TYPE
         sqlda.sqlvar(col_num).sqllen = 4
         sqlda.sqlvar(col_num).sqldata = loc(rowstate)
         sqlda.sqlvar(col_num).sqlind = 0
         sqlda.sqlvar(col_num).sqlname.sqlnamel = 6
         sqlda.sqlvar(col_num).sqlname.sqlnamec(1:6)
    1                                       = '_state'

...

     exec frs getrow :formname :tablename using
    1       descriptor :sqlda
```

Advanced Processing

This section describes user-defined handlers. It includes information about user-defined error, dbevent, and message handlers as well as data handlers for large objects.

User-Defined Error, DBevent, and Message Handlers

You can use user-defined handlers to capture errors, messages, or events during the processing of a database statement. Use these handlers instead of the **sql whenever** statements with the SQLCA when you want to do the following:

- Capture more than one error message on a single database statement
- Capture more than one message from database procedures fired by rules
- Trap errors, events, and messages as the DBMS raises them
  If an event is raised when an error occurs during query execution, the WHENEVER mechanism detects only the error and defers acting on the event until the next database statement is executed.

User-defined handlers offer you flexibility. If, for example, you want to trap an error, you can code a user-defined handler to issue an **inquire_sql** to get the error number and error text of the current error. You can then switch sessions and log the error to a table in another session; however, you must switch back to the session from which the handler was called before returning from the handler. When the user handler returns, the original statement continues executing. User code in the handler cannot issue database statements for the session from which the handler was called.

The handler must be declared to return an integer. However, the preprocessor ignores the return value.

Syntax Notes:

The following syntax describes the three types of handlers:

```
exec sql set_sql (errorhandler   = error_routine|0)
exec sql set_sql (dbeventhandler = event_routine|0)
exec sql set_sql (messagehandler = message_routine|0)
```

- Errorhandler, dbeventhandler, and messagehandler denote a user-defined handler to capture errors, events, and database messages respectively, as follows:
  - error_routine is the name of the function the Ingres runtime system calls when an error occurs.
  - event_routine is the name of the function the Ingres runtime system calls when an event is raised. message_routine is the name of the function the Ingres runtime system calls whenever a database procedure generates a message.

  Errors that occur in the error handler itself do not cause the error handler to be re-invoked. You must use **inquire_sql** to handle or trap any errors that may occur in the handler.
- Unlike regular variables, the handler must not be declared in an ESQL declare section; therefore, do not use a colon before the handler argument. (However, you must declare the handler to the compiler.)
- If you specify a zero (0) instead of a name, the zero will unset the handler.

User-defined handlers are also described in the *SQL Reference Guide.*

Declaring and Defining User-Defined Handlers

The following example shows how to declare a handler for use in the **set_sql errorhandler** statement for ESQL/Fortran:

```
program TestProg
```

```
     exec sql include sqlca

     external error_func
     integer error_func

         exec sql connect dbname

     exec sql set_sql (errorhandler = error_func)
     ...
                  program code
     ...
end

integer function error_func

         exec sql include sqlca

         exec sql begin declare section
                integer errnum
         exec sql end declare section
         exec sql inquire_sql (:errnum = ERRORNO)
     write (*,60) errnum
60     format ('Errnum is ', I)
end
```

User-Defined Data Handlers for Large Objects

You can use user-defined datahandlers to transmit large object column values to or from the database a segment at a time. For more details on Large Objects, the **datahandler** clause, the **get data** statement, and the **put data** statement, see the *SQL Reference Guide* and the *Forms-based Application Development Tools User Guide*.

ESQL/Fortran Usage Notes

Use datahandlers in the following ways:

- The datahandler, and the datahandler argument, should not be declared in an ESQL declare section. Therefore do not use a colon before the datahandler or its argument.
- You must ensure that the datahandler argument is a valid Fortran variable address. ESQL will not do any syntax or datatype checking of the argument.
- The datahandler must be declared to return an integer. However, the actual return value will be ignored.

DATAHANDLERS and the SQLDA

You may specify a user-defined datahandler as an SQLVAR element of the SQLDA, to transmit large objects to or from the database. The "eqsqlda.h" file included via the **include sqlda** statement defines an IISQLHDLR type which may be used to specify a datahandler and its argument. It is defined:

```
structure /IISQLHDLR/
       integer*4 sqlarg  ! Optional argument to pass
       integer*4 sqlhdlr ! User-defined datahandler
end structure
```

The file does not declare an IISQLHDLR variable; the program must declare a variable of the specified type and set the values:

```
record /IISQLHDLR/   dathdlr
structure   /hdlr_arg/
     character*100    argstr
     integer          argint
end structure
record /hdlr_arg/ hdlarg

external Get_Handler()
integer  Get_Handler()
```

UNIX

```
dathdlr.sqlarg  = loc(hdlarg)
dathdlr.sqlhdlr = loc(Get_Handler)
```

VMS

```
     dathdlr.sqlarg  = %loc(hdlarg)
     dathdlr.sqlhdlr = %loc(Get_Handler)
```

Windows

```
 dathdlr.sqlarg  = %loc(hdlarg)
 dathdlr.sqlhdlr = %loc(Get_Handler)
```

The **sqltype**, **sqlind** and **sqldata** fields of the SQLVAR element of the SQLDA should then be set as follows:

```
/*
** assume sqlda is a pointer to a dynamically allocated
** SQLDA
*/
sqlda.sqlvar[i].sqltype = IISQ_HDLR_TYPE;
sqlda.sqlvar[i].sqlind  = loc(indvar)
sqlda.sqlvar[i].sqldata = loc(dathdlr)
```

Sample Programs

The programs in this section are examples of how to declare and use user-defined datahandlers in an ESQL/Fortran program. There are examples of a handler program, a put handler program, a get handler program and a dynamic SQL handler program.

Handler Program

This example assumes that the book table was created with the statement:

```
exec sql create table book (chapter_num integer,
      chapter_name char(50), chapter_text long varchar)
```

This program inserts a row into the book table using the data handler Put_Handler to transmit the value of column chapter_text from a text file to the database. Then it selects the column chapter_text from the table book using the data handler Get_Handler to retrieve the chapter_text column a segment at a time.

```
C main program
C ***************
                program handler

                exec sql include sqlca

C Do not declare the datahandlers nor the datahandler
C argument to the ESQL pre-processor.

                external Put_Handler
                integer  Put_Handler

                external Put_Handler
                integer  Get_Handler

      structure       /hdlr_arg/
        character*1000     argstr
        integer            argint
      end structure

            record /hdlr_arg/hdlarg

C Null indicator for datahandler must be declared
C to ESQL

            exec sql begin declare section
                                 integer*2 indvar
            integer*4 chapter_num
            exec sql end declare section

C INSERT a long varchar value chapter_text into the table book
C using the datahandler Put_Handler.
C The argument passed to the datahandler is the address of
C the record hdlarg.

            . . .

            exec sql insert into book (chapter_num, chapter_name,
           chapter_text)
     1     values (5, 'One Dark and Stormy Night',
     2                  Datahandler(Put_Handler(hdlarg)))
      . . .

C Select the column chapter_num and the long varchar column
C chapter_text from the table book.
C The Datahandler (Get_Handler) will be invoked for each non-null
C value of column chapter_text retrieved. For null values the
C indicator variable will be set to "-1" and the datahandler will
C not be called.
```

```
         ...
            exec sql select chapter_num, chapter_text into
    1       :chapter_num,
    2       datahandler(Get_Handler(hdlarg)):indvar from book

            exec sql begin
                                process row ...
            exec sql end

            . . .

            end
```

Put Handler

This example shows how to read the **long varchar** chapter_text from a text file and insert it into the database a segment at a time:

```
C Put_Handler
C ***********

         integer function Put_Handler(info)

         structure        /hdlr_arg/
                character*100     argstr
                integer*4         argint
         end structure
         record /hdlr_arg/ info

         exec sql begin declare section
                character*1000    segbuf
                integer*4         seglen
                integer*4         datend
         exec sql end declare section

            process information passed in via the info record ...
            open file ...


            datend = 0

             do while not end-of-file

            read segment of less than 1000 characters from file into segbuf . . .

                if end-of-file then
                     datend = 1

                end if

                exec sql put data (segment = :segbuf,
        1                 segmentlength = :seglen, dataend = :datend)
            end do
            . . .
            close file ...
            set info record to return appropriate values ...
            . . .
            Put_Handler = 0
            end
```

Get Handler

This example shows how to get the **long varchar** chapter_text from the database and write it to a text file:

```
C Get_Handler
C ***********

         integer function Get_Handler(info)

         structure        /hdlr_arg/
                character*100     argstr
                integer           argint
         end structure
         record /hdlr_arg/ info

         exec sql begin declare section
                character*2000    segbuf
                integer*4         seglen
                integer*4         datend
                integer*4         maxlen
         exec sql end declare section

            process information passed in via the info record ...
            open file ...


C Get a maximum segment length of 2000 bytes

        maxlen = 2000
        datend = 0


        do while (datend .eq. 0)
C segmentlength: will contain the length of the segment retrieved.
C seg_buf:       will contain a segment of the column chapter_text
C data_end:      will be set to '1' when the entire value in
C                chapter_text has been retrieved.

                exec sql get data (:seqbuf = segment, :seglen =
    1           segmentlength, :datend = dataend)
    2           with maxlength= :maxlen

                write segment to file ...
        end do
        ...
        set info record to return appropriate values ...
        ...
            Get_Handler = 0
        end
```

Dynamic SQL Handler Program

The following examples are of a dynamic SQL handler program that uses the SQLDA. This program fragment shows the declaration and usage of a datahandler in a dynamic SQL program, using the SQLDA. It uses the datahandler Get_Handler() and the HDLR_PARAM structure described in the previous example.

UNIX

```
C main program using SQLDA
C *************************

        program dynamic_hdlr

        exec sql include sqlca
        exec sql include sqlda

C   Do not declare the datahandlers nor the datahandler argument
C   to the ESQL pre-processor.

        external  Put_Handler
        integer*4 Put_Handler

        external  Get_Handler
        integer*4 Get_Handler

C  Declare argument to be passed to datahandler.

        structure          /hdlr_arg/
            character*100     argstr
            integer*4         argint
        end structure
        record /hdlr_arg/ hdlarg

C  Declare SQLDA and IISQLHDLR
        record /IISQLDA/ sqlda
            common /sqlda_area/sqlda
        record /IISQLHDLR/ dathdlr

        integer base_type

C   Declare null indicator to ESQL
        exec sql begin declare section
            integer*2       indvar
            Character*100   stmt_buf
        exec sql end declare section
        . . .
C  Set the IISQLHDLR structure with the appropriate datahandler
C  and datahandler argument.

     dathdlr.sqlhdlr = loc(Get_Handler)
     dathdlr.sqlarg  = loc(hdlarg)

C  Describe the statement into the SQLDA.

        stmt_buf = 'select * from book'.
        exec sql prepare stmt from :stmt_buf
        exec sql describe stmt into SQLDA
        . . .
C  Determine the base_type of the sqldata variables.
        do 20, i = 1, sqlda.sqld

                if (sqlda.sqlvar(i).sqltype .gt. 0) then
                        base_type = sqlda.sqlvar(i).sqltype
                else
                        base_type = -sqlda.sqlvar(i).sqltype
                end if
C Set the sqltype, sqldata and sqlind for each column
C The long varchar column chapter_text will be set to use a
C datahandler
```

```
        if (base_type .eq. IISQ_LVCH_TYPE) then
                sqlda.sqlvar(i).sqltype = IISQ_HDLR_TYPE
                sqlda.sqlvar(i).sqldata = loc(dathdlr)
                sqlda.sqlvar(i).sqlind = loc(indvar)
        else
                . . .
        end if

20      continue

C The Datahandler (Get_Handler) will be invoked for each non-null
C value of column chapter_text retrieved. For null values the
C indicator variable will be set to "-1" and the datahandler
C will not be called.

        . . .
        exec sql execute immediate :stmt_buf using :sqlda
        exec sql begin
                                process row...
        exec sql end
        . . .
        end
```

VMS

```
C main program using SQLDA
C ************************

        program dynhdl
        exec sql include sqlca
        exec sql include sqlda


C  Do not declare the datahandlers nor the datahandler argument
C  to the ESQL pre-processor.

        external  Put_Handler
        integer*4 Put_Handler

        external  Get_Handler
        integer*4 Get_Handler

C   Declare argument to be passed to datahandler.

        structure          /hdlr_arg/
            character*100     argstr
            integer*4         argint
        end structure
        record /hdlr_arg/ hdlarg

C  Declare SQLDA and IISQLHDLR

        record /IISQLDA/ sqlda
            common /sqlda_area/sqlda

        record /IISQLHDLR/ dathdlr

        integer base_type

C   Declare null indicator to ESQL

    exec sql begin declare section
                integer*2       indvar
                Character*100   stmt_buf
    exec sql end declare section
```

```
          . . .
C  Set the IISQLHDLR structure with the appropriate datahandler and
C  datahandler argument.

     dathdlr.sqlhdlr = %loc(Get_Handler)
     dathdlr.sqlarg  = %loc(hdlarg)

C  Describe the statement into the SQLDA.

    stmt_buf = 'select * from book'.
    exec sql prepare stmt from :stmt_buf
    exec sql describe stmt into SQLDA

    . . .

C  Determine the base_type of the sqldata variables.

    do 20, i = 1, sqlda.sqld

            if (sqlda.sqlvar(i).sqltype .gt. 0) then
                        base_type = sqlda.sqlvar(i).sqltype
            else
                        base_type = -sqlda.sqlvar(i).sqltype
            end if

C Set the sqltype, sqldata and sqlind for each column
C The long varchar column chapter_text will be set to use a
C datahandler

            if (base_type .eq. IISQ_LVCH_TYPE) then
                        sqlda.sqlvar(i).sqltype = IISQ_HDLR_TYPE
                        sqlda.sqlvar(i).sqldata = %loc(dathdlr)
                        sqlda.sqlvar(i).sqlind = %loc(indvar)
            else
                        . . .
            end if

20    continue

C  The Datahandler (Get_Handler) will be invoked for each non-null
C  value of column chapter_text retrieved. For null values the
C  indicator variable will be set to “-1” and the datahandler
C  will not be called.

            . . .
            exec sql execute immediate :stmt_buf using :sqlda
      exec sql begin
                process row...
            exec sql end
            . . .
      end
```

Windows

```
C main program using SQLDA
C *************************

        program dynhdl
        exec sql include sqlca
        exec sql include sqlda
```

```
C  Do not declare the datahandlers nor the datahandler argument
C  to the ESQL pre-processor.

        external  Put_Handler
        integer*4 Put_Handler

        external  Get_Handler
        integer*4 Get_Handler

C   Declare argument to be passed to datahandler.

        structure          /hdlr_arg/
            character*100     argstr
            integer*4         argint
        end structure
        record /hdlr_arg/ hdlarg

C  Declare SQLDA and IISQLHDLR

        record /IISQLDA/ sqlda
            common /sqlda_area/sqlda

        record /IISQLHDLR/ dathdlr

        integer base_type

C   Declare null indicator to ESQL

    exec sql begin declare section
                integer*2       indvar
                Character*100   stmt_buf
    exec sql end declare section

            . . .

C  Set the IISQLHDLR structure with the appropriate datahandler and
C  datahandler argument.

     dathdlr.sqlhdlr = %loc(Get_Handler)
     dathdlr.sqlarg  = %loc(hdlarg)

C  Describe the statement into the SQLDA.

    stmt_buf = 'select * from book'.
    exec sql prepare stmt from :stmt_buf
    exec sql describe stmt into SQLDA

    . . .

C  Determine the base_type of the sqldata variables.

    do 20, i = 1, sqlda.sqld

            if (sqlda.sqlvar(i).sqltype .gt. 0) then
                        base_type = sqlda.sqlvar(i).sqltype
            else
                        base_type = -sqlda.sqlvar(i).sqltype
            end if

C Set the sqltype, sqldata and sqlind for each column
C The long varchar column chapter_text will be set to use a
C datahandler
```

```
            if (base_type .eq. IISQ_LVCH_TYPE) then
                        sqlda.sqlvar(i).sqltype = IISQ_HDLR_TYPE
                        sqlda.sqlvar(i).sqldata = %loc(dathdlr)
                        sqlda.sqlvar(i).sqlind = %loc(indvar)
            else
                        . . .
            end if

20    continue

C  The Datahandler (Get_Handler) will be invoked for each non-null
C  value of column chapter_text retrieved. For null values the
C  indicator variable will be set to "-1" and the datahandler
C  will not be called.

            . . .
            exec sql execute immediate :stmt_buf using :sqlda
      exec sql begin
                process row...
            exec sql end
            . . .
      end
```

Preprocessor Operation

This section describes the embedded SQL preprocessor for Fortran and the steps required to create, compile, and link an Embedded SQL program.

Include File Processing

The following sections describe include file processing for UNIX, VMS, and Windows.

Including Files – UNIX

The Embedded SQL **include** statement provides a means to include external files in your program's source code. Its syntax is:

exec sql include *filename*

where *filename* is a single quoted string constant specifying a file name, or an external symbol that points to the file name. If you do not specify an extension to the filename, the default Fortran input file extension ".sf" is assumed.

This statement is normally used to include variable declarations, although it is not restricted to such use. For more details on the **include** statement, see the *SQL Reference Guide.*

The included file is preprocessed and an output file with the same name but with the default output extension ".f" (UNIX) or ".for" (VMS and Windows), is generated. You can override this default output extension with the **-o**.*ext* flag on the command line. In the original source file that specified the **include** statement, a new reference is made to the output file with the Fortran **include** statement. If you use the **-o** flag with no extension, an output file is not generated for the **include** statement.

If you use both the **-o.***ext* and the **-o** flags, then the preprocessor generates the specified extension for the translated **include** statements in the program, but does not generate new output files for the statements.

For example, assume that no overriding output extension was explicitly given on the command line. The Embedded SQL statement:

```
exec sql include 'employee.dcl'
```

is preprocessed to the Fortran statement:

```
include 'employee.f'
```

and the "employee.dcl" file is translated into the Fortran file "employee.f".

As another example, assume that a source file called "inputfile" contains the following **include** statement:

```
exec sql include 'mydecls'
```

The name "mydecls" can be defined as a system environment variable pointing to the file "/usr/headers/myvars.sf". For example:

```
setenv mydecls "/usr/headers/myvars.sf"
```

Because the extension ".sf" is the default input extension for Embedded SQL **include** files, you do not need to specify it when defining a logical name for the file.

Assume now that "inputfile" is preprocessed with the command:

```
esqlf -o.hdr inputfile
```

The command line specifies ".hdr" as the output file extension for include files. As the file is preprocessed, the **include** statement shown earlier is translated into the Fortran statement:

```
include '/usr/headers/myvars.hdr'
```

and the Fortran file "/usr/headers/myvars.hdr" is generated as output for the original include file, "/usr/headers/myvars.sf".

You can also specify include files with a relative path. For example, if you preprocess the file "/dev/mysource/myfile.sf", the ESQL statement:

```
exec sql include '../headers/myvars.sf'
```

is preprocessed to the Fortran statement:

```
include '../headers/myvars.f'
```

and the Fortran file "/dev/headers/myvars.f" is generated as output for the original include file, "/dev/headers/myvars.sf".

Including Files – VMS

The Embedded SQL **include** statement provides a means to include external files in your program's source code. Its syntax is:

> **exec sql include** *filename*

where *filename* is a single quoted string constant specifying a file name, or a logical name that points to the file name. If you do not specify an extension to the filename, the default Fortran input file extension ".sf" is assumed.

This statement is normally used to include variable declarations, although it is not restricted to such use. For more details on the **include** statement, see the *SQL Reference Guide.*

The included file is preprocessed and an output file with the same name but with the default output extension ".for" is generated. You can override this default output extension with the **-o***.ext* flag on the command line. In the original source file that specified the **include** statement, a new reference is made to the output file with the Fortran **include** statement. If you use the **-o** flag with no extension, an output file is not generated for the **include** statement. This is useful for program libraries that use MMS dependencies.

If you use both the **-o.***ext* and the **-o** flags, then the preprocessor generates the specified extension for the translated **include** statements in the program, but does not generate new output files for the statements.

For example, assume that no overriding output extension was explicitly given on the command line. The Embedded SQL statement:

```
exec sql include 'employee.dcl'
```

is preprocessed to the Fortran statement:

```
include 'employee.for'
```

and the employee.dcl file is translated into the Fortran file "employee.for".

As another example, assume that a source file called "inputfile" contains the following **include** statement:

```
exec sql include 'mydecls'
```

The name “mydecls” can be defined as a system logical name pointing to the file “dra1:[headers]myvars.sf” by means of the following command at the system level:

```
define mydecls dra1:[headers]myvars
```

Because the extension “.sf” is the default input extension for Embedded SQL **include** files, it does not need to be specified when defining a logical name for the file.

Assume now that “inputfile” is preprocessed with the command:

```
esqlf -o.hdr inputfile
```

The command line specifies “.hdr” as the output file extension for include files. As the file is preprocessed, the **include** statement shown earlier is translated into the Fortran statement:

```
include 'dra1:[headers]myvars.hdr'
```

and the Fortran file “dra1:[headers]myvars.hdr” is generated as output for the original include file, “dra1:[headers]myvars.sf”.

You can also specify include files with a relative path. For example, if you preprocess the file “dra1:[mysource]myfile.sf”, the Embedded SQL statement:

```
exec sql include '[-.headers]myvars.sf'
```

is preprocessed to the Fortran statement:

```
include '[-.headers]myvars.for'
```

and the Fortran file “dra1:[headers]myvars.for” is generated as output for the original include file, “dra1:[headers]myvars.sf”.

Including Files – Windows

The Embedded SQL **include** statement provides a means to include external files in your program’s source code. Its syntax is:

> **exec sql include** *filename*

where *filename* is a single quoted string constant specifying a file name, or a logical name that points to the file name. If you do not specify an extension to the filename, the default Fortran input file extension “.sf” is assumed.

This statement is normally used to include variable declarations, although it is not restricted to such use. For more details on the **include** statement, see the *SQL Reference Guide.*

The included file is preprocessed and an output file with the same name but with the default output extension ".for" is generated. You can override this default output extension with the **-o**.*ext* flag on the command line. Within the original source file that specified the **include** statement, a new reference is made to the output file with the Fortran **include** statement. If you use the **-o** flag with no extension, an output file is not generated for the **include** statement.

If you use both the **-o.***ext* and the **-o** flags, then the preprocessor generates the specified extension for the translated **include** statements in the program, but does not generate new output files for the statements.

For example, assume that no overriding output extension was explicitly given on the command line. The Embedded SQL statement:

```
exec sql include 'employee.dcl'
```

is preprocessed to the Fortran statement:

```
include 'employee.for'
```

and the employee.dcl file is translated into the Fortran file "employee.for".

As another example, assume that a source file called "inputfile" contains the following **include** statement:

```
exec sql include 'mydecls'
```

The name "mydecls" can be defined as a system logical name pointing to the file "c:\usr\header\myvars.for" by means of the following command at the system level:

```
set mydecls=c:\usr\header\myvars
```

Because the extension ".for" is the default input extension for Embedded SQL **include** files, it does not need to be specified when defining a logical name for the file.

Assume now that "inputfile" is preprocessed with the command:

```
esqlf -o.hdr inputfile
```

The command line specifies ".hdr" as the output file extension for include files. As the file is preprocessed, the **include** statement shown earlier is translated into the Fortran statement:

```
include 'c:\usr\header\myvars.hdr'
```

and the Fortran file "c:\usr\header\myvars.hdr" is generated as output for the original include file, "c:\usr\header\myvars.for".

You can also specify include files with a relative path. For example, if you preprocess the file "c:\usr\mysource\myfile.sf", the Embedded SQL statement:

```
exec sql include '..\header\myvars'
```

is preprocessed to the Fortran statement:

```
include '..header\myvars.for'
```

and the Fortran file "..header\myvars.for" is generated as output for the original include file, "..header\myvars".

Including Source Code with Labels

Some Embedded SQL statements generate labels (statement numbers). The statement numbers 7000 through 12000 are reserved for the preprocessor. If you include a file containing statements that generate labels, be careful to include the file only once in a given Fortran scope. Otherwise, you may find that the compiler later complains that the generated labels are defined more than once in that scope.

The statements that generate labels are the Embedded SQL **select** statement and all the Embedded SQL/FORMS block-type statements, such as **display** and **unloadtable**.

Coding Requirements for Writing Embedded SQL Programs

The following sections describe the coding requirements for writing Embedded SQL programs.

Comments Embedded in Fortran Output

Each Embedded SQL statement generates one comment and a few lines of Fortran code. You may find that the preprocessor translates 50 lines of Embedded SQL into 200 lines of Fortran. This can confuse the program developer who is trying to debug the original source code. To facilitate debugging, a comment corresponding to the original Embedded SQL source delimits each group of Fortran statements associated with a particular statement. Each comment is one line long and informs the reader of the file name, line number, and type of statement in the original source file.

Embedded SQL Statements and Fortran If Blocks

Because each Embedded SQL statement must be on a line by itself, you must use the block-style Fortran **if** statement to conditionally transfer control to Embedded SQL statements. For example:

```
if (error) then
     exec sql message 'Error on update'
         exec sql sleep 2
end if
```

Note that the **esqlf** preprocessor also generates many nested constructs of **do** loops and **if** blocks—specifically, for Embedded SQL block-structured statements, such as **display** and **unloadtable**. If you mistakenly omit an **end if** from your Fortran source, the Fortran compiler complains that there is a missing **end** statement, which you can trace back to a preprocessor-generated **if** or **do** (VMS or Windows).

You can usually solve this problem by checking for matching **if-end** pairs in the original Embedded SQL Fortran source file. In VMS or Windows, you can also check for **do-end** pairs as well.

Embedded SQL Statements that Generate Labels

The Embedded SQL statements that generate labels are the Embedded SQL **select** statement and all the Embedded SQL/FORMS block-type statements. Each of these statements reserves its own range of 200 labels in a defined range for such statements of 7000 through 12000. Consequently, you cannot have more than 200 of any single label-generating statement in the same program unit. For example, 201 **display** statements in a single subroutine causes a compiler error indicating that a particular label was used more than once. You could, however, have 200 **display** statements and 200 **unloadtable** statements without causing a problem.

Embedded SQL Statements that Do Not Generate Code

The following Embedded SQL declarative statements do not generate any Fortran code:

declare cursor
declare statement
declare table
whenever

These statements must not contain labels. Also, they must not be coded as the only statements in Fortran constructs that do not allow *null* statements.

Command Line Operations

The following sections describe command line operations that you can use to turn your Embedded SQL/Fortran source program into an executable program. The commands to preprocess, compile, and link your program are also described in these sections.

The Embedded SQL Preprocessor Command

The Fortran preprocessor is invoked by the following command line:

esqlf {*flags*} {*filename*}

where *flags* are

Flag	Description
-d	Adds debugging information to the runtime database error messages generated by Embedded SQL. The source file name, error number and the statement in error are printed with the error message.
-f[*filename*]	Writes preprocessor output to the named file. If you do not specify *filename*, the output is sent to standard output, one screen at a time.
-i*N*	Sets the default size of integers to *N* bytes. *N* must be either 2 or 4. The default is 4. If 2 is used, you must also use the **-i2** compiler flag (UNIX), the **noi4** qualifier (VMS), or the /integer_size:16 compiler flag (Windows).
-l	Writes preprocessor error messages to the preprocessor's listing file, as well as to the terminal. The listing file includes preprocessor error messages and your source text in a file named *filename***.**lis, where *filename* is the name of the input file.
-lo	Like **-l**, but the generated Fortran code also appears in the listing file.
-o	Directs the processor not to generate output files for include files. This flag does not affect the translated **include** statement in the main program. The preprocessor generates a default extension for the translated include file statements unless you use the **-o**.*ext* flag.
-o.*ext*	Specifies the extension given by the preprocessor to both the translated **include** statements in the main program and the generated output files. If this flag is not provided, the default extension is ".f" (UNIX) or ".for" (VMS and Windows). If you use this flag in combination with the **-o** flag, then the preprocessor generates the specified extension for the translated **include** statements, but does not generate new output files for the **include** statements.

Flag	Description
-s	Reads input from standard input and generates Fortran code to standard output. This is useful for testing unfamiliar statements. If you specify the **-l** option with this flag, the listing file is called "stdin.lis." To terminate the interactive session, type **Control-D** (UNIX) or **Control-Z** (VMS and Windows).
-sqlcode **-nosqlcode**	Indicates the file declares an integer variable named **SQLCODE** to receive status information from SQL statements. That declaration need not be in an exec sql begin/end declare section. This feature is provided for ISO Entry SQL-92 conformity. However, the ISO Entry SQL92 specification describes **SQLCODE** as a "deprecated feature," and recommends using the **SQLSTATE** variable. Tells the preprocessor not to assume the existence of a status variable named **SQLCODE**.
-w	Prints warning messages.
-wopen	This flag is identical to **-wsql=open**. However, **-wopen** is supported only for backwards capability. See **-wsql=open** for more information.
VMS **-?**	Shows the command line options for the **esqlf** command.
UNIX **--**	Shows the command line options for the **esqlf** command.
Windows **-?**	Shows the command line options for the **esqlf** command.
-wsql=entry_ SQL92 **-wsql=open**	Causes the preprocessor to flag any usage of syntax or features that do not conform to the ISO Entry SQL92 entry level standard. (This is also known as the "FIPS flagger" option.) Use *open* only with OpenSQL syntax. **-wsql = open** generates a warning if the preprocessor encounters an Embedded SQL statement that does not conform to OpenSQL syntax. (For OpenSQL syntax, see the *OpenSQL Reference Guide*.) This flag is useful if you intend to port an application across an Enterprise Access product. The warnings do not affect the generated code and the output file may be compiled. This flag does not validate the statement syntax for any Enterprise Access product whose syntax is more restrictive than that of OpenSQL.

The Embedded SQL/Fortran preprocessor assumes that input files are named with the extension ".sf". You can override this default by specifying the file extension of the input file(s) on the command line. The output of the preprocessor is a file of generated Fortran statements in tab format with the same name and the extension ".f" (UNIX) or ".for" (VMS and Windows).

If you enter the command without specifying any flags or a filename, Ingres displays a list of flags available for the command.

The following table presents a range of the options available with **esqlf**.

Esqlf Command Examples

Command	Comment
esqlf file1	Preprocesses "file1.sf" to: "file1.f" (UNIX) "file1.for" (VMS and Windows)
esqlf file2.xf	Preprocesses "file2.xf" to "file2.f" (UNIX "file2.for" (VMS and Windows)
esqlf -l file3	Preprocesses "file3.sf" to: "file3.f" (UNIX) "file3.for" (VMS and Windows) and creates listing "file3.lis"
esqlf -s	Accepts input from standard input
esqlf -ffile4.out file4	Preprocesses "file4.sf" to "file4.out"
esqlf	Displays a list of flags available for this command

The Fortran Compiler

The preprocessor generates Fortran code. The code generated is in tab format, in which each Fortran statement follows an initial tab. (For information on the Embedded SQL format acceptable as input to the preprocessor, see Embedded SQL Statement Syntax for Fortran in this chapter.)

UNIX

Use the UNIX **f77** command to compile this code. You can use most of the **f77** command line options. If you use the **-i2** compiler flag to interpret **integer** and **logical** declarations as 2-byte objects, you must have run the Fortran preprocessor with the **-i2** preprocessor flag.

As mentioned in The SQL Communications Area in this chapter, you may want to use the **-u** compiler flag to verify that the SQLCA has been declared correctly with an **include sqlca** statement in all program units containing Embedded SQL statements.

The following example preprocesses and compiles the file "test1." The Embedded SQL preprocessor assumes the default extension:

```
esqlf test1
f77 test1.f
```

VMS

Use the VMS **fortran** command to compile this code. Most of the **fortran** command line options can be used. If you use the **noi4** qualifier to interpret **integer** and **logical** declarations as 2-byte objects, you must have run the Fortran preprocessor with the **-i2** flag. You must not use the **g_floating** qualifier if floating-point values in the file are interacting with Ingres floating-point objects. Note, too, that many of the statements that the Embedded SQL preprocessor generates are nonstandard extensions provided by VAX/VMS. Consequently, you should not attempt to compile with the **nof77** qualifier.

As mentioned in The SQL Communications Area in this chapter, you may want to use the **warnings=declarations** qualifier to verify that the SQLCA has been declared correctly with an **include sqlca** statement in all program units containing Embedded SQL statements.

As of Ingres II 2.0/0011 (axm.vms/00) Ingres uses member alignment and IEEE floating-point formats. Embedded programs must be compiled with member alignment turned on. In addition, embedded programs accessing floating-point data (including the MONEY data type) must be compiled to recognize IEEE floating-point formats.

The following example preprocesses and compiles the file "test1." The Embedded SQL preprocessor assumes the default extension:

```
esqlf test1
fortran/list test1
```

Windows

Use the Windows **df** command to compile this code. The following compile options are required for Windows:

/name:as_is	Treat uppercase and lowercase letters as different.
/iface:nomixed_str_len_arg	Requests that the hidden lengths be placed in sequential order at the *end* of the argument list.
/iface:cref	Names are not decorated, the caller cleans the call stack, and var args are supported.

If you use the **/integer_size:16** qualifier to interpret **integer** and **logical** declarations as 2-byte objects, you must have run the Fortran preprocessor with the **-i2** flag.

As mentioned in the chapter "The SQL Communications Area," you may want to use the **warnings=declarations** qualifier to verify that the SQLCA has been declared correctly with an **include sqlca** statement in all program units containing Embedded SQL statements.

The following example preprocesses and compiles the file "test1." The Embedded SQL preprocessor assumes the default extension:

```
esqlf test1
df /compile_only /name:as_is /iface:nomixed_str_len_arg /iface:cref test1
```

Note: For any operating system specific information on compiling and linking ESQL/Fortran programs, see the Readme file.

Linking an Embedded SQL Program

Embedded SQL programs require procedures from an Ingres library or libraries depending on your operating system as described below.

UNIX

The Ingres library "libingres.a" must be included in your compile (**f77**) or link (**ld**) command after all user modules. The following example demonstrates how to compile and link an Embedded SQL program called "dbentry" that has passed through the preprocessor:

```
f77 -o dbentry dbentry.f \
     $II_SYSTEM/ingres/lib/libingres.a\
     -lm\
     -lc
```

Note that you must include the math library (the "m" argument to the **-l** flag).

Ingres shared libraries are available on some Unix platforms. To link with these shared libraries replace "libingres.a" in your link command with:

```
-L $II_SYSTEM/ingres/lib -linterp.1 -lframe.1 -lq.1 \
     -lcompat.1
```

To verify if your release supports shared libraries check for the existence of any of these four shared libraries in the $II_SYSTEM/ingres/lib directory. For example:

```
ls -l $II_SYSTEM/ingres/lib/libq.1.*
```

VMS

Embedded SQL programs require procedures from several VMS shared libraries. When you have preprocessed and compiled an Embedded SQL program, you can link it. Assuming the object file for your program is called "dbentry," use the following link command:

```
link dbentry.obj,-
 ii_system:[ingres.files]esql.opt/opt
```

Windows

The Ingres library libingres.lib must be included in your compile (**df**) or link (**link**) command after all user modules. The following example demonstrates how to compile and link an Embedded SQL program called "dbentry" that has passed through the preprocessor:

```
df /name:as_is /iface:nomixed_str_len_arg /iface:cref dbentry.for \
     %II_SYSTEM%\ingres\lib\libingres.lib \
     /link /nodefaultlib
```

Linking Precompiled Forms

The *Forms-based Application Development Tools User Guide* and the Fortran Variables and Data Types section in this chapter, discuss how to declare a precompiled form to the FRS. In order to use such a form in your program, you must also follow the steps described below depending on your operating system.

UNIX

In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a C language source file in your directory that contains a description of the form. VIFRED lets you select the name for the file. Before compiling and linking this file with your Embedded SQL program, you must make the form name, or *formid*, contained therein consistent with the way Fortran stores external symbols.

When you compile the Fortran source file generated from your Embedded SQL program, the Fortran compiler appends an underscore to all external symbols. Some Fortran compilers also truncate names to six characters before appending the underscore. Because the *formid* is an external symbol, it too has an underscore appended and may be truncated. In order to resolve this link-time inconsistency, you must change the *formid* as it appears in the file created by VIFRED.

This means you must edit the C source file created by VIFRED that contains your compiled form. When you invoke the editor, go to the end of the file. You will see a line that begins "FRAME * *formid*" where *formid* is the name of the form. You must append an underscore to *formid* and truncate the name, if necessary. The following example shows the relevant lines of a C source file created by VIFRED where "empfrm" is the formid:

```
     ...

FRAME * empfrm = {
      &_empfrm, };
```

You should modify the file to append the required underscore, as follows:

```
     ...

FRAME * empfrm_ = {
      &_empfrm, };
```

This example assumes that your compiler does not truncate external symbols.

Note that you do not need to make changes to the declarations containing the *formid* in your Embedded SQL program. The Fortran compiler changes this reference when it creates the object file.

After modifying your C file this way, you can compile it into linkable object code with the UNIX command:

```
cc -c formfile.c
```

where "formfile.c" is the name of the compiled form source file created by VIFRED.

The output of this command is a file with the extension ".o". You then link this object file with your program, as in the following example:

```
f77 -o formentry formentry.f \
         formfile.o \
         $II_SYSTEM/ingres/lib/libingres.a\
     -lm \
     -lc
```

VMS

In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a file in your directory describing the form in the MACRO language. VIFRED lets you select the name for the file. Once you have created the MACRO file this way, you can assemble it into linkable object code with the VMS command:

macro *filename*

The output of this command is a file with the extension ".obj". You then link this object file with your program by listing it in the link command, as in the following example:

```
link formentry,-
 empform.obj,-
 ii_system:[ingres.files]esql.opt/opt
```

Windows

In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a C language file in your directory describing the form. VIFRED lets you select the name for the file. Once you have created the C language file this way, you can compile it into linkable object code with the Windows command:

cl –c –MD *filename*

The output of this command is a file with the extension ".obj". You then link this object file with your program by listing it in the link command, as in the following example:

```
link /out:formentry.exe, \
 empform.obj,\
 %II_SYSTEM%\ingres\lib\libingres.lib
```

Linking an Embedded SQL Program without Shared Libraries -VMS

While the use of shared libraries in linking Embedded SQL programs is recommended for optimal performance and ease of maintenance, non-shared versions of the libraries have been included in case you need them. Non-shared libraries required by Embedded SQL are listed in the "esql.noshare" options file. The options file must be included in your link command *after* all user modules. The libraries must be specified in the order given in the options file.

The following example demonstrates the link command for an Embedded SQL program called "dbentry" that has been preprocessed and compiled:

```
link dbentry,-
 ii_system:[ingres.files]esql.noshare/opt
```

Placing User-written Embedded SQL Routines in Shareable Images -VMS

When you plan to place your code in a shareable image, note the following about the **psect** attributes of your global or external variables:

- As a default, some compilers mark global variables as shared (SHR: every user who runs a program linked to the shareable image sees the same variable) and others mark them as not shared (NOSHR: every user who runs a program linked to the shareable image gets their own private copy of the variable).
- Some compilers support modifiers you can place in your source code variable declaration statements to explicitly state which attributes to assign a variable.
- The attributes that a compiler assigns to a variable can be overridden at link time with the **psect_attr** link option. This option overrides attributes of all variables in the **psect**.

Consult your compiler reference manual for further details.

Embedded SQL/Fortran Preprocessor Errors

To correct most errors, you may wish to run the Embedded SQL preprocessor with the listing (**-l**) option on. The listing is sufficient for locating the source and reason for the error.

For preprocessor error messages specific to Fortran, see Preprocessor Error Messages in this chapter.

Preprocessor Error Messages

The following is a list of error messages specific to the Fortran language.

E_DC000A

"Table 'employee' contains column(s) of unlimited length."

Explanation: Character string(s) of zero length have been generated. This causes a compile-time error. You must modify the output file to specify an appropriate length.

E_E10001

"Unsupported Fortran type '%0c' used. Double assumed. Ingres does not support the Fortran types complex and double complex."

Explanation: There is no Ingres type corresponding to complex or double complex, so the preprocessor does not map this declaration to a Ingres type. The preprocessor will continue to generate code as if you had declared the variable in question to be of type double precision. If you want to store the two real (or double precision) components of a complex (or double complex) variable, declare a pair of real (or double precision) variables to the preprocessor, copy the components to them, and then store the copies.

E_E10002

"Fortran parameter may only be used with values. Type names, variable names, and parameter names are not allowed."

Explanation: You have used the Fortran "parameter *name* = *value*" statement, but *value* is not an integer constant, a floating constant, or a string constant. You may have used the name of a Fortran data type, or a variable (or parameter) name instead of one of the legal constant types. If you do wish Ingres to know about this name then you must change the *value* to be a constant.

E_E10003

"Incorrect indirection on variable '%0c'. The variable is declared as an array and is not subscripted, or is subscripted but is not declared as an array (%1c, %2c)."

Explanation: This error occurs when the value of a variable is incorrectly expressed because of faulty indirection. For example, the name of an integer array has been given instead of a single array element, or, in the case of string variables, a single element of the string (for example, a character) has been given instead of the name of the array. The preprocessor will continue to generate code, but the program will not execute correctly if it is compiled and run. Either redeclare the variable with the intended indirection, or change its use in the current statement.

E_E10004 "Last Fortran structure field referenced in '%0c' is unknown."

Explanation: This error occurs when the preprocessor encounters an unrecognized name in a structure reference. The preprocessor will continue to generate code, but this statement will either cause a runtime error or produce the wrong result if the resulting program is compiled and run. Check for misspellings in field names and ensure that all of the structure fields have been declared to the preprocessor.

E_E1000A "Undefined structure name '%0c' used in record declaration."

Explanation: You have declared a record variable using the name of a structure that is unknown to the preprocessor. The preprocessor will continue to generate code, but the resulting program will not run properly. If you do not use this variable with a Ingres statement, remove the record declaration. Otherwise, ensure that the corresponding structure declaration is made known to the preprocessor.

E_E1000B "Field '%0c' in record '%1c' is not an elementary variable."

Explanation: Record variables used in SQL as a single object must contain only scalar fields. Arrays and nested records are not allowed in this context. For example the following will cause an error on "obj.oname" in the select statement because it is an array variable:

```
exec sql begin declare section
     structure /object/
        character*10   oname
        integer        ovals(4)
     end structure
     record/object/ obj
exec sql end declare section
exec sql select * into :obj from objects
```

Either flatten the record variable declaration or enumerate all fields when using the variable.

E_E1000C "Illegal length specified for Fortran numeric variable."

Explanation: Fortran integer variables can be 1, 2, or 4 bytes, and floating-point variables can be either 4 or 8 bytes. Specifying any other value is illegal.

Sample Applications

This section contains sample applications.

The Department-Employee Master/Detail Application

This application uses two database tables joined on a specific column. This typical example of a department and its employees demonstrates how to process two tables as a master and a detail.

The program scans through all the departments in a database table, in order to reduce expenses. Based on certain criteria, the program updates department and employee records. The conditions for updating the data are the following:

Departments:

- If a department has made less than $50,000 in sales, the department is dissolved.

Employees:

- If an employee was hired since the start of 1985, the employee is terminated.
- If the employee's yearly salary is more than the minimum company wage of $14,000 and the employee is not nearing retirement (over 58 years of age), the employee takes a 5% pay cut.
- If the employee's department is dissolved and the employee is not terminated, the employee is moved into a state of limbo to be resolved by a supervisor.

This program uses two cursors in a master/detail fashion. The first cursor is for the "department" table, and the second cursor is for the "employee" table. Both tables are described in **declare table** statements at the start of the program. The cursors retrieve all the information in the tables, some of which is updated. The cursor for the "employee" table also retrieves an integer date interval whose value is positive if the employee was hired after January 1, 1985.

Each row that is scanned, from both the "department" table and the "employee" table, is recorded in an output file. This file serves both as a log of the session and as a simplified report of the updates that were made.

Each section of code is commented for the purpose of the application and also to clarify some of the uses of the Embedded SQL statements. The program illustrates table creation, multi-statement transactions, all cursor statements, direct updates and error handling.

If your application requires the use of structures, see Fortran Variables and Data Types in this chapter for more information.

This application runs in UNIX, VMS, and Windows environments.

```
C
C  Program: ProcessExpenses
C  Purpose: Main entry point to process department and employee expenses
C

      program ProcessExpenses

      exec sql include sqlca

      exec sql declare dept table
     1 (name         char(12) not null,
     2 totsales      decimal(14,2) not null,
     3 employees     integer2 not null)
      exec sql declare employee table
     1 (name         char(20) not null,
     2 age           integer1 not null,
     3 idno          integer4 not null,
     4 hired         date not null,
     5 dept          char(12) not null,
     6 salary        decimal(14,2)  not null)

C "State-of-Limbo" for employees who lose their departments
      exec sql declare toberesolved table
     1 (name       char(20) not null,
     2 age         integer1 not null,
     3 idno        integer4 not null,
     4 hired       date not null,
     5 dept        char(12) not null,
     6 salary      decimal(14,2) not null)

      print *, 'Entering application to process expenses.'
      open(unit = 1, file = 'expenses.log', status = 'new')
      call InitDb
      call ProcessDepts
      call EndDb
      close(unit = 1, status = 'keep')
      print *, 'Successful completion of application.'
      end

C
C Subroutine: InitDb
C Purpose:    Initialize the database. Connect to the database and
C             abort if an error. Before processing employees,
C             confirm that the table for employees who lose
C             their departments,"toberesolved,"
C             exists. Initiate multi-statement transaction.
C Parameters: None.
C

      subroutine InitDb

      exec sql include sqlca

      exec sql whenever sqlerror stop

      exec sql connect personnel
```

```
      write (1, 10)
10    format ('Creating "To_Be_Resolved" table.')
      exec sql create table toberesolved
     1     (name       char(20) not null,
     2      age        integer1 not null,
     3      idno       integer4 not null,
     4      hired      date not null,
     5      dept       char(10) not null,
     6      salary     decimal(14,2) not null)

      end

C
C Subroutine:     EndDb
C Purpose:        End the multi-statement transaction and access
C                 to the database.
C Parameters:     None.
C

      subroutine EndDb

      exec sql include sqlca

      exec sql commit
      exec sql disconnect

      end

C
C Subroutine: ProcessDepts
C Purpose:    Scan through all the departments, processing each
C             one. If the department has made less than $50,000
C             in sales,then the department is
C             dissolved. For each department, process all the
C             employees (they may even be moved to
C             another table.) If an employee was terminated,
C             update the department's employee counter.
C Parameters: None
C

      subroutine ProcessDepts

      exec sql include sqlca

      exec sql begin declare section

           character*12        dname
           double precision    dsales
           integer*2           demps
C Employees terminated
           integer*2           dterm

      exec sql end declare section

C Minimum sales of department
      parameter (mindeptsales = 50000.00)
C Was the dept deleted?
      logical deldept
C Formatting value
      character*20 deptformat

      exec sql declare deptcsr cursor for
     1    select name, totsales, employees
     2    from dept
     3    for direct update of name, employees
```

```
C All errors from this point on close down the application
      exec sql whenever sqlerror call closedown
C Close deptcsr
      exec sql whenever not found go to 100

      exec sql open deptcsr
      dterm = 0
55    if (sqlcod .ne. 0) go to 555
      exec sql fetch deptcsr into :dname, :dsales, :demps

C Did the department reach minimum sales?
      if (dsales .lt. mindeptsales) then
          exec sql delete from dept
     1          where current of deptcsr
          deldept = .true.
          deptformat = ' -- DISSOLVED --'
      else
          deldept = .false.
          deptformat = ' '
      endif

C Log what we have just done
       write (1, 11) dname, dsales, deptformat
11     format ('Department: ', a14, ', Total Sales: ', f12.3, a)

C Now process each employee in the department
      call ProcessEmployees(dname, deldept, dterm)

C If some employees were terminated, record this fact
      if (dterm .gt. 0 .and. .not. deldept) then
          exec sql update dept
     1         set employees = :demps - :dterm
     2         where current of deptcsr
      endif
      go to 55

555   continue

      exec sql whenever not found continue

100   exec sql close deptcsr

      end
C
C Subroutine:  ProcessEmployees
C Purpose:     Scan through all the employees for a particular
C              department.Based on given conditions, the employee
C              may be terminated or given a salary reduction.
C            1. If an employee was hired since 1985, the employee
C               is terminated.
C            2. If the employee's yearly salary is more than the
C               minimum company wage of $14,000 and the employee
C               is not close to retirement (over 58 years of age),
C               the employee takes a 5% salary reduction.
C            3. If the employee's department is dissolved and the
C               employee is not terminated, then the employee is
C               moved into the "toberesolved" table.
C
C Parameters: sname      - Name of current department
C             sdel       - Is current department being dissolved?
C             sterm      - Set locally to record how many employees
C                            were terminated for the current
C                            department.
C

      subroutine ProcessEmployees(sname, sdel, sterm)
```

```
      exec sql include sqlca

      exec sql begin declare section

           character*12     sname
           character*20     name
           integer*2        age
           integer*4        idno
           character*25     chired
           real             salary
           integer*4        ihired

           parameter (salreduc = 0.95)

      exec sql end declare section

C Minimum employee salary
      parameter        (minsal = 14000.00)
      parameter        (nearlyretired = 58)
C Formatting values
      character*12 title
      character*25 description

C Subroutine arguments
      logical        sdel
      integer*2      sterm

C Note the use of the Ingres function to find out who
C has been hired since 1985.

      exec sql declare empcsr cursor for
     1   select name, age, idno, hired, salary,
     2     int4(interval('days', hired-date('01-jan-1985')))
     3   from employee
     4   where dept = :sname
     5   for direct update of name, salary

C  All errors from this point on close down the application
      exec sql whenever sqlerror call closedown
C Close empcsr
      exec sql whenever not found go to 200

      exec sql open empcsr

      sterm = 0
66    if (sqlcod .ne. 0) go to 666

      exec sql fetch empcsr into :name, :age, :idno,
     1     :chired, :salary, :ihired

      if (ihired .gt. 0) then
           exec sql delete from employee
     1          where current of empcsr
           title = 'Terminated:'
           description = 'Reason: Hired since 1985.'
           sterm = sterm + 1

      else if (salary .gt. minsal) then
```

```
C Reduce salary if not nearly retired
           if (age .lt. nearlyretired) then
               exec sql update employee
    1                 set salary = salary * :salreduc
    2                 where current of empcsr
               title = 'Reduction:'
               description = 'Reason: Salary.'
           else
C  Do not reduce salary
               title = 'No Changes:'
               description = 'Reason: Retiring.'
           endif

      else
C  Leave employee alone
           title = 'No Changes:'
           description = 'Reason: Salary.'
      endif

C  Was employee's department dissolved?
      if (deldept) then
          exec sql insert into toberesolved
    1          select *
    2          from employee
    3          where idno = :idno
          exec sql delete from employee
    1          where current OF empcsr
      endif

C  Log the employee's information
      write (1, 12) title, idno, name, age, salary, description
12    format (' ', a, ' ', i6, ', ', a, ', ', i2, ', ', f8.2, ';',
     1        ' ' a)

      go to 66
666   continue

      exec sql whenever not found continue

200   exec sql close empcsr

      end
C
C Subroutine:   CloseDown
C Purpose:      Error handler called any time after InitDb has been
C               successfully completed. In all cases, print the
C               cause of the error and abort the transaction,
C               backing out
C               change Note that disconnecting from the database
C               will implicitly close any open cursors.
C Parameters:   None
C

      subroutine CloseDown

      exec sql include sqlca

      exec sql begin declare section
          character*100 errbuf
      exec sql end declare section

C  Turn off error handling
      exec sql whenever sqlerror continue
```

```
      exec sql copy sqlerror into :errbuf with 100
      write (1, 13)
13    format ('Closed down because of database error:')
      write (1, 14) errbuf
14    format (a)
      close(unit = 1, status = 'keep')

      exec sql rollback
      exec sql disconnect
      print *, stop 'An SQL error occurred - Check the log file.'
      stop

      end
```

The Table Editor Table Field Application

This application edits the Person table in the Personnel database. It is a forms application that allows the user to update a person's values, remove the person, or add new persons. Various table field utilities are provided with the application to demonstrate how they work.

The objects used in this application are shown in the following table:

Object	Description
personnel	The program's database environment.
person	A table in the database, with three columns: name (**char(20)**) age (**smallint**) number (**integer**) Number is unique.
personfrm	The VIFRED form with a single table field.
persontbl	A table field in the form, with two columns: name (**char(20)** age (**integer**) When initialized, the table field includes the hidden column number (**integer**).

At the start of the application, a database cursor is opened to load the table field with data from the "person table". After loading the table field, you can browse and edit the displayed values. You can add, update or delete entries. When finished, the values are unloaded from the table field, and your updates are transferred back into the "person" table.

The application runs in UNIX, VMS, and Windows environments.

```
C
C Program: TableEdit
C Purpose: entry point to edit the "person"
C           table in the database,
C           via a table field.

      program TableEdit

      exec sql include sqlca

      exec sql declare person table
     1 (name  char(20),
     2 age    integer2,
     3 number integer4)

      exec sql begin declare section

C Person information
           character*20   pname
           integer        page
           integer        pnum

           integer maxid

C Table field entry information
C State of data set entry
           integer state
C Record number
           integer    recnum
C Last row in table field
           integer    lastrow

C Utility buffers
C Message buffer
           character*100 msgbuf
C Response buffer for prompts
           character*20 respbuf

      exec sql end declare section

C Update error from database
      logical updaterr
C Transaction aborted
      logical xaborted

C Function to fill table field
      integer LoadTable

C Table field row states
C Empty or undefined row
      parameter (rowundef = 0)
C Appended by user
      parameter (rownew = 1)
C Loaded by program - not updated
      parameter (rowunchanged = 2)
C Loaded by program - since changed
      parameter (rowchanged = 3)
C Deleted by program
      parameter (rowdeleted = 4)

C SQL value for no rows
      parameter (notfound = 100)
```

```
C Set up error handling for main program
      exec sql whenever sqlwarning continue
      exec sql whenever not found continue
      exec sql whenever sqlerror stop

C Start up Ingres and the FORMS system
      exec sql connect 'personnel'

      exec frs forms

C Verify that the user can edit the "person" table
   exec frs prompt noecho ('Password for table editor: ', :respbuf)

      if (respbuf .ne. 'MASTER_OF_ALL') then
           exec frs endforms
           exec sql disconnect
           stop 'No permission for task. Exiting . . .'
      endif

C Assume no SQL errors can happen during screen updating
      exec sql whenever sqlerror continue

      exec frs message ‘Initializing Person Form . . .'
      exec frs forminit personfrm

C
C Initialize "persontbl" table field with a data set in FILL mode,
C so that the runtime user can append rows. To keep track of
C events occurring to original rows loaded into the table field,
C hide the unique person number.
C
    exec frs inittable personfrm persontbl fill (number = integer4)

      maxid = LoadTable()

      exec frs display personfrm update
      exec frs initialize

      exec frs activate menuitem 'Top'
      exec frs begin
C
C Provide menu items to scroll to both extremes of
C the table field.
C
           exec frs scroll personfrm persontbl to 1
      exec frs end

      exec frs activate menuitem 'Bottom'
      exec frs begin
           exec frs scroll personfrm persontbl to end
      exec frs end

      exec frs activate menuitem 'Remove'
      exec frs begin
C
C Remove the person in the row the user's cursor is on.
C If there are no persons, exit operation with message.
C Note that this check cannot really happen, as there is
C always an UNDEFINED row in FILL mode.
C
           exec frs inquire_frs table personfrm
     1           (lastrow = lastrow(persontbl))
           if (lastrow .eq. 0) then
                exec frs message 'Nobody to Remove'
                exec frs sleep 2
                exec frs resume field persontbl
```

```
           endif

C Record it later
           exec frs deleterow personfrm persontbl

      exec frs end

      exec frs activate menuitem 'Find'
      exec frs begin
C
C Scroll user to the requested table field entry.
C Prompt the user for a name, and if one is typed in,
C loop through the data set searching for it.
C
          exec frs prompt ('Person''s name : ', :respbuf)
          if (respbuf(1:1) .eq. ' ') then
               exec frs resume field persontbl
          endif

          exec frs unloadtable personfrm persontbl
     1           (:pname  = name,
     2            :recnum = _record,
     3            :state  = _state)
          exec frs begin

C Do not compare with deleted rows
              if ((pname .eq. respbuf) .and.
     1            (state .ne. rowdeleted)) then

                   exec frs scroll personfrm persontbl
     1                 to :recnum
                   exec frs resume field persontbl
              endif

          exec frs end

C Fell out of loop without finding name
          write (msgbuf, 10) respbuf
10        format ('Person "', a,
     1         '" not found in table [HIT RETURN] ')
          exec frs prompt noecho (:msgbuf, :respbuf)

      exec frs end

      exec frs activate menuitem 'Exit'
      exec frs begin
          exec frs validate field persontbl
          exec frs breakdisplay
      exec frs end

      exec frs finalize

C
C Exit person table editor and unload the table field. If any
C updates, deletions or additions were made, duplicate these
C changes in the source table. If the user added new people,
C assign a unique person id to each person before adding the person
C to the table. To do this, increment the previously-saved maximum
C id number with each insert.
C

C Do all the updates in a transaction
      exec sql savepoint savept

C
C Hard code the error handling in the UNLOADTABLE loop, in
```

```
C order to cleanly exit the loop.
C
      exec sql whenever sqlerror continue

      updaterr = .false.
      xaborted = .false.

      exec frs message 'Exiting Person Application . . .'
      exec frs unloadtable personfrm persontbl
     1      (:pname = name, :page = age,
     2       :pnum = number, :state = _state)
      exec frs begin

C Appended by user. Insert with new unique id.
           if (state .eq. rownew) then

                maxid = maxid + 1
                exec sql insert into person (name, age, number)
     1                 values (:pname, :page, :maxid)

C Updated by user. Reflect in table.
           else if (state .eq. rowchanged) then

                exec sql update person set
     1               name = :pname, age = :page
     2               where number = :pnum
C
C Deleted by user, so delete from table. Note that only
C original rows, not rows appended at runtime, are saved
C by the program.
C
           else if (state .eq. rowdeleted) then

                exec sql delete from person
     1               where number = :pnum

C Ignore UNDEFINED or UNCHANGED - No updates
           endif

C
C Handle error conditions -
C If an error occurred, abort the transaction.
C If no rows were updated, inform user and prompt
C for continuation.
C
           if (sqlcod .lt. 0) then
C SQL error
                exec sql inquire_sql (:msgbuf = errortext)
                exec sql rollback to savept
                updaterr = .true.
                xaborted = .true.
                exec frs endloop

           else if (sqlcod .eq. notfound) then

                write (msgbuf, 11) pname
11              format ('Person "', a,
     1                        '" not updated. Abort all updates?')
                exec frs prompt (:msgbuf, :respbuf)
                if ((respbuf(1:1) .eq. 'y') .or.
     1               (respbuf(1:1) .eq. 'y')) then
```

```
                     exec sql rollback to savept
                     xaborted = .true.
                     exec frs endloop
                endif
           endif

      exec frs end

      if (.not. xaborted) then
C Commit the updates
            exec sql commit
      endif

C Terminate the FORMS and Ingres
      exec frs endforms
      exec sql disconnect

        if (updaterr) then
            print *, 'Your updates were aborted because of error:'
            print *, msgbuf
        endif

        end

C
C Function:   LoadTable
C Purpose:      Load the table field from the 'person' table. The
C             columns 'name' and 'age' will be displayed, and
C             'number' will be hidden.
C Parameters: None
C Returns:    Maximum employee number
C

      integer function LoadTable()

      exec sql include sqlca
C
C Declare person information:
C The preprocessor already knows that these variables have been
C declared, from their declarations in the main program.
C
      character*20 pname
      integer      page
      integer      pnum

C Max person id number to return
      integer maxid

      exec sql declare loadtab cursor for
     1    select name, age, number
     2    from person

C Set up error handling for loading procedure
C Close loadtab
      exec sql whenever sqlerror go to 100
C Close loadtab
      exec sql whenever not found go to 100

      exec frs message 'Loading Person Information . . .'

      maxid = 0

C Fetch the maximum person id number for later use
      exec sql select max(number)
     1      into :maxid
     2      from person
```

```
      exec sql open loadtab

55    if (sqlcod .ne. 0) go to 555

C Fetch data into record and load table field
      exec sql fetch loadtab into :pname, :page, :pnum

      exec frs loadtable personfrm persontbl
     1      (name = :pname, age = :page, number = :pnum)

      go to 55

555   continue

      exec sql whenever sqlerror continue

100   exec sql close loadtab
      LoadTable = maxid
      end
```

The Professor-Student Mixed Form Application

This application lets the user browse and update information about graduate students who report to a specific professor. The program is structured in a master/detail fashion, with the professor being the master entry, and the students the detail entries. The application uses two forms—one to contain general professor information and another for detailed student information.

The objects used in this application are shown in the following table:

Object	Description
personnel	The program's database environment.
professor	A database table with two columns: pname (**char(25)**) pdept (**char(10)**). See its **declare table** statement in the program for a full description.

Object	Description
student	A database table with seven columns: sname (**char(25)**) sage (**integer1**) sbdate (**char(25)**) sgpa (**float4**) sidno (**integer**) scomment (**var**char(200) sadvisor (**char(25)**) See its **declare table** statement for a full description. The "sadvisor" column is the join field with the "pname" column in the "professor" table.
masterfrm	The main form has the "pname" and "pdept" fields, which correspond to the information in the "professor" table, and "studenttbl" table field. The "pdept" field is display-only.
studenttbl	A table field in "masterfrm" with the "sname" and "sage" columns. When initialized, it also has five hidden columns corresponding to information in the "student" table.
studentfrm	The detail form, with seven fields, which correspond to information in the "student" table. Only the "sgpa", "scomment", and "sadvisor" fields are updatable. All other fields are display-only.
grad	A Fortran common area, whose fields correspond in name and type to the columns of the "student" database table, the "studentfrm" form and the "studenttbl" table field.

The program uses the "masterfrm" as the general-level master entry, in which you can only retrieve and browse data, and the "studentfrm" as the detailed screen, in which you can update specific student information.

Enter a name in the pname field and then select the Students menu operation. The operation fills the studenttbl table field with detailed information of the students reporting to the named professor. This is done by the studentcsr database cursor in the LoadStudents procedure.

The program assumes that each professor is associated with exactly one department. You can then browse the table field (in **read** mode), which displays only the names and ages of the students. You can request more information about a specific student by selecting the Zoom menu operation. This operation displays the studentfrm form (in **update** mode). The fields of “studentfrm” are filled with values stored in the hidden columns of “studenttbl”. You can make changes to three fields (“sgpa”, “scomment”, and “sadvisor”). If validated, these changes are written back to the Database table (based on the unique student ID), and to the table field’s data set. You can repeat this process for different professor names.

Note: Records can be used in this application but variables must be used with F77.

The application runs in UNIX, VMS, and Windows environments.

```
C
C Program: ProfessorStudent
C Purpose: Main entry point into "Professor-Student" mixed-form
C            master-detail application.
C
      program ProfessorStudent

      exec sql include sqlca

C Graduate student table
      exec sql declare student table
     1    (sname     char(25),
     2     sage      integer1,
     3     sbdate    char(25),
     4     sgpa      float4,
     5     sidno     integer4,
     6     scomment  char(200),
     7     sadvisor  char(25))

C Professor table
      exec sql declare professor table
     1     (pname    char(25),
     2      pdept    char(10))

      exec sql begin declare section

C Externally compiled master and student form
           integer masterfrm, studentfrm

      exec sql end declare section

      external masterfrm, studentfrm

C Start up Ingres and the FORMS system
      exec frs forms

      exec sql whenever sqlerror stop
      exec frs message 'Initializing Student Administrator . . .'
      exec sql connect personnel

      exec frs addform :masterfrm
      exec frs addform :studentfrm

      call Master
```

```
      exec frs clear screen
      exec frs endforms
      exec sql disconnect

      end

C
C Subroutine: Master
C Purpose:    Drive the application, by running "masterfrm" and
C              allowing the user to "zoom" into a selected student.
C Parameters:
C              None - Uses the global student "grad" common area.
C

      subroutine Master

      exec sql include sqlca

      exec sql begin declare section

C Global grad common area maps to database table
            character*25    sname
            integer*2       sage
            character*25    sbdate
            real            sgpa
            integer         sidno
            character*200   scomment
            character*25    sadvisor

C Professor info maps to database table
            character*25 pname
            character*10 pdept

C Useful forms system information
C Lastrow in table field
            integer lastrow
C Is a table field?
            integer istable

C Local utility buffers
C Message buffer
            character*100 msgbuf
C Response buffer
            character respbuf
C Old advisor before ZOOM
            character*25 oldavisor

      exec sql end declare section

C Make definition global
      common /grad/ sgpa, sidno, sage, sname, sbdate, scomment,
     1     sadvisor

C Function defined below
      logical StudentInfoChanged
```

```
C
C Initialize "studenttbl" with a data set in READ mode.
C Declare hidden columns for all the extra fields that
C the program will display when more information is
C requested about a student. Columns "sname" and "sage"
C are displayed. All other columns are hidden, to be
C used in the student information form.
C
      exec frs inittable masterfrm studenttbl read
     1    (sbdate = char(25),
     2     sgpa   = float4,
     3     sidno    = integer4,
     4     scomment = char(200),
     5     sadvisor = char(25))

      exec frs display masterfrm update

      exec frs initialize
      exec frs begin
          exec frs message 'Enter an Advisor name . . .'
          exec frs sleep 2
      exec frs end

      exec frs activate menuitem 'Students', field 'pname'
      exec frs begin

C Load the students of the specified professor
           exec frs getform (:pname = pname)

C If no professor name is given, resume
           if (pname(1:1) .eq.' ') then
               exec frs resume field pname
           endif

C
C Verify that the professor exists. Local error
C handling just prints the message and continues.
C Assume that each professor has exactly one
C department.
C
           exec sql whenever sqlerror call sqlprint
           exec sql whenever not found continue
           pdept = ' '
           exec sql select pdept
     1          into :pdept
     2          from professor
     3          where pname = :pname

           if (pdept(1:1) .eq.' ') then

                write (msgbuf, 10) pname
10              format ('No professor with name "', a,
     1                   '" [Press RETURN]')
                exec frs prompt noecho (:msgbuf, :respbuf)
                exec frs clear field all
                exec frs resume field pname

           endif

C Fill the department field and load students
           exec frs putform (pdept = :pdept)
C Refresh for query
           exec frs redisplay

           call loadstudents(pname)
```

```
           exec frs resume field studenttbl

      exec frs end

      exec frs activate menuitem 'Zoom'
      exec frs begin

C
C Confirm that user is in "studenttbl" and that
C the table field is not empty. Collect data from
C the row and zoom for browsing and updating.
C
           exec frs inquire_frs field masterfrm
     1         (:istable = table)

           if (istable .eq. 0) then
               exec frs prompt noecho
     1             ('Select from the student table [Press RETURN]',
     2               :respbuf)
               exec frs resume field studenttbl
           endif

           exec frs inquire_frs table masterfrm
     1          (:lastrow = lastrow)

           if (lastrow .eq. 0) then
                exec frs prompt noecho
     1              ('There are no students [Press RETURN]',
     2                :respbuf)
                exec frs resume field pname
           endif

C Collect all data on student into global record
           exec frs getrow masterfrm studenttbl
     1          (:sname = sname,
     2           :sage = sage,
     3           :sbdate = sbdate,
     4           :sgpa = sgpa,
     5           :sidno = sidno,
     6           :scomment = scomment,
     7           :sadvisor = sadvisor)

C
C Display "studentfrm", and if any changes were made,
C make the updates to the local table field row.
C Only make updates to the columns corresponding to
C writable fields in "studentfrm". If the student
C changed advisors, then delete the row from the
C display.
C
           oldadvisor = sadvisor
           if (StudentInfoChanged()) then

                if (oldadvisor .ne. sadvisor) then
                      exec frs deleterow masterfrm studenttbl
                else
                      exec frs putrow masterfrm studenttbl
     1                   (sgpa     = :sgpa,
     2                    scomment = :scomment,
     3                   sadvisor = :sadvisor)
                endif
           endif

      exec frs end
```

```
      exec frs activate menuitem 'Exit'
      exec frs begin
             exec frs breakdisplay
      exec frs end

      exec frs finalize

      end

C
C Subroutine:  LoadStudents
C Purpose:     Given an advisor name, load into the "studenttbl"
C              table field all the students who report to the
C              professor with that name.
C Parameters:
C              advisor - User-specified professor name.
C              Uses the global student record.
C

      subroutine LoadStudents(advisor)

      exec sql include sqlca

      exec sql begin declare section
             character*(*) advisor
      exec sql end declare section

C Global "grad" common fields
          character*25     sname
          integer*2        sage
          character*25     sbdate
          real             sgpa
          integer          sidno
          character*200    scomment
          character*25     sadvisor

      common /grad/ sgpa, sidno, sage, sname, sbdate, scomment,
     1           sadvisor

       exec sql declare studentcsr cursor for
     1      select sname, sage, sbdate, sgpa,
     2          sidno, scomment, sadvisor
     3          from student
     4          where sadvisor = :advisor

C
C Clear previous contents of table field. Load the table
C field from the database table based on the advisor name.
C Columns "sname" and "sage" will be displayed, and all
C others will be hidden.
C
      exec frs message 'Retrieving Student Information . . .'

      exec frs clear field studenttbl

C End loading
      exec sql whenever sqlerror go to 100
      exec sql whenever not found go to 100

      exec sql open studentcsr

C
C Before we start the loop, we know that the OPEN was
C successful and that NOT FOUND was not set.
C
55    if (sqlcod .ne. 0) go to 555
```

```
        exec sql fetch studentcsr into :sname, :sage, :sbdate,
       1     sgpa, :sidno, :scomment, :sadvisor

        exec frs loadtable masterfrm studenttbl
      1   (sname    = :sname,
      2    sage     = :sage,
      3    sbdate   = :sbdate,
      4    sgpa     = :sgpa,
      5    sidno    = :sidno,
      6    scomment = :scomment,
      7    sadvisor = :sadvisor)

       go to 55

555    continue

C Clean up on an error, and close cursors
       exec sql whenever not found continue

100    exec sql whenever sqlerror continue
       exec sql close studentcsr

       end

C
C Function: StudentInfoChanged
C Purpose:  Allow the user to zoom into the details of a selected
C            student. Some of the data can be updated by the user.
C            If any updates were made, then reflect these back into
C            the database table. The procedure returns TRUE if any
C            changes were made.
C Parameters:
C            None - Uses data in the global "grad" common area.
C Returns:
C           true/false - Changes were made to the database.
C           Sets the global "grad" common area with the new data.
C
       logical function StudentInfoChanged()

       exec sql include sqlca

       exec sql begin declare section
C Changes made to data in form
           integer changed
C Valid advisor name?
           integer validadvisor
       exec sql end declare section

C Global "grad" common fields
       character*25   sname
       integer*2      sage
       character*25   sbdate
       real           sgpa
       integer        sidno
       character*200  scomment
       character*25   sadvisor

       common /grad/ sgpa, sidno, sage, sname, sbdate, scomment,
      1      sadvisor

C Local error handler just prints error and continues
       exec sql whenever sqlerror call sqlprint
       exec sql whenever not found continue
```

```
      exec frs display studentfrm fill
      exec frs initialize
     1     (sname = :sname,
     2      sage = :sage,
     3      sbdate = :sbdate,
     4      sgpa = :sgpa,
     5      sidno = :sidno,
     6      scomment = :scomment,
     7      sadvisor = :sadvisor)

      exec frs activate menuitem 'Write'
      exec frs begin

C
C If changes were made, then update the database
C table. Only bother with the fields that are not
C read-only.
C
           exec frs inquire_frs form (:changed = change)

           if (changed .eq. 1) then

                exec frs validate

                exec frs getform
     1               (:sgpa = sgpa,
     2                :scomment = scomment,
     3                :sadvisor = sadvisor)

C Enforce integrity of professor name
                validadvisor = 0
                exec sql select 1 into :validadvisor
     1               from professor
     2               where pname = :sadvisor

                if (validadvisor .eq. 0) then
                     exec frs message 'Not a valid advisor name'
                     exec frs sleep 2
                     exec frs resume field sadvisor
                endif

                exec frs message 'Writing changes to database. . .'
                exec sql update student set
     1                sgpa = :sgpa,
     2                scomment = :scomment,
     3                sadvisor = :sadvisor
     4                where sidno = :sidno

           endif
           exec frs breakdisplay
      exec frs end

      exec frs activate menuitem 'Quit'
      exec frs begin
C Quit without submitting changes
           changed = 0
           exec frs breakdisplay
      exec frs end

      exec frs finalize

      StudentInfoChanged = (changed .EQ. 1)

      end
```

The SQL Terminal Monitor Application

This application executes SQL statements that are read in from the terminal. The application reads statements from input and writes results to output. Dynamic SQL is used to process and execute the statements.

When the application starts, it prompts the user for the database name. The program then prompts for an SQL statement. Each SQL statement can continue over multiple lines, and must end with a semicolon. No SQL comments are accepted. The SQL statement is processed using Dynamic SQL, and results and SQL errors are written to output. At the end of the results, the program displays an indicator of the number of rows affected. The loop is then continued and the program prompts the user for another SQL statement. When the user types in end-of-file, the application rolls back any pending updates and disconnects from the database.

The user's SQL statement is prepared using **prepare** and **describe**. If the SQL statement is not a **select** statement, then it is run using **execute** and the number of rows affected is printed. If the SQL statement is a **select** statement, a Dynamic SQL cursor is opened, and all the rows are fetched and printed. The routines that print the results do not try to tabulate the results. A row of column names is printed, followed by each row of the results.

Keyboard interrupts are not handled. Fatal errors, such as allocation errors, and boundary condition violations are handled by rolling back pending updates and disconnecting from the database session.

Note: Use your system function to obtain the address.

The application runs in UNIX, VMS, and Windows environments.

```
C
C Program: SQL_Monitor
C Purpose: Main entry of SQL Monitor application. Prompt for
C          database name and connect to the database. Run the
C          monitor and disconnect from the database. Before
C          disconnecting roll back any pending updates.
C
C Note:   UNIX compiler will generate - “Warning: %LOC function
C         treated as LOC.”
C         This is for compatibility with VMS. Just ignore the
C         message or change %LOC to LOC.
C

      program SQL_Monitor

      exec sql include sqlca

      exec sql begin declare section
           character*50   dbname
      exec sql end declare section
```

```
C    Prompt for database name.
      write   (*, 50)
50    format (' SQL Database: ', $)
      read (*, 51, err = 59, end = 59) dbname
51    format (A)

      print *, ' -- SQL Terminal Monitor --'

C     Treat connection errors as fatal.
      exec sql whenever sqlerror stop
      exec sql connect :dbname

      call Run_Monitor()

      exec sql whenever sqlerror continue

      print *, 'SQL: Exiting monitor program.'

      exec sql rollback
      exec sql disconnect
59    end

C
C Subroutine:Run_Monitor
C Purpose:   Run the SQL monitor. Initialize the global SQLDA with
C            the number of SQLVAR elements. Loop while prompting
C            the user for input; if end-of-file is detected then
C            return to the main program.
C
C            If the statement is not a SELECT statement then
C            EXECUTE it, otherwise open a cursor a process a
C            dynamic SELECT statement (using Execute_Select).
C

      subroutine Run_Monitor

C     Declare the SQLCA and the SQLDA structure definition
      exec sql include sqlca
      exec sql include sqlda

      exec sql begin declare section
          character*1000  stmt_buf

      exec sql end declare section
      record /IISQLDA/ sqlda
      common /sqlda_area/ sqlda

      integer  stmt_num

      integer      rows
      logical      Read_Stmt
      integer      Execute_Select
      exec sql declare stmt statement

C     Initialize the SQLDA
      sqlda.sqln = IISQ_MAX_COLS

C     Now we are set for input
      stmt_num = 0
      do while (.TRUE.)

           stmt_num = stmt_num + 1
```

```
C
C     Prompt and read the next statement. If Read_Stmt
C     returns FALSE then end-of-file was detected.
C
           if (.not. Read_Stmt(stmt_num, stmt_buf)) return

C     Handle database errors.
           exec sql whenever sqlerror goto 62

C
C     Prepare and describe the statement. If the statement is not
C     a SELECT then EXECUTE it, otherwise inspect the contents of
C     the SQLDA and call Execute_Select.
C
           exec sql prepare stmt from :stmt_buf
           exec sql describe stmt into :sqlda

C     If SQLD = 0 then this is not a SELECT.
           if (sqlda.sqld .eq. 0) then
                exec sql execute stmt
                rows = sqlerr(3)

           else

C        Are there enough result variables
                if (sqlda.sqld .le. sqlda.sqln) then
                     rows = Execute_Select()
                else

                     write(*, 60) sqlda.sqld, sqlda.sqln
60                   format (' SQL Error: SQLDA requires ', I3,
     1                        ' variables, but has only ', I3 '.')
                     rows = 0
                end if

           end if

C     Print number of rows processed.
           write (*, 61) rows
61         format (' [', I6, ' row(s)]')

           exec sql whenever sqlerror continue
C    If we got here because of an error then print the error
C    message.
62         if (sqlcod .lt. 0) call Print_Error()
      end do
      return
      end

C
C Function: Execute_Select
C Purpose:  In a dynamic SELECT statement. The SQLDA has already
C           been described, so print the column header (names),
C           open a cursor and retrieve and print the results.
C           Accumulate the number of rows processed.
C Parameters:
C            None
C Returns:
C            Number of rows processed.
C

      integer function Execute_Select()
```

```
      exec sql include sqlca
      exec sql include sqlda
      record /IISQLDA/ sqlda
      common /sqlda_area/ sqlda

      integerrows
      logical Print_Header

      exec sql declare csr cursor for stmt

C
C  Print result column names, set up the result types and
C  variables. Print_Header returns FALSE if the dynamic
C  set-up failed.
C
      if (.not. Print_Header()) then
           Execute_Select = 0
           return
      end if

      exec sql whenever sqlerror goto 70

C  Open the dynamic cursor.
      exec sql open csr for readonly

C  Fetch and print each row.
      rows = 0

      do while (sqlcod .eq. 0)

           exec sql fetch csr using descriptor :sqlda
           if (sqlcod .eq. 0) then
                rows = rows + 1
                call Print_Row()
           end if

      end do

C   If we got here because of an error then print the error
C   message.
70    if (sqlcod .lt. 0) call Print_Error()

      exec sql whenever sqlerror continue
      exec sql close csr

      Execute_Select = rows
      return
      end

C
C Function: Print_Header
C Purpose:  A statement has just been described so set up the SQLDA
C           for result processing. Print all the column names and
C           allocate result variables for retrieving data. The
C           result variables are chosen out of a pool of variables
C           (integers, floats and a large character string space).
C           The SQLDATA and SQLIND fields are pointed at the
C           addresses of the result variables.
C Returns:
C           TRUE if successfully set up the SQLDA for
C           result variables,
C           FALSE if an error occurred.
C

      logical function Print_Header ()
```

```
      exec sql include sqlda
      record /IISQLDA/ sqlda
      common /sqlda_area/ sqlda

C  User defined handler for large objects
      external UsrDataHandler
      integer  UsrDataHandler

C  Limit the size of a large object
C  If you increase BLOB_MAX than increase hdlarg.argstr
C  and 'segbuf'
       parameter (BLOB_MAX = 50)

       record /IISQLHDLR/ datahdlr(IISQ_MAX_COLS)

C     Global result data storage
      structure      /hdlr_arg/
          character*50 argstr
          integer arglen
      end structure
      record /hdlr_arg/ hdlarg(IISQ_MAX_COLS)

      integer*4         integers(IISQ_MAX_COLS)
      real*8            reals(IISQ_MAX_COLS)
      integer*2         inds(IISQ_MAX_COLS)
      character*2500    characters
      character*3000    disp_results
      common /result_area/ integers, reals, inds, characters,
     1      disp_results

      integer          cl
      integer          clc
      integer          dl
      character*2000   names
      integer          nl
      integer          nlc
      integer          i
      integer          base_type
      logical          is_null

C
C Add the name and number of each column into a column name buffer.
C Display this buffer as a header when done with all the columns.
C While processing each column determine the type of the column
C and to where SQLDATA must point in order to retrieve compatible
C results.
C
      cl = 1
      nl = 1
      dl = 0
      do 85, i = 1, sqlda.sqld
C
C  Fill up the names buffer. If it overflows print an error and
C  return that we failed.
C
      if (nl .gt. (len(names) - 40)) then
          print *, 'SQL Error: Result column name overflow.'
          Print_Header = .false.
          return
      end if
```

```
C
C  Store column title in the form "[column #] column_name "
C  For example, the employee table may look like:
C     [1] name [ 2] age [ 3] salary [ 4] dept
C
      write (names(nl:),80)i
80    format ('[', I3, '] ')
      nl = nl + 6
      nlc = sqlda.sqlvar(i).sqlname.sqlnamel
      names(nl:nl+nlc) = sqlda.sqlvar(i).sqlname.sqlnamec(1:nlc)
      nl = nl + nlc
      names(nl:nl) = ' '
      nl = nl + 1

C
C  At this point we've stored away the column name. Now we
C  process the column for type and length information. Use the
C  global numeric array and the large character buffer from which
C  pieces can be allocated.
C
C  Also accumulate the length of the display buffer that we will
C  need later to print the results - they will all be converted
C  into character data in the display buffer. Make sure that
C  the default field widths of the different types will fit into
C  the buffer 'disp_results'. For example, the display buffer for
C  a single row of the employee table may look like:
C  [ 1] mark [ 2] 36 [ 3] 52000.0 [ 4] eng
C

      dl = dl + 7

C  Find the base-type of the result (non-nullable).
      if (sqlda.sqlvar(i).sqltype .gt. 0) then
          base_type = sqlda.sqlvar(i).sqltype
          is_null = .false.
      else
          base_type = -sqlda.sqlvar(i).sqltype
          is_null = .true.
      end if

C
C  Collapse all different types into one of INTEGER, REAL
C  or CHARACTER. Accumulate the number of characters required
C  from the display buffer (use default format lengths).
C
      if (base_type .eq. IISQ_INT_TYPE) then

         sqlda.sqlvar(i).sqltype = IISQ_INT_TYPE
         sqlda.sqlvar(i).sqllen = 4
         sqlda.sqlvar(i).sqldata = %loc(integers(i))
         dl = dl + 12

      else if ((base_type .eq. IISQ_FLT_TYPE) .or.
     1          (base_type .eq. IISQ_DEC_TYPE)    .or.
     2          (base_type .eq. IISQ_MNY_TYPE)) then

         sqlda.sqlvar(i).sqltype = IISQ_FLT_TYPE
         sqlda.sqlvar(i).sqllen = 8
         sqlda.sqlvar(i).sqldata = %loc(reals(i))
         dl = dl + 25

      else if ((base_type .eq. IISQ_CHA_TYPE) .or.
     1          (base_type .eq. IISQ_VCH_TYPE) .or.
     2          (base_type .eq. IISQ_DTE_TYPE)) then
```

```
C
C Determine the length of the sub-string required from the
C the large character array. If we have enough space left
C then point at the start of the corresponding substring,
C otherwise print an error and return.
C
           if (base_type .eq. IISQ_DTE_TYPE) then
              clc = 25
           else
              clc = sqlda.sqlvar(i).sqllen
           end if

           if ((cl + clc) .gt. len(characters)) then
                write (*, 81) cl+clc
81            format (' SQL Error: Character result data overflow. '
     1                   'Need ', I4, ' bytes.')
                Print_Header = .false.
                return
           end if

C       Grab space out of the large character buffer
           sqlda.sqlvar(i).sqltype = IISQ_CHA_TYPE
           sqlda.sqlvar(i).sqllen = clc
           sqlda.sqlvar(i).sqldata = %loc(characters(cl:))
           cl = cl + clc
           dl = dl + clc
      else if (base_type .eq. IISQ_LVCH_TYPE) then
C
C Long varchar, so use datahandler. Use Blob Max to limit the
C length of the Blob sub-string returned/displayed.
C
           sqlda.sqlvar(i).sqltype = IISQ_HDLR_TYPE
           sqlda.sqlvar(i).sqllen = BLOB_MAX
           sqlda.sqlvar(i).sqldata = %loc(datahdlr(i))

           datahdlr(i).sqlhdlr = %loc(UserDataHandler)
           datahdlr(i).sqlarg = %loc(hdlrag(i))

           hdlarg(i).arglen = BLOB_MAX

           d1 = d1 + BLOB_MAX
      end if

C Remember to save the null indicator
      if (is_null) then
           sqlda.sqlvar(i).sqltype = -sqlda.sqlvar(i).sqltype
           sqlda.sqlvar(i).sqlind = %loc(inds(i))
      else
           sqlda.sqlvar(i).sqlind = 0
      end if

85    continue

C
C  Print all the saved column names. This loop does not use any
C  special formats, but just breaks the line at column 75.
C
      nl = nl - 1
      do 88 i = 1, nl , 75
           write (*, 87) names(i:min(i+74,nl))
87         format (' ', A)
88    continue
      print *, '--------------------------------'
```

```
C
C   Confirm that the character representation of the results
C   will fit inside the display buffer.
C
      if (dl .gt. len(disp_results)) then
           write (*, 81) dl
89         format (' SQL Error: Result display buffer overflow. '
     1                'Need ', I4, ' bytes.')
           Print_Header = .false.
           return
      end if

      Print_Header = .true.
      return

      end
C
C Procedure:Print_Row
C Purpose:  For each element inside the SQLDA, print the value.
C           Print its column number too in order to identify it
C           with a column name printed earlier in Print_Header. If
C           the value is NULL print "N/A".
C Parameters:
C           None
C

      subroutine Print_Row
      exec sql include sqlda
      record /IISQLDA/ sqlda
      common /sqlda_area/ sqlda

C  Global result data storage
      structure     /hdlr_arg/
         character*50   argstr
         integer        arglen
      end structure
      record /hdlr_arg/ hdlarg(IISQ_MAX_COLS)

      integer*4     integers(IISQ_MAX_COLS)
      real*8        reals(IISQ_MAX_COLS)
      integer*2     inds(IISQ_MAX_COLS)
      character*2500     characters
      character*3000     disp_results
      common /result_area/ integers, reals, inds, characters,
     1    disp_results, hdlarg

      integer      cl
      integer      clc

      integer      dl

      integer      i
      integer      base_type
      logical      is_null

C
C  For each column, print the column number and the data.
C  NULL columns print as "N/A". The printing is done by
C  encoding the complete row into a display buffer (that is
C  already guaranteed to be able to contain the whole row),
C  and then displaying the data at the end of the row.
C
      cl = 1
      dl = 1
      do 95, i = 1, sqlda.sqld
```

```
C Store result column number in the form "[ # ]"
           write(disp_results(dl:),90)i
90         format ('[', I3, '] ')
           dl = dl + 6

C  Find the base-type of the result (non-nullable)
           if (sqlda.sqlvar(i).sqltype .gt. 0) then
               base_type = sqlda.sqlvar(i).sqltype
               is_null = .false.
           else
               base_type = -sqlda.sqlvar(i).sqltype
               is_null = .true.
           end if

C
C Collapse different types into INTEGER, REAL or CHARACTER.
C If the data is NULL then just print "N/A".
C
           if (is_null .and. (inds(i) .eq. -1)) then

                disp_results(dl:dl+2) = 'N/A'
                dl = dl + 3

           else if (base_type .eq. IISQ_INT_TYPE) then

                write(disp_results(dl:),91)i
91              format (I)
                dl = dl + 12

           else if (base_type .eq. IISQ_FLT_TYPE) then

                write(disp_results(dl:),92)i
92              format (G)
                dl = dl + 25

           else if (base_type .eq. IISQ_CHA_TYPE) then

C    Use the characters out of the large character buffer
                clc = sqlda.sqlvar(i).sqllen
                disp_results(dl:dl+clc-1) = characters(cl:)
                dl = dl + clc
                cl = cl + clc
           else if (base_type .eq. IISQ_HDLR_TYPE) then
C  Use the argstr out of the handler structure buffer
                clc = sqlda.sqlvar(i).sqllen
                disp_results(d1:d1+clc-1) = hdlarg(i).argstr
                dl = dl + clc
           end if

           disp_results(dl:dl) = ' '
           dl = dl + 1

95     continue

      !
      ! Print the result data. This loop does not use any special
      ! formats, but just breaks the line at column 75.
      !
      dl = dl - 1
      do 98 i = 1, dl , 75
           write (*, 97) disp_results(i:min(i+74,dl))
97         format (' ', A)
98     continue
      return
      end
```

```
C
C Subroutine: Print_Error
C Purpose:    SQLCA error detected. Retrieve the error message
C             and print it.
C Parameters:
C             None
C
      subroutine Print_Error

      exec sql include sqlca
      exec sql begin declare section
           character*200        error_buf
      exec sql end declare section

      exec sql inquire_sql (:error_buf = ERRORTEXT)
      print *, 'SQL Error:'
      print *, error_buf
      return
      end

C
C Function:Read_Stmt
C Purpose: Reads a statement from standard input. This routine
C          prompts the user for input (using a statement
C          number) and scans input tokens for the statement
C          delimiter (semicolon). The scan continues
C          over new lines, and uses SQL string literal
C          rules.
C Parameters:
C            stmt_num - Statement number for prompt.
C            stmt_buf - Buffer to fill for input.
C Returns:
C           TRUE if a statement was read, FALSE if
C          end-of-file typed.
C

      integer function Read_Stmt(stmt_num, stmt_buf)

      integer         stmt_num
      character*(*)   stmt_buf

      integer      stmt_max
      integer      sl

      character   input_buf(256)
      integer     line_len
      integer     i

      logical     in_string
      logical     current_line

      stmt_max = len(stmt_buf)

C    Prompt user for SQL statement.
110   write (*, 111) stmt_num
111   format (' ', I3, ' ', $)

      stmt_buf = ' '
      in_string = .false.
      sl = 1

C   Loop while scanning input for statement terminator.
      do while (.TRUE.)
```

```
C   Read input line up to the number of characters entered
           read (*, 112, err = 119, end = 119) line_len,
     1          (input_buf(i), i = 1, line_len)
112        format (Q, 100A1)

           current_line = .true. ! We are in a line
C
C Keep processing while we can (we have not reached the end of
C the line, and our statement buffer is not full).
C
           i = 1
           do while (current_line .and. (sl .le. stmt_max))

C     Not in string - check for delimiters and new lines
                if (.not. in_string) then

                     if (i .gt. line_len) then
C                 New line outside of string is replaced with blank
                          input_buf(i) = ' '
                          current_line = .false.
                     else if (input_buf(i) .eq. ';') then
                          Read_Stmt = .true.
                          return
                     else if (input_buf(i) .eq. '''') then
                          in_string = .true.
                     end if
                     stmt_buf(sl:sl) = input_buf(i)
                     sl = sl + 1
                     i = i + 1

                else

C     End of line inside string is ignored
                     if (i .gt. line_len) then
                          current_line = .false.
                     else if (input_buf(i) .eq. '''') then
                          in_string = .false.
                     end if

                     if (current_line) then
                          stmt_buf(sl:sl) = input_buf(i)
                          sl = sl + 1
                          i = i + 1
                     end if

                end if
           end do
C
C Dropped out of above loop because end of line reached or buffer
C limit exceeded.
C

C Statement is too large - ignore it and try again.
           if (sl .gt. stmt_max) then
                write (*, 113) stmt_max
113             format (' SQL Error: Maximum statement length
     1                 (', I4,') exceeded.')
                goto 110
           else
                write (*, 114)
114             format (' ---> ', $)
           end if
```

```
      end do
119   Read_Stmt = .false.
      return
      end

C
C Procedure: UsrDataHandler
C Purpose:   Use GET DATA to get the BLOB from the database.
C Parameters:
C            hdlarg - the structure with handler info

      subroutine UsrDataHandler (hdlarg)

      exec sql include sqlca

      exec sql whenever sqlerror stop

      exec sql begin declare section
            structure /hdlr_arg/
                character*50 argstr
                 integer     arglen
            end structure
            record /hdlr_arg/ hdlarg

            character*50      segbuf
            integer*4         dataend
            integer*4         seglen
      exec sql end declare sections

      integer totlen
      integer nsegs

      if (hdlarg.arglen .gt. len(hdlarg.argstr)) then
          hdlarg.arglen = len(hdlarg.argstr)
          write (*,120) hdlarg.arglen
120       format ('BLOB length error....reducing to: ',I)
      end if

      nsegs = 0
      totlen = 0
      dataend = 0
      do while ((dataend .eq. 0) .and. (totlen .lt. hdlarg.arglen))
           segbuf= ' '
           exec sql get data (:segbuf  = segment,
     1                        :seglen  = segmentlength,
     2                        :dataend = dataend)
     3          with maxlength = :hdlarg.arglen;

           hdlarg.argstr(totlen+1:) = segbuf(1:seglen)

           nsegs = nsegs + 1
           totlen = totlen + seglen
      end do

      if (dataend .eq. 0) then
           exec sql enddata;
      end if

      end
```

A Dynamic SQL/Forms Database Browser

This program lets the user browse data from and insert data into any table in any database, using a dynamically defined form. The program uses Dynamic SQL and Dynamic FRS statements to process the interactive data. You should already have used VIFRED to create a Default Form based on the database table that you want to browse. VIFRED will build a form with fields that have the same names and data types as the columns of the specified database table.

When run, the program prompts the user for the name of the database, the table and the form. The form is profiled using the **describe form** statement, and the field name, data type, and length information is processed. From this information, the program fills in the SQLDA data and null indicator areas, and builds two Dynamic SQL statement strings to **select** data from and **insert** data into the database.

The **Browse** menu item retrieves the data from the database using an SQL cursor associated with the dynamic **select** statement, and displays that data using the dynamic **putform** statement. A **submenu** allows the user to continue with the next row or return to the main menu. The **Insert** menu item retrieves the data from the form using the dynamic **getform** statement, and adds the data to the database table using a prepared **insert** statement. The **Save** menu item commits the changes and, because prepared statements are discarded, prepares the **select** and **insert** statements again. When the user selects the **Quit** menu item, all pending changes are rolled back and the program is terminated.

Note: Use your system function to obtain the address.

```
C
C Program: Dynamic_FRS
C Purpose: Main body of Dynamic SQL forms application. Prompt for
C          database, form and table name. Call Describe_Form
C          to obtain a profile of the form and set up the SQL
C          statements. Then allow the user to interactively browse
C          database table and append new data.
C

C
C Note: The UNIX compiler will generate - "Warning: %LOC function
C       treated as LOC". This is for compatibility with VMS.
C       Just ignore the message. Or Change %LOC to LOC.
C

      program Dynamic_FRS

C  Declare the SQLCA and the SQLDA
      exec sql include sqlca
      exec sql include sqlda

      record /IISQLDA/ sqlda
      common /sqlda_are/ sqlda

      exec sql declare sel_stmt statement
      exec sql declare ins_stmt statement
      exec sql declare csr cursor for sel_stmt
```

```
      logical Describe_Form

      exec sql begin declare section
           character*40       dbname
           character*40       formname
           character*40       tabname
           character*1000     sel_buf
           character*1000     ins_buf
           integer*4          err
           character*1        ret
      exec sql end declare section

      exec frs forms

C  Prompt for database name - will abort on errors
      exec sql whenever sqlerror stop
      exec frs prompt ('Database name: ', :dbname)
      exec sql connect :dbname

      exec sql whenever sqlerror call sqlprint

C
C  Prompt for table name - later a Dynamic SQL SELECT statement
C  will be built from it.
C
      exec frs prompt ('Table name: ', :tabname)

C
C  Prompt for form name. Check forms errors reported
C   through INQUIRE_FRS.
C
      exec frs prompt ('Form name: ', :formname)
      exec frs message 'Loading form ...'
      exec frs forminit :formname
      exec frs inquire_frs frs (:err = ERRORNO)
      if (err .gt. 0) then
          exec frs message 'Could not load form. Exiting.'
          exec frs endforms
          exec sql disconnect
          stop
      end if

C  Commit any work done so far - access of forms catalogs
      exec sql commit

C  Describe the form and build the SQL statement strings
      if (.not. Describe_Form(formname, tabname, sel_buf, ins_buf))
     1 then
           exec frs message 'Could not describe form. Exiting.'
           exec frs endforms
           exec sql disconnect
           stop
      end if
```

```
C
C  PREPARE the SELECT and INSERT statements that correspond to the
C  menu items Browse and Insert. If the Save menu item is chosen
C  the statements are reprepared.
C
      exec sql prepare sel_stmt from :sel_buf
      err = sqlcod
      exec sql prepare ins_stmt from :ins_buf
      if ((err .lt. 0) .or. (sqlcod .lt. 0)) then
       exec frs message 'Could not prepare SQL statements. Exiting'
          exec frs endforms
          exec sql disconnect
          stop
      end if

C
C  Display the form and interact with user, allowing browsing
C  and the inserting of new data.
C

      exec frs display :formname fill
      exec frs initialize
      exec frs activate menuitem 'Browse'
      exec frs begin
C
C  Retrieve data and display the first row on the form,
C  allowing the user to browse through successive rows. If
C  data types from the database table are not consistent
C  with data descriptions obtained from the form, a
C  retrieval error will occur. Inform the user of this or other
C  errors.
C
C  Note that the data will return sorted by the first field
C  that was described, as the SELECT statement, sel_stmt,
C  included an ORDER BY clause.
C
           exec sql open csr

C  Fetch and display each row
           do while (sqlcod .eq. 0)

                exec sql fetch csr using descriptor :sqlda
                if (sqlcod .ne. 0) then
                     exec sql close csr
                    exec frs prompt noecho ('No more rows :', :ret)
                     exec frs clear field all
                     exec frs resume
                end if

                exec frs putform :formname using descriptor :sqlda
                exec frs inquire_frs frs (:err = ERRORNO)
                if (err .gt. 0) then
                     exec sql close csr
                     exec frs resume
                end if

C  Display data before prompting user with submenu
                exec frs redisplay
```

```
               exec frs submenu
               exec frs activate menuitem 'Next', frskey4
               exec frs begin
C  Continue with cursor loop
                    exec frs message 'Next row ...'
                    exec frs clear field all
               exec frs end
               exec frs activate menuitem 'End', frskey3
               exec frs begin
                    exec sql close csr
                    exec frs clear field all
                    exec frs resume
               exec frs end

          end do

     exec frs end

     exec frs activate menuitem 'Insert'
     exec frs begin
          exec frs getform :formname using descriptor :sqlda
          exec frs inquire_frs frs (:err = ERRORNO)
          if (err .gt. 0) then
               exec frs clear field all
               exec frs resume
          end if
          exec sql execute ins_stmt using descriptor :sqlda
          if ((sqlcod .lt. 0) .or. (sqlerr(3) .eq. 0)) then
              exec frs prompt noecho ('No rows inserted :', :ret)
          else
              exec frs prompt noecho ('One row inserted :', :ret)
          end if
     exec frs end

     exec frs activate menuitem 'Save'
     exec frs begin
C
C  COMMIT any changes and then re-PREPARE the SELECT and INSERT
C  statements as the COMMIT statements discards them.
C
          exec sql commit
          exec sql prepare sel_stmt from :sel_buf
          err = sqlcod
          exec sql prepare ins_stmt from :ins_buf
          if ((err .lt. 0) .or. (sqlcod .lt. 0)) then
                exec frs prompt noecho
    1               ('Could not reprepare SQL statements :', :ret)
                exec frs breakdisplay
          end if
     exec frs end

     exec frs activate menuitem 'Clear'
     exec frs begin
          exec frs clear field all
     exec frs end

     exec frs activate menuitem 'Quit', frskey2
     exec frs begin
          exec sql rollback
          exec frs breakdisplay
     exec frs end
     exec frs finalize

     exec frs endforms
     exec sql disconnect
```

```
      end

C
C Procedure: Describe_Form
C Purpose:   Profile the specified form for name and data type
C            information.
C            Using the DESCRIBE FORM statement, the SQLDA is loaded
C            with field information from the form. This procedure
C            processes this information to allocate result storage,
C            point at storage C for dynamic FRS data retrieval and
C            assignment, and build C SQL statements strings for
C            subsequent dynamic SELECT and INSERT statements. For
C            example, assume the form (and table) 'emp' has the
C            following fields:
C
C             Field Name   Type       Nullable?
C             ----------   ----       ---------
C             name         char(10)   No
C             age          integer4   Yes
C             salary       money      Yes
C
C         Based on 'emp', this procedure will construct the SQLDA.
C         The procedure allocates variables from a result variable
C         pool (integers, floats and a large character string
C         space). The SQLDATA and SQLIND fields are pointed at the
C         addresses of the result variables in the pool. The
C         following SQLDA is built:
C
C                    sqlvar(1)
C                           sqltype   = IISQ_CHA_TYPE
C                           sqllen    = 10
C                           sqldata = pointer into characters array
C                           sqlind    = null
C                           sqlname   = 'name'
C                    sqlvar(2)
C                           sqltype   = -IISQ_INT_TYPE
C                           sqllen    = 4
C                           sqldata   = address of integers(2)
C                           sqlind    = address of indicators(2)
C                           sqlname   = 'age'
C                    sqlvar(3)
C                           sqltype   = -IISQ_FLT_TYPE
C                           sqllen    = 8
C                           sqldata   = address of floats(3)
C                           sqlind    = address of indicators(3)
C                           sqlname   = 'salary'
C
C       This procedure also builds two dynamic SQL statements
C       strings. Note that the procedure should be extended to
C       verify that the statement strings do fit into
C       the statement buffers (this was not done in this example).
C       The above example would construct the following
C       statement strings:
C
C       'SELECT name, age, salary FROM emp ORDER BY name'
C       'INSERT INTO emp (name, age, salary) VALUES (?, ?, ?)'
C
C Parameters:
C         formname - Name of form to profile.
C         tabname - Name of database table.
C         sel_buf - Buffer to hold SELECT statement string.
C         ins_buf - Buffer to hold INSERT statement string.
C Returns:
C         TRUE/FALSE - Success/failure - will fail on error
C                       or upon finding a table field.
C
```

```
      logical function
     1      Describe_Form (formname, tabname, sel_buf, ins_buf)

      character*(*) formname, tabname, sel_buf, ins_buf

C  Declare the SQLCA and the SQLDA
      exec sql include sqlca
      exec sql include sqlda
      record /IISQLDA/ sqlda
      common /sqlda_area/ sqlda

C  Global result data storage
      integer*4       integers(IISQ_MAX_COLS)
      real*8          reals(IISQ_MAX_COLS)
      integer*2       inds(IISQ_MAX_COLS)
      character*2500  characters
      common /result_area/ integers, reals, inds, characters

      integer        char_cnt
      integer        char_cur

      integer        i
      integer        base_type
      logical        nullable

      character*1000 names
      integer        name_cnt
      integer        name_cur
      character*1000 marks
      integer        mark_cnt

      integer*4      err
      character*     ret

C
C Initialize the SQLDA and DESCRIBE the form. If we cannot fully
C describe the form (our SQLDA is too small) then report an error
C and return.
C
      sqlda.sqln = IISQ_MAX_COLS

      exec frs describe form :formname all into :sqlda
      exec frs inquire_frs frs (:err = ERRORNO)
      if (err .gt. 0) then
           Describe_Form = .false.
           return
      end if
      if (sqlda.sqld .gt. sqlda.sqln) then
          exec frs prompt noecho ('SQLDA is too small for form :',
     1                             :ret)
          Describe_Form = .false.
          return
      else if (sqlda.sqld .eq. 0) then
           exec frs prompt noecho
     1                  ('There are no fields in the form :', :ret)
           Describe_Form = .false.
           return
      end if
```

```
C
C   For each field determine the size and type of the result data
C   area. This data area will be allocated out of the result
C   variable pool (integers, floats and characters) and will be
C   pointed at by SQLDATA and SQLIND.
C
C   If a table field type is returned then issue an error.
C
C   Also, for each field add the field name to the 'names' buffer
C   and the SQL place holders '?' to the 'marks' buffer, which
C   will be used to build the final SELECT and INSERT statements.
C

      char_cnt = 1
      name_cnt = 1
      mark_cnt = 1

      do 20, i = 1, sqlda.sqld

C  Find the base-type of the result (non-nullable).
              if (sqlda.sqlvar(i).sqltype .gt. 0) then
                   base_type = sqlda.sqlvar(i).sqltype
                   nullable = .false.
              else
                   base_type = -sqlda.sqlvar(i).sqltype
                   nullable = .true.
              end if

C
C  Collapse all different types into one of INTEGER, REAL
C  or CHARACTER. Figure out where to point SQLDATA and
C  SQLIND - which member of the result variable pool is
C  compatible with the data.
C
              if (base_type .eq. IISQ_INT_TYPE) then

                   sqlda.sqlvar(i).sqltype  = IISQ_INT_TYPE
                   sqlda.sqlvar(i).sqllen   = 4
                   sqlda.sqlvar(i).sqldata  = %loc(integers(i))

              else if ((base_type .eq. IISQ_FLT_TYPE) .or.
     1             (base_type .eq. IISQ_DEC_TYPE) .or.
     2             (base_type .eq. IISQ_MNY_TYPE)) then

                   sqlda.sqlvar(i).sqltype = IISQ_FLT_TYPE
                   sqlda.sqlvar(i).sqllen = 8
                   sqlda.sqlvar(i).sqldata = %loc(reals(i))

              else if ((base_type .eq. IISQ_CHA_TYPE) .or.
     1             (base_type .eq. IISQ_VCH_TYPE) .or.
     2             (base_type .eq. IISQ_DTE_TYPE)) then

C
C  Determine the length of the sub-string required from the
C  the large character array. If we have enough space left
C  then point at the start of the corresponding substring,
C  otherwise display an error and return.
C
                        if (base_type .eq. IISQ_DTE_TYPE) then
                             char_cur = IISQ_DTE_LEN
                        else
                             char_cur = sqlda.sqlvar(i).sqllen
                        end if
```

```
                    if ((char_cnt + char_cur) .gt. len(characters))
     1                    then
                          exec frs prompt noecho
     1                   ('Character pool buffer overflow :', :ret)
                          Describe_Form = .false.
                          return
                     end if

C
C  Grab space out of the large character buffer and accumulate used
C  characters.
C
                     sqlda.sqlvar(i).sqltype  = IISQ_CHA_TYPE
                     sqlda.sqlvar(i).sqllen   = char_cur
                     sqlda.sqlvar(i).sqldata
     1                                = %loc(characters(char_cnt:))
                     char_cnt                 = char_cnt + char_cur

                else if (base_type .eq. IISQ_TBL_TYPE) then

                     exec frs prompt noecho
     1                     ('Table field found in form :', :ret)
                     Describe_Form = .false.
                     return

                else

                     exec frs prompt noecho
     1                      ('Invalid field type :', :ret)
                     Describe_Form = .false.
                     return

                end if

C  Remember to save the null indicator
                if (nullable) then
                 sqlda.sqlvar(i).sqltype = -sqlda.sqlvar(i).sqltype
                     sqlda.sqlvar(i).sqlind = %loc(inds(i))
                else
                     sqlda.sqlvar(i).sqlind = 0
                end if

C
C  Store field names and place holders (separated by commas)
C  for the SQL statements.
C
                if (i .gt. 1) then
                     names(name_cnt:name_cnt) = ','
                     name_cnt = name_cnt + 1
                    marks(mark_cnt:mark_cnt) = ','
                    mark_cnt = mark_cnt + 1
               end if
               name_cur = sqlda.sqlvar(i).sqlname.sqlnamel
               names(name_cnt:name_cnt+name_cur) =
     1                sqlda.sqlvar(i).sqlname.sqlnamec(1:name_cur)
               name_cnt = name_cnt + name_cur
               marks(mark_cnt:mark_cnt) = '?'
               mark_cnt = mark_cnt + 1

20         continue
```

```
C
C  Create final SELECT and INSERT statements. For the SELECT
C  statement ORDER BY the first field.
C
      name_cur = sqlda.sqlvar(1).sqlname.sqlnamel
      sel_buf = 'select ' // names(1:name_cnt-1) // ' from '
     1              // tabname // ' order by '
     2              // sqlda.sqlvar(1).sqlname.sqlnamec(1:name_cur)
      ins_buf = 'insert into ' // tabname // ' ('
     1      // names(1:name_cnt-1) // ') values ('
     2      // marks(1:mark_cnt-1) // ')'

      Describe_Form = .true.
      return

      end
```

Chapter 5: Embedded SQL for Ada

This chapter describes the use of Embedded SQL with the Ada programming language.

Embedded SQL Statement Syntax for Ada

This section describes the language-specific issues inherent in embedding SQL database and forms statements in an Ada program. An Embedded SQL database statement has the following general syntax:

> [*margin*] **exec sql** *SQL_statement terminator*

The syntax of an Embedded SQL/FORMS statement is almost identical:

> [*margin*] **exec frs** *SQL/FORMS_statement terminator*

For information on SQL statements, see the *SQL Reference Guide*. For information on SQL/FORMS statements, see the *Forms-based Application Development Tools User Guide.*

The sections below describe the various syntactical elements of these statements as implemented in Ada.

Margin

There are no specified margins for Embedded SQL statements in Ada. The **exec** keyword can begin anywhere on the source line.

Terminator

The terminator for Ada is the semicolon (;). For example, a **select** statement embedded in an Ada program would look like:

```
exec sql select ename
        into :namevar
        from employee
        where eno = :numvar;
```

An Embedded SQL statement cannot be followed on the same line by another embedded statement or an Ada statement. Doing so will cause preprocessor syntax errors on the second statement. Following the Ada terminator, only comments and white space (blanks and tabs) are allowed to the end of the line.

Labels

Like Ada statements, Embedded SQL statements can have a label prefix. The label must begin with an alphabetic character, must be the first word on the line (optionally preceded by white space), and must be delimited with double angle brackets. For example:

```
<<close_cursor>> exec sql close cursor1;
```

The label can appear anywhere an Ada label can appear. Even though the preprocessor accepts the label in front of any **exec sql** or **exec frs** prefix, it may not be appropriate to code it on some lines. For example, the following, although acceptable to the preprocessor, later generates a compiler error because labels are not allowed before declarations:

```
<<include_sqlca>> exec sql include sqlca;
```

As a general rule, use labels only with executable statements.

Line Continuation

There are no line continuation rules for Embedded SQL statements in Ada. Statements can continue across multiple lines, extending to the Ada terminator. You can also include blank lines.

Comments

You can include Ada comments delimited by "--" and extending to the end of the line, anywhere in an Embedded SQL statement that a line break is allowed, with the following exceptions:

- In string constants.
- In parts of statements that are dynamically defined. For example, a comment in a string variable specifying a form name is interpreted as part of the form name and causes a runtime syntax error.
- Between component lines of Embedded SQL block-type statements. All block-type statements (such as **activate** and **unloadtable**) are compound statements that include a statement section delimited by **begin** and **end**. Comment lines must not appear between the statement and its section. The preprocessor interprets such comments as Ada host code, which causes preprocessor syntax errors. (Note, however, that comments can appear on the same line as the statement.) For example, the following statement causes a syntax error on the first Ada comment:

```
exec frs unloadtable empform
        employee (:namevar = ename);
 -- Illegal comment before statement body
exec frs begin; -- comment legal here
        msgbuf := namevar;
 exec frs end;
```

- Statements made up of more than one compound statement, such as the **display** statement, which typically consists of the **display** clause, an **initialize** section, **activate** sections, and a **finalize** section, cannot have Ada comments between any of the components. These comments are translated as host code and cause syntax errors on subsequent statement components.

String Literals

Embedded SQL string literals are delimited by single quotes. To embed a single quote in a string literal, precede it with another single quote character, as in:

```
exec sql insert
        into comments (id, val)
        values (15, 'This is ''Student'' information');
```

Because the single quote is the SQL string delimiter, Ada single-character literals are treated like SQL string literals. Embedded SQL/Ada string literals cannot be continued over multiple lines.

Note that the preprocessor does not accept the Ada character string delimiter, the double quote ("). No special characters are required to embed a double quote in an Embedded SQL string literal.

String Literals and Statement Strings

The Dynamic SQL statements **prepare** and **execute immediate** both use statement strings that specify an SQL statement. The statement string can be specified by a string literal or character string variable, as in:

```
exec sql execute immediate 'drop employee';
```

or:

```
str := "drop employee";
exec sql execute immediate :str;
```

As with regular Embedded SQL string literals, the statement string delimiter is the single quote. However, single quotes embedded in statement strings must conform to the *runtime* rules of SQL when the statement is executed. For example, the following two dynamic **insert** statements are equivalent:

```
exec sql prepare s1 from
     'insert into t1 values (''single''''double" '');
```

and:

```
str := "insert into t1 values ('single'' double"" ')";
exec sql prepare s1 from :str;
```

In fact, the string literal generated by the Embedded SQL/Ada preprocessor for the first example is identical to the string literal assigned to the variable "str" in the second example.

The runtime evaluation of the above statement string is:

```
insert into t1 values ('single''double" ')
```

As a general rule it is best to avoid using a string literal for a statement string whenever it may contain quotes. Instead you should build the statement string using the Ada language rules for string literals together with the SQL rules for the runtime evaluation of the string.

The Create Procedure Statement

As mentioned in the *SQL Reference Guide*, the **create procedure** statement has language-specific syntax rules for line continuation, string literal continuation, comments, and the final terminator. These syntax rules follow the rules discussed in this chapter. For example, the final terminator is a semicolon (;). Although the preprocessor treats the **create procedure** statement as a single statement, you must terminate all statements in the body of the procedure with a semicolon as is an Embedded SQL/Ada statement.

The following example shows a **create procedure** statement that follows the Embedded SQL/Ada syntax rules:

```
exec sql
  create procedure proc (parm integer) as
  declare
    var integer;
  begin
    if parm > 10 then -- use Ada comment delimiter
      message 'Ada strings cannot continue over lines';
      insert into tab values (:parm);
    endif;
  end;
```

Ada Variables and Data Types

This section describes how to declare and use Ada program variables in Embedded SQL.

Embedded SQL/Ada Declarations

The following sections discuss syntax, types, and definitions of Embedded SQL/Ada declarations.

Embedded SQL Variable Declaration Sections

Embedded SQL statements use Ada variables to transfer data to and from the database or a form into the program. You must declare Ada variables and constants to Embedded SQL before using them in any Embedded SQL statements. Ada variables, types and constants are declared to Embedded SQL in a *declaration section*. This section has the following syntax:

> **exec sql begin declare section;**
> *Ada type and variable declarations*
> **exec sql end declare section;**

Note that placing a label in front of the **exec sql end declare section** statement causes a preprocessor syntax error.

Embedded SQL variable declarations are global to the program file from the point of declaration onwards. You can incorporate multiple declaration sections into a single program, as would be the case when a few different Ada procedures issue embedded statements using local variables. Each procedure can have its own declaration section. For a discussion of the declaration of variables that are local to Ada procedures, see The Scope of Variables in this chapter.

Reserved Words in Declarations

The following keywords are reserved by the Embedded SQL/Ada preprocessor. Therefore you cannot declare types or variables with the same name as these keywords:

access	**array**	**case**	**constant**	**delta**
digits	**end**	**for**	**function**	**is**
limited	**new**	**null**	**of**	**others**
package	**private**	**raise**	**range**	**record**
renames	**return**	**sql_standard**	**subtype**	**type**

Data Types and Constants

The Embedded SQL/Ada preprocessor defines certain data types and constants from the Ada STANDARD and SYSTEM packages. The table below maps the types to their corresponding Ingres type categories. For a description of the exact type mapping, see Data Type Conversion in this chapter.

Ada Data Types and Corresponding Ingres Types

Ada Type	Ingres Type
short_short_integer	integer
short_integer	integer
integer	integer
natural	integer
positive	integer
boolean	integer
float	float
long_float	float
f_float	float
d_float	float
character	character
string	character

None of the types listed above should be redefined by your program. If they are redefined, your program might not compile and will not work correctly at runtime.

The following table maps the Ada constants to their corresponding Ingres type categories.

Constants and Corresponding Ingres Types

Ada Constant	Ingres Type
max_int	**integer**
min_int	**integer**
true	**integer**
false	**integer**

Note that if the type or constant is derived from the SYSTEM package, the program unit must specify that the SYSTEM package should be included—Embedded SQL does not do so itself. You cannot refer to a SYSTEM object by using the package name as a prefix, because Embedded SQL does not allow this type of qualification. The types **f_float** and **d_float** and the constants **max_int** and **min_int** are derived from the SYSTEM package.

The Integer Data Type

All **integer** types and their derivatives are accepted by the preprocessor. Even though some integer types have Ada constraints, such as the types **natural** and **positive**, Embedded SQL does not check these constraints, either during preprocessing or at runtime. An **integer** constant is treated as an Embedded SQL constant value and cannot be the target of an Ingres assignment.

The type **boolean** is handled as a special type of **integer**. In Ada, the **boolean** type is defined as an enumerated type with enumerated literals **false** and **true**. Embedded SQL treats the **boolean** type as an enumerated type and generates the correct code in order to use this type to interact with an Ingres integer. Enumerated types are described in more detail later.

The Float Data Type

The preprocessor accepts four floating-point types. The types **float** and **f_float** are the 4-byte floating-point types. The types **long_float** and **d_float** are the 8-byte floating-point types. **Long_float** requires some extra definitions for default Ada pragmas to be able to interact with Ingres floating-point types. Note that the preprocessor does not accept the **long_long_float** and **h_float** data types.

The Long Float Storage Format

Ingres requires that the storage representation for long floating-point variables be **d_float**, because the Embedded SQL runtime system uses that format for floating-point conversions. If your Embedded SQL program has **long_float** variables that interact with the Embedded SQL runtime system, you must make sure they are stored in the **d_float** format. Floating-point values of types **g_float** and **h_float** are stored in different formats and sizes. The default Ada format is **g_float**; consequently, you must convert your long floating-point variables to type **d_float**. There are three methods you can use to ensure that the Ada compiler always uses the **d_float** format.

The first method is to issue the following Ada pragma before every compilation unit that declares **long_float** variables:

```
pragma long_float( d_float );
exec sql begin declare section;
         dbl: long_float;
exec sql end declare section;
```

Note that the **pragma** statement is not an Embedded SQL statement, but an Ada statement that directs the compiler to use a different storage format for **long_float** variables.

The second method is a more general instance of the first. If you are certain that all **long_float** variables in your Ada program library will use the **d_float** format, including those not interacting with Ingres, then you can install the pragma into the program library by issuing the following ACS command:

```
acs set pragma/long_float=d_float
```

This system-level command is equivalent to issuing the Ada **pragma** statement for each file that uses **long_float** variables.

The third method is to use the type **d_float** instead of the type **long_float**. This has the advantage of allowing you to mix both **d_float** and **g_float** storage formats in the same compilation unit. Of course, all Embedded SQL floating-point variables must be of the **d_float** type and format. For example:

```
exec sql begin declare section;
        d_dbl: d_float;
exec sql end declare section;

        g_dbl: g_float; -- Unknown to Embedded SQL
```

One side effect of all the above conversions is that some default system package instantiations for the type **long_float** become invalid because they are set up under the **g_float** format. For example, the package **long_float_text_io**, which is used to write long floating-point values to text files, must be reinstantiated. Assuming that you have issued the following ACS command on your program library:

```
acs set pragma/long_float=d_float
```

you must reinstantiate the **long_float_text_io** package before you can use it. A typical file might contain the following two lines, which serve to enter your own copy of **long_float_text_io** into your library:

```
with text_io;
package long_float_text_io is new
             text_io.float_io(long_float);
```

A later statement, such as:

```
with long_float_text_io; use long_float_text_io;
```

will pick up your new copy of the package, which is defined using the **d_float** internal storage format.

The Character and String Data Types

Both the **character** and **string** data types are compatible with Ingres string objects. By default, the **string** data type is an array of characters.

The **character** data type does have some restrictions. Because it must be compatible with Ingres string objects, you can use only a one-dimensional array of characters. Therefore, you cannot use a single character or a multi-dimensional array of characters as a Ingres string. Note that you *can* use a multi-dimensional array of strings. For example, the following four declarations are legal:

```
subtype Alphabet is Character range 'a'..'z';
type word_5 is array(1..5) of Character;
                                -- 1-dimensional array
word_6: String(1..6);           -- Default string type
word_arr: array(1..5) of String(1..6);
                                -- Array of strings
```

However, the declarations below are illegal because they violate the Embedded SQL restrictions for the **character** type. Although the declarations may not generate Embedded SQL errors, the Ada compiler does not accept the references when used with Embedded SQL statements.

```
letter: Character;        -- 1 character
word_arr: array(1..5) of word_5;
                          -- 2-dimensional array of char
```

Both could be declared instead with the less restrictive **string** type:

```
letter: String(1..1);
word_arr: array(1..5) of String(1..5);
                         -- Array of strings
```

Character strings containing embedded single quotes are legal in SQL, for example:

```
mary's
```

User variables may contain embedded single quotes and need no special handling unless the variable represents the entire search condition of a where clause:

```
where :variable
```

In this case you must escape the single quote by reconstructing the *:variable* string so that any embedded single quotes are modified to double single quotes, as in:

```
mary''s
```

Otherwise, a runtime error will occur.

For more information on escaping single quotes, see String Literals in this chapter. For more information on character strings that contain embedded nulls, see The Character and String Data Types in this chapter.

Variable and Number Declaration Syntax

The following sections discuss variable and number declaration syntax.

Simple Variable Declarations

An Embedded SQL/Ada variable declaration has the following syntax:

identifier {**,** *identifier*} **:**
[**constant**]
[**array (***dimensions***) of**]
type_name
[*type_constraint*]
[**:=** *initial_value];*

Syntax Notes:

- The *identifier* must be a legal Ada identifier beginning with an alphabetic character.
- If you specify the **constant** clause, the declaration must include an explicit initialization.
- If you specify the **constant** clause, the declared variables cannot be targets of Ingres assignments.
- The Embedded SQL preprocessor does not parse the *dimensions* of an **array** specification. Consequently, the preprocessor accepts unconstrained array bounds and multi-dimensional array bounds. However, an illegal *dimension* (such as a non-numeric expression) is also accepted but causes Ada compiler errors.

  For example, both of the following declarations are accepted, even though only the first is legal Ada:

  ```
  square:        array (1..10, 1..10) of Integer;
   bad_array:     array ("dimensions") of Float;
  ```

- A variable or type name must begin with an alphabetic character, which can be followed by alphanumeric characters or underscores.
- The *type_name* must be either an Embedded SQL/Ada type (refer to the list of acceptable types earlier in this chapter) or a type name already declared to Embedded SQL.
- The legal *type_constraints* are described in the next section.
- The preprocessor does not parse *initial_value*. Consequently, the preprocessor accepts any initial value, even if it can later cause an Ada compiler error. For example, both of the following initializations are accepted, even though only the first is legal Ada:

  ```
  rowcount: Integer := 1;
  msgbuf:   String(1..100) := 2; -- Incompatible value
  ```

You must not use a single quote in an initial value to specify an Ada attribute. Embedded SQL treats it as the beginning of a string literal and generates an error. For example, the following declaration generates an error:

```
id: Integer := Integer'First
```

The following is a sample variable declaration:

```
rows, records:    Integer range 0..500 := 0;
was_error:        Boolean;
min_sal:          constant Float := 15000.00;
msgbuf:           String(1..100) := (1..100 => ' ');
operators:        constant array(1..6) of String(1..2) :=
      ("= ", "!=", "<=", ">=");
```

Type Constraints

Type constraints can optionally follow the type name in an Ada object declaration. In general, they do not provide Embedded SQL with runtime type information, so they are not fully processed. The following two constraints describe the syntax and restrictions of Embedded SQL type constraints.

The Range Constraint

The syntax of the range constraint is:

> **range** *lower_bound .. upper_bound*

In a variable declaration, its syntax is:

> *identifier***:** *type_name* **range** *lower_bound .. upper_bound;*

Syntax Notes:

- Even if Ada does not allow a range constraint, Embedded SQL does accept it. For example, both of the following range constraints are accepted, although the second is illegal in Ada because the **string** type is not a discrete scalar type:

  ```
  digit: Integer range 0..9;
   chars: String range 'a'..'z';
  ```

- The two bounds, *lower_bound* and *upper_bound*, must be integer literals, floating-point literals, character literals, or identifiers. Other expressions are not accepted.
- The bounds are not checked for compatibility with the *type_name* or with each other. For example, the preprocessor accepts the following three range constraints, even though only the first is legal Ada:

  ```
  byte: Integer range -128..127;
  word: Integer range 1.0..30000.0;
                               --Incompatible with type name
  long: Integer range 1..'z';
                               --Incompatible with each other
  ```

The Discriminant and Index Constraints

The discriminant and index constraints have the following syntax:

(*discriminant_or_index_constraint***)**

In a variable declaration the syntax is:

identifier**:** *type_name* **(***discriminant_or_index_constraint***);**

Syntax Notes:

- Even if Ada does not allow a constraint, Embedded SQL does accept it. For example, Embedded accepts both of the following constraints, even though the second is illegal in Ada because the **integer** type does not have a discriminant:

```
who: String(1..20); -- Legal index constraint
nat: Integer(0);  -- Illegal context for discriminant
```

- The contents of the constraint contained in the parentheses are not processed. Consequently, Embedded SQL accepts any constraint, even if Ada does not allow it. For example, Embedded SQL accepts the following declaration but generates a later Ada compiler error because of the illegal index constraint:

```
password: String(secret word);
```

Note that the above type constraints are not discussed in detail after this section, and their rules and restrictions are considered part of the Embedded SQL/Ada declaration syntax.

Formal Parameter Declarations

An Embedded SQL/Ada formal parameter declaration has the following syntax:

identifier {, identifier} :
[**in** | **out** | **in out**
type_name
[*:= default_value*]
[*;*]

Like other Embedded SQL declarations, the formal parameter declaration must occur inside a declaration section. In a subprogram specification, its syntax is:

procedure *name*
exec sql begin declare section;
(*parameter_declaration {;*
parameter_declaration} **)**
exec sql end declare section;
is
...

or:

function *name*
exec sql begin declare section;
(*parameter_declaration* {;
parameter_declaration} **)**
exec sql end declare section;
return *type_name* ***is***
...

Syntax Notes:

- The Embedded SQL preprocessor processes only the formal parameter declarations in a subprogram specification.
- If you specify the **in** mode alone, the declared parameters are considered constants and cannot be targets of Ingres assignments.
- If you do not specify a mode, the default **in** mode is used and the declared parameters are considered constants.
- The *type_name* must be either an Embedded SQL/Ada type or a type name already declared to Embedded SQL.
- The preprocessor does not parse the *default_value*. Consequently, the preprocessor accepts any default value, even if it can later cause a Ada compiler error. For example, Embedded SQL accepts both of the following parameter defaults, even though only the first is legal in Ada:

```
procedure Load_Table
    exec sql begin declare section;
        (clear_it: in Boolean := TRUE;
         is_error: out Boolean := "FALSE")
    exec sql end declare section;
    is
    ...
```

You must not use a single quote in a default value to specify an Ada attribute. Embedded SQL treats it as the beginning of a string literal and generates an error.

- You must use the semicolon with all parameter declarations except the last.

- As with all other Embedded SQL/Ada declarations, formal parameter declarations are global from the point of declaration to the end of the file. For more information, see The Scope of Variables in this chapter.

Number Declarations

An Embedded SQL/Ada number declaration has the following syntax:

identifier {, *identifier*} *:*
constant *:= initial_value;*

Syntax Notes:

- You can only use a number declaration for integer numbers. You cannot declare a floating-point number declaration using this format. If you do, Embedded SQL treats it as an integer number declaration, later causing compiler errors. For example, the preprocessor treats the following two number declarations as integer number declarations, even though the second is a float number declaration:

  ```
  max_employees: constant := 50000;
   min_salary: constant := 13500.0; -- Treated as INTEGER
  ```

  In order to declare a constant float declaration, you must use the **constant** variable syntax. For example, you should declare the second declaration above as:

  ```
  min_salary: constant Float := 13500.0;
                                   -- Treated as FLOAT
  ```

- The declared numbers cannot be the targets of Ingres assignments.
- The preprocessor does not parse the *initial_value*. Consequently, the preprocessor accepts any initial value, even if it can later cause an Ada compiler error. For example, Embedded SQL accepts both of the following initializations, even though only the first is a legal Ada number declaration:

  ```
  no_rows: constant := 0;
  bad_num: constant := 123 + "456";
  ```

You must not use a single quote in an initial value to specify an Ada attribute. Embedded SQL treats it as the beginning of a string literal and generates an error.

Renaming Variables

The syntax for renaming variables is:

*identifier***:** *type_name* **renames** *declared_object;*

Syntax Notes:

- The *type_name* must be an Embedded SQL/Ada type or a type name already declared to Embedded SQL and the *declared_object* must be a known Embedded SQL variable or constant.
- The *declared_object* must be compatible with the *type_name* in base type, array dimensions, and size.
- If the declared object is a record component, any subscripts used to qualify the component are ignored. For example, the preprocessor accepts both of the following **rename** statements, even though one of them must be wrong, depending on whether "emprec" is an array:

```
eage1: Integer renames emprec(2).age;
eage2: Integer renames emprec.age;
```

Type Declaration Syntax

Embedded SQL/Ada supports a subset of Ada type declarations. In a declaration, the Embedded SQL preprocessor only notes semantic information relevant to the use of the variable in Embedded SQL statements at runtime. The preprocessor ignores other semantic information. Refer to the syntax notes in this section and its subsections for details.

Type Definition

An Embedded SQL/Ada full type declaration has the following syntax:

type *identifier* [*discriminant_part*] **is** *type_definition;*

Syntax Notes:

- The *discriminant_part* has the syntax:

  (*discriminant_specifications)*

  and is not processed by Embedded SQL. As with variable declarations, Embedded SQL always accepts a discriminant specification, even if Ada does not allow it. For example, Embedded SQL accepts the following declaration but later generates an Ada compiler error because the discriminant type is not a discrete type, and the discriminant part is not allowed in a non-record declaration:

```
type shapes(name: String := "BOX")
            is array(1..10) of String(1..3);
```

  From this point on, discriminant parts are not included in the syntax descriptions or notes.
- The legal *type_definitions* allowed in type declarations are described below.

Subtype Definition

An Embedded SQL/Ada **subtype** declaration has the following syntax:

subtype *identifier* **is** *type_name* [*type_constraint*]**;**

Syntax Note:

- The *type_constraint* has the same rules as the type constraint of a variable declaration. The range, discriminant and index constraints are all allowed and are not processed against the *type_name* being used. For more details about these constraints, see the section above on variable type constraints. The floating-point constraint and the **digits** clause, which are allowed in subtype declarations, are discussed later.

Integer Type Definitions

The syntax of an Embedded SQL/Ada integer type definition is:

range *lower_bound .. upper_bound*

In the context of an integer type declaration, the syntax is:

type *identifier* **is range** *lower_bound .. upper_bound;*

In the context of an integer subtype declaration, the syntax is:

subtype *identifier* **is** *integer_type_name*
range *lower_bound .. upper_bound;*

Syntax Notes:

- In an integer type declaration (*not* a subtype declaration), Embedded SQL processes the range constraint of an integer type definition to evaluate storage size information. Both *lower_bound* and *upper_bound* must be integer literals. Based on the specified range and the actual values of the bounds, Embedded SQL treats the type as a byte-size, a word-size or a longword-size integer. For example:

```
type Table_Num is range 1..200;
```

- In an integer subtype declaration, the range constraint is treated as a variable range constraint and is not processed. Consequently, the same rules that apply to range constraints for variable declarations apply to integer range constraints for integer subtype declarations. The base type and storage size information is determined from the *integer_type_name* used. For example:

```
subtype Ingres_I1 is Integer range -128..127;
subtype Ingres_I2 is Integer range -32768..32767;
subtype Table_Low is Table_Num range 1..10;
subtype Null_Ind is  Short_Integer range -1..0;
                                   -- Null Indicator
```

Floating-point Type Definitions

The syntax of an Embedded SQL/Ada floating-point type definition is:

> **digits** *digit_specification* [*range_constraint*]

In the context of a floating-point type declaration, the syntax is:

> **type** *identifier* **is digits** *digit_specification* [*range_constraint*]**;**

The syntax of a floating-point subtype declaration is:

> **subtype** *identifier* **is** *floating_type_name*
> [**digits** *digit_specification*]
> [*range_constraint*]**;**

Syntax Notes:

- The value of *digit_specification* must be an integer literal. Based on the value of the specification, Embedded SQL determines whether to treat a variable of that type as a 4-byte float or an 8-byte float. The following rules apply:

Digit Range	Type
1 < = *d* < = 6	4-byte floating-point type
7 <= *d* < = 16	8-byte floating-point type

  Note that if the digits specified are out of range, the type is unusable. Recall that Embedded SQL does not accept either the **long_long_float** or the **h_float** type. For detailed information on the internal storage format for 8-byte floating-point variables, see The Long Float Storage Format in this chapter.

- The *range_constraint* for floating-point types and subtypes is treated as a variable range constraint and is not processed. Although Embedded SQL allows any range constraint, you should not specify a range constraint that alters the size needed to store the declared type. Embedded SQL obtains its type information from the **digits** clause, and altering this type information by a range clause, which may require more precision, results in runtime errors.
- The **digits** clause in a subtype declaration does not have any effect on the Embedded SQL type information. This information is obtained from *floating_type_name*.

```
type Emp_Salary is digits 8 range 0.00..500000.00;
subtype Directors_Sal
               is Emp_Salary 100500.00..500000.00;
subtype Raise_Percent
               is Float range 1.05..1.20;
```

Enumerated Type Definitions

The syntax of an Embedded SQL/Ada enumerated type definition is:

> **(***enumerated_literal* {**,** *enumerated_literal*}**)**

In the context of a type declaration, the syntax is:

> **type** *identifier* **is (***enumerated_literal {, enumerated_literal}***);**

In the context of a subtype declaration, the syntax is:

> **subtype** *identifier* **is** *enumerated_type_name* [*range_constraint*]**;**

Syntax Notes:

- An enumerated type declaration can contain no more than 1000 enumerated literals. The preprocessor treats all literals and variables declared with this type as integers. Enumerated literals are treated as though they were declared with the **constant** clause, and therefore cannot be the targets of Ingres assignments. When using an enumerated literal with Embedded SQL statements, only the ordinal position of the value in relation to the original enumerated list is relevant. When assigning from an enumerated literal, Embedded SQL generates:

  > enumerated_type_name**'pos(***enumerated_variable_or_literal***)**

When assigning from or into an enumerated variable, Embedded SQL passes the object by address and assumes that the value being assigned from or into the variable will not raise a runtime constraint error.

- An enumerated literal can be an identifier or a character literal. Embedded SQL does not store or process enumerated literals that are character literals.

- Enumerated literal identifiers must be unique in their scope. Embedded SQL does not allow the overloading of variables or constants.

- The *range_constraint* for enumerated subtypes is treated as a variable range constraint and is not processed. The type information is determined from *enumerated_type_name*.

  ```
  type Table_Field_States is
      (UNDEFINED, NEWROW, UNCHANGED,CHANGED, DELETED);
   subtype Updated_States is
               Table_Field_States range CHANGED..DELETED;
   tbstate: Table_Field_States := UNDEFINED;
  ```

- ESQL accepts the predefined enumeration type name **Boolean,** which contains the two literals FALSE and TRUE. You can use a representation clause for enumerated types. When you do so, however, you should not reference any enumerated literals of that type in embedded statements, though you can reference the variables.

Enumerated literals are interpreted into their integer relative position (**pos**) and representation clauses invalidate the effect of the pos attribute that the preprocessor generates. The representation clause must be outside of the **declare** section.

- You can only use enumerated variables and literals to assign to or from Ingres. You cannot use these objects to specify simple numeric objects, such as table field row numbers or sleep **statement** seconds.

Array Type Definitions

The syntax of an Embedded SQL/Ada array type definition is:

array (*dimensions***) of** *type_name;*

In the context of a type declaration, the syntax is:

type *identifier* **is array (***dimensions***) of**
type_name [*type_constraint*]**;**

Syntax Notes:

- The *dimensions* of an **array** specification are not parsed by the Embedded SQL preprocessor. Consequently, the preprocessor accepts unconstrained array bounds and multi-dimensional array bounds. However, an illegal dimension (such as a non-numeric expression) is also accepted but later causes Ada compiler errors. For example, both of the following type declarations are accepted, even though only the first is legal in Ada:

```
type Square is array(1..10, 1..10) of Integer;
type What is array("dimensions") of Float;
```

  Because the preprocessor does not store the array dimensions, it only checks to determine that when you use the array variable, it is followed by a subscript in parentheses.

- The *type_constraint* for **array** types is treated as a variable type constraint and is not processed. The type information is determined from *type_name*.
- Any array built from the base type **character** (*not* **string**) must be exactly one-dimensional. Embedded SQL treats the whole array as though you declared it as type **string**. If you declare more dimensions for a variable of type **character**, Embedded SQL still treats it as a one-dimensional array.
- The type **string** is the only array type.

Record Type Definitions

The syntax of an Embedded SQL/Ada record type definition is:

> **record**
> *record_component* {*record_component*}
> **end record;**

where *record_component* is:

> *component_declaration* **;** | *variant_part***;** | **null;**

where *component_declaration* is:

> *identifier* {**,** *identifier*} **:**
> *type_name* [*type_constraint*] [**:=** *initial_value*]

In the context of a type declaration, the syntax of a record type definition is:

> **type** *identifier* **is**
> **record**
> *record_component* { *record_component*}
> **end record;**

Note that the *SQL Reference Guide* refers to records as structures and record components as structure members.

Syntax Notes:

- In a *component_declaration*, all clauses have the same rules and restrictions as they do in a regular type declaration. For example, as in regular declarations, the preprocessor does not check initial values for correctness.
- The *variant_part* accepts the Ada syntax for variant records: if specified, it must be the last component of the record. The variant discriminant name, choice names, and choice ranges are all accepted. There is no syntactic or semantic checking on those variant objects. Embedded SQL uses only the final component names of the variant part and not any of the variant object names.
- You can specify the **null** record.

The following example illustrates the use of record type definitions:

```
type Address_Rec is
    record
        street:    String(1..30);
        town:      String(1..10);
        zip:       Positive;
    end record;

type Employee_Rec is
    record
```

```
    name:      String(1..20);
    age:       Short_Short_Integer;
    salary:    Float := 0.0;
    address:   Address_Rec;
end record;
```

Incomplete Type Declarations and Access Types

The incomplete type declaration should be used with an access type. The syntax for an incomplete type declaration is:

type *identifier [discriminant_part]***;**

Syntax Notes:

- As with other type declarations, the *discriminant_part* is ignored.
- You must fully define an incomplete type before using any object declared with it.

The syntax for an access type declaration is:

type *identifier* **is access** ***t****ype_name* [*type_constraint*]**;**

Syntax Notes:

- The *type_name* must be an Embedded SQL/Ada type or a type name already declared to Embedded SQL, whether it is a full type declaration or an incomplete type declaration.
- The *type_constraint* has the same rules as other type declarations.

```
type Employee_Rec; -- Incomplete declaration
type Employee is access Employee_Rec;
                  -- Access to above

type Employee_Rec is -- Real definition
        record
            name:     String(1..20);
            age:      Short_Short_Integer;
            salary:   Float := 0.0;
            link:     Employee;
        end record;
```

Derived Types

The syntax for a derived type is:

type *identifier* **is new** *type_name* [*type_constraint*]**;**

Syntax Notes:

- The *type_name* must be an embedded SQL/Ada type or a type name already declared to Embedded SQL, whether it is a full type declaration or an incomplete type declaration.

- Embedded SQL assigns the type being declared the same properties as the *type_name* specified. The preprocessor makes sure that any variables of a derived type are cast into the original base type when used with the runtime routines.
- The *type_constraint* has the same rules as other type declarations.

```
type Dbase_Integer is new Integer;
```

Private Types

The syntax for a private type is:

type *identifier* **is** [**limited**] **private;**

Syntax Note:

This type declaration is treated as an incomplete type declaration. You must fully define a private type before using any object declared with it.

Representation Clauses

With one exception, you must not use representation clauses for any types or objects you have declared to Embedded SQL and intend to use with the Embedded SQL runtime system. Any such clauses causes runtime errors. These clauses include the Ada statement:

for *type_or_attribute* **use** *expression;*

and the Ada pragma:

pragma pack(*type_name***);**

The exception is that you can use a representation clause to specify internal values for enumerated literals. When you do so, however, you should not reference any enumerated literals of the modified enumerated type in embedded statements. The representation clause invalidates the effect of the **pos** attribute that the preprocessor generates. If the application context is one that requires the assignment from the enumerated type, then you should deposit the literal into a variable of the same enumerated type and assign that variable to Ingres. In all cases, do not include the representation clause in a **declare** section. For example:

```
exec sql begin declare section;
    type opcode is (opadd, opsub, opmul);
exec sql end declare section;
...

for opcode use (opadd => 1, opsub => 2, opmul => 4);
...

opcode_var := opsub;
```

```
exec sql insert into codes values (:opcode_var);
```

The DCLGEN Utility

DCLGEN (Declaration Generator) is a record-generating utility that maps the columns of a database table into a record that can be included in a variable declaration. Use the following command to invoke DCLGEN from the operating system level:

dclgen *language dbname tablename filename recordname*

where

- *language* is the Embedded SQL host language, in this case, "ada."
- *dbname* is the name of the database containing the table.
- *tablename* is the name of the database table.
- *filename* is the output file into which the record declaration is placed.
- *recordname* is the name of the Ada record variable that the command creates. The command generates a record type definition named *recordname*, followed by "_rec". It also generates a variable declaration for *recordname*.

This command creates the declaration file *filename*. The file contains a record type definition corresponding to the database table and a variable declaration of that record type. The file also includes a **declare table** statement that serves as a comment and identifies the database table and columns from which the record was generated.

After you have generated the file, you can use an Embedded SQL **include** statement to incorporate it into the variable declaration section. The following example demonstrates how to use DCLGEN in an Ada program.

Assume the Employee table was created in the Personnel database as:

```
exec sql create table employee
            (eno    smallint not null,
            ename   char(20) not null,
            age     integer1,
            job     smallint,
            sal     decimal(14,2) not null,
            dept    smallint);
```

and the DCLGEN system-level command is:

```
dclgen ada personnel employee employee.dcl emprec
```

The employee.dcl file created by this command contains a comment and three statements. The first statement is the **declare table** description of “employee,” which serves as a comment. The second statement is a declaration of the Ada record type definition “emprec_rec.” The last statement is a declaration, using the “emprec_rec” type, for the record variable “emprec.” The exact contents of the employee.dcl file are:

```
-- Description of table employee from database personnel
exec sql declare employee table
    (eno        smallint not null,
    ename       char(20) not null,
    age         integer1,
    job         smallint,
    sal         decimal(14,2) not null,
    dept        smallint);
type emprec_rec is
    record
    eno:        short_integer;
    ename:      string(1..20);
    age:        short_short_integer;
    job:        short_integer;
    sal:        long_float;
    dept:       short_integer;
    end record;
emprec: emprec_rec;
```

You should include this file, by means of the Embedded SQL **include** statement, in an Embedded SQL declaration section:

```
exec sql begin declare section;
                exec sql include 'employee.dcl';
exec sql end declare section;
```

You can then use the emprec record in a **select**, **fetch**, or **insert** statement.

The field names of the structure that DCLGEN generates are identical to the column names in the specified table. Therefore, if the column names in the table contain any characters that are illegal for host language variable names you must modify the name of the field before attempting to use the variable in an application.

DCLGEN and Large Objects

When a table contains a large object column, DCLGEN will issue a warning message and map the column to a zero length character string variable. You must modify the length of the generated variable before attempting to use the variable in an application.

For example, assume that the “job_description” table was created in the personnel database as:

```
create table job_description (job smallint,
     description long varchar);
```

and the DCLGEN system level command is:

```
dclgen ada personnel job_descriptionjobs.dcl jobs_rec
```

The contents of the jobs.dcl file would be:

```
-- Description of table job_description from
-- database personnel
exec sql declare job_description table
                (job         smallint,
                long_column long varchar);

type jobs_rec_rec is
record
         job:         short_integer;
         description: string(1..0);

end record
jobs_rec: jobs_rec_rec;
```

Indicator Variables

An *indicator variable* is a 2-byte integer variable. You can use an indicator variable in three possible ways in an application:

- In a statement that retrieves data from Ingres. You can use an indicator variable to determine if its associated host indicator variable was assigned a **null**.
- In a statement that sets data to Ingres. You can use an indicator variable to assign a null to the database column, form field, or table field column.
- In a statement that retrieves character data from Ingres, you can use the indicator variable as a check that the associated host variable was large enough to hold the full length of the returned character string. However, the preferred method is to use SQLSTATE.

In order to declare an indicator variable, you should use the **short_integer** data type. The following example declares two indicator variables:

```
ind:     Short_Integer; -- Indicator variable
ind_arr: array(1..10) of Short_Integer; --Indicator array
```

When using an indicator variable with an Ada record, you must declare the indicator variable as an array of 2-byte integers. In the above example, you can use the variable “ind_arr” as an indicator array with a record assignment. Note that a variable declared with any derivative of the **short_integer** data type will be accepted as an indicator variable

Assembling and Declaring External Compiled Forms

You can pre-compile your forms in the Visual Forms Editor (VIFRED). This saves the time otherwise required at runtime to extract the form’s definition from the database forms catalogs. When you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED prompts you for the name of the file with the MACRO description. After the file is created, use the following VMS command to assemble it into a linkable object module:

macro *filename*

This command produces an object file containing a global symbol with the same name as your form. Before the Embedded SQL/FORMS statement **addform** can refer to this global object, it must be declared in an Embedded SQL declaration section. The Ada compiler requires that the declaration be in a package and that the objects be imported with the **import_object** pragma.

The syntax for a compiled form package is:

package *compiled_forms_package* **is**
  exec sql begin declare section;
    *formname***: Integer;**
  exec sql end declare section;
  pragma import_object(*formname* **);**
end *compiled_forms_package;*

You must then issue the Ada **with** and **use** statements on the compiled form package before every compilation unit that refers to the form:

with *compiled_forms_package;* **use** *compiled_forms_package;*

Syntax Notes:

- The *formname* is the actual name of the form. VIFRED gives this name to the address of the external object. The *formname* is also used as the title of the form in other Embedded SQL/FORMS statements.
- The **import_object** pragma associates the object with the external form definition. To use this pragma, the package must be issued in the outermost scope of the file.

The example below shows a typical form declaration and illustrates the difference between using the form's object definition and the form's name.

```
package Compiled_Forms is
        exec sql begin declare section;

                empform: Integer;
        exec sql end declare section;
        pragma import_object( empform );
end Compiled_Forms;
        ...

with Compiled_Forms; use Compiled_Forms;
         ...

exec frs addform :empform; -- The imported object
exec frs display empform; -- The name of the form
         ...
```

Concluding Example

The following example demonstrates some simple Embedded SQL/Ada declarations.

```
package Compiled_Forms is

        exec sql begin declare section;

                        empform, deptform: Integer; -- Compiled forms
        exec sql end declare section;

        pragma import_object( empform );
        pragma import_object( deptform );
    end Compiled_Forms;

with Compiled_Forms; use Compiled_Forms;

exec sql include sqlca; -- Include error handling

package Concluding_Example is
        exec sql begin declare section;

                        max_persons: constant := 1000;
                        dbname:             String(1..9):="personnel";
                        formname, tablename, columnname: String(1..12);
                        salary:             Float;

          type datatypes_rec is -- Structure of all types
                      d_byte: Short_Short_Integer;
                      d_word: Short_integer;
                      d_long: Integer;
                      d_single: Float;
                      d_double: Long_float;
                      d_string: String(1..20);
                end record;
                d_rec: datatypes_rec;

          -- Record with a discriminant
               record persontype_rec (married: in Boolean) is
                      age:   Short_Short_Integer;
                      flags: Integer;
                      case married:
                            when TRUE =
                                 spouse_name: String(1..30);
                            when FALSE =
                                  dog_name: String(1..12);
                      end case;
                end record;
                person: persontype_rec(TRUE);
                person_store: array(1..max_persons) of
                                        persontype_rec(false);

             exec sql include 'employee.dcl'; -- From dclgen
             ind_var: Short_Integer := -1;    -- Indicator
                                           -- variable

    exec sql end declare section;
end concluding_examples;
```

The Scope of Variables

The preprocessor can reference and accept all variables declared in an Embedded SQL declaration section from the point of declaration to the end of the file, regardless of the Ada scope of the declaration. This holds true for declarations in a package body or specification (even if they are **private**), formal parameters, and local variables of functions and procedures. Once an object has been declared to Embedded SQL, it must be the same size and type. It must not be redeclared to Embedded SQL for use in a different Ada scope; the preprocessor uses the type information supplied by the original declaration. The object must, however, be redeclared to Ada in the second scope to avoid errors from the Ada compiler.

This restriction means that two package specifications cannot declare two different objects with the same name. The following example generates an error because of the redeclaration of the object "ptr":

```
package Stack is
        exec sql begin declare section;
                stack_max:    constant := 50;
                ptr:          Integer range 1..stack_max;
                stack_arr:    array(1..stack_max) of integer;
        exec sql end declare section;
end Stack;

package Employees is
        exec sql begin declare section;
                ename_arr: array(1..1000) of string(1..20);
                ptr: string(1..20);
        exec sql end declare section;
end Employees;
```

In the following program fragment, the variable "dbname" is passed as a parameter to the second procedure. In the first declaration section, the variable is a local variable. In the second procedure, the variable is a formal parameter passed as a string to be used with the **connect** statement. The declaration of "dbname" as a formal parameter to the second procedure should not occur in an Embedded SQL declaration section. In both procedures, the preprocessor uses the type information from the variable's declaration in the first procedure.

```
package Decl_Test is
        procedure Open_Db(dbname: in String);
        procedure Access_Db;
end Decl_Test;

exec sql include sqlca;
package body Decl_Test is
    procedure Access_Db is
        exec sql begin declare section;
            dbname: String(1..15);
        exec sql end declare section;
    begin
        -- Prompt for database name
        put( "Database:" );
        get( dbname );
        Open_Db( dbname );
            ...
```

```
    end Access_Db;

    procedure Open_Db (dbname: in String) is
    begin
        exec sql whenever error stop;
        exec sql connect :dbname;
            ...

    end Open_Db;
end Decl_Test;
```

Note that you can declare record components with the same name but different record types. The following example declares two records, each of which has the components "firstname" and "lastname":

```
exec sql begin declare section;
    type child is
        record
            firstname: String(1..15);
            lastname:  String(1..20);
            age:       Integer;
        end record;

    type some_childs is array(1..10) of child;

    type mother is
        record
            firstname: String(1..15);
            lastname:  String(1..20);
            num_child: Integer range 1..10;
            children:  Some_Childs;
        end record;
exec sql end declare section;
```

Special care should be taken when using variables with a **declare cursor** statement. The variables used in such a statement must also be valid in the scope of the **open** statement for that same cursor. The preprocessor actually generates the code for the **declare** at the point that the **open** is issued and, at that time, evaluates any associated variables. For example, in the following program fragment, even though the variable "number" is valid to the preprocessor at the point of both the **declare cursor** and **open** statements, it is not a valid variable name for the Ada compiler at the point that the **open** is issued.

```
package Bad_Cursors is -- This example contains an error
      procedure Init_Csr1 is
            exec sql begin declare section;
                number: Integer;
            exec sql end declare section;
        begin
            exec sql declare cursor1 cursor for
                select ename, age
                from employee
                where eno = :number;

              -- Initialize "number" to a particular value
            ...

        end Init_Csr1;

      procedure Process_Csr1 is
```

```
              exec sql begin declare section;
                  ename: String(1..15);
                  age:   Integer;
              exec sql end declare section;
        begin

            -- illegal evaluation of "number"
            exec sql open cursor1;

            exec sql fetch cursor1 into :ename, :age;
            ...

          end Process_Csr1;
end Bad_Cursors;
```

If you must use a group of types and variables in multiple subprograms and package bodies, you can put their declarations in a package and explicitly issue **with** and **use** clauses before each subprogram or package that uses them. The following example declares two variables inside a package specification. The variables are used by two procedures, each of which must be preceded by the **with** and **use** clauses:

```
package Vars is
    exec sql begin declare section;
            var1: Integer;
            var2: String(1..3);
    exec sql end declare section;
end Vars;

with Vars; use Vars; -- Explicit Ada visibility clauses

procedure Read_Vars is
begin
    -- Embedded sql statements that retrieve "var1" and
    -- "var2"
end Read_Vars;

with Vars; use Vars; -- Explicit ada visibility clauses

procedure Write_Vars is
begin
    -- Embedded sql statements that insert "var1"
    -- and "var2"
end Write_Vars;
```

Variable Usage

Ada variables declared to Embedded SQL can substitute for many non key-word elements of Embedded SQL statements. Of course, the variable and its data type must make sense in the context of the element. When you use an Ada variable (or named constant) in an Embedded SQL statement, you must precede it with a colon. You must further verify that the statement using the variable is in the scope of the variable's declaration. As an example, the following **select** statement uses the variables "namevar" and "numvar" to receive data, and the variable "idnovar" as an expression in the **where** clause:

```
exec sql select name, num
        into :namevar, :numvar
        from employee
```

```
where idno = :idnovar;
```

When referencing a variable, you cannot use an Ada attribute, because the attribute is introduced by a single quote. Embedded SQL treats this single quote as the beginning of a string literal and generates a syntax error.

Various rules and restrictions apply to the use of Ada variables in Embedded SQL statements. The sections below describe the usage syntax of different categories of variables and provide examples of such use.

Simple Variables

A simple scalar-valued variable (integer, floating-point or character string) is referred to by the syntax:

:simplename

Syntax Notes:

- If you use the variable to send data to Ingres, it can be any scalar-valued variable, constant, or enumerated literal.
- If you use the variable to receive data from Ingres, it cannot be a variable declared with the **constant** clause, a formal parameter that does not specify the **out** mode, a number declaration, or an enumerated literal.
- A string variable (a 1-dimension array of characters) is referenced as a simple variable.

The following program fragment demonstrates a typical message-handling routine that uses two scalar-valued variables, “buffer” and “seconds”:

```
procedure Msg
    exec sql begin declare section;
        (buffer: String; seconds: Integer)
    exec sql end declare section;
is
begin
    exec frs message :buffer;
    exec frs sleep :seconds;
end Msg;
```

A special case of a scalar type is the enumerated type. Embedded SQL treats all enumerated literals and any variables declared with an enumerated type as integers. When an enumerated literal is used in an Embedded SQL statement, only the ordinal position of the value in relation to the original enumerated list is relevant. When assigning from an enumerated variable or literal, Embedded SQL generates the following:

enumerated_type_name**'pos(***enumerated_variable_or_literal)*

When assigning from or into an enumerated variable, the preprocessor passes the object by address and assumes that the value being assigned from or into the variable does not raise a runtime constraint error. In order to relax the restriction imposed by the preprocessor on enumerated literal assignments (of enumerated types that have included representation clauses to modify their values), you should assign the literal to a variable of the same enumerated type before using it in an embedded statement. For example, the following enumerated type declares the states of a table field row, and the variable of that type always receives one of those values:

```
exec sql begin declare section;
      type Table_Field_States is
            (undefined, newrow, unchanged, changed, deleted);
      tbstate: Table_Field_States := undefined;
      ename: String(1..20);
exec sql end declare section;
        ...

exec frs getrow empform employee (:ename = name,
        :tbstate = _state);

case tbstate is
            when undefined =>
            ...
end case;
```

Another example retrieves the value TRUE (an enumerated literal of type **boolean**) into a variable when a database qualification is successful:

```
exec sql begin declare section;
        found: Boolean;
        name:  String(1..30);
exec sql end declare section;
        ...

exec sql select :true
        into :found
        from personnel
        where ename = :name;
if (not found) then
        ...
end if;
```

Note that a colon precedes the Ada enumerated literal "TRUE." The colon is required before all named Ada objects—constants and enumerated literals, as well as variables—used in Embedded SQL statements.

Array Variables

An array variable is referred to by the syntax:

:arrayname(subscript{,subscript})

Syntax Notes:

- You must subscript the variable because only scalar-valued elements (integers, floating-point, and character strings) are legal Embedded SQL values.
- When you declare the array, the Embedded SQL preprocessor does not parse the array bounds specification. Consequently, the preprocessor accepts illegal bounds values. Also, when you reference an array, the subscript is not parsed, allowing you to use illegal subscripts. The preprocessor only confirms that you used an array subscript for an array variable. You must make sure that the subscript is legal and that you used the correct number of indices.
- A character string variable is *not* an array and cannot be subscripted in order to reference a single character or a *slice* of the string. For example, if the following variable were declared:

  ```
  abc: String(1..3) := "abc";
  ```

  you could not reference

  ```
  :abc(1)
  ```

  to access the character "a". To perform such a task, you should declare the variable as an array of three one-character long strings:

  ```
  abc: array(1..3) of String(1..1) := ("a","b","c");
  ```

  Note that you can only declare variables of the Ada **character** type as a one-dimensional array. When you use a variable of that type, you must not subscript it.
- Arrays of null indicator variables used with record assignments should *not* include subscripts when referenced.

In the following example, the loop variable "i" is used as a subscript and need not be declared to Embedded SQL, as it is not parsed.

```
exec sql begin declare section;
        formnames: array(1..3) of String(1..8);
exec sql end declare section;
    ...

for i in 1..3 loop
        exec frs forminit :formnames(i);
end loop;
```

Record Variables

You can use a record variable in two different ways. First, you can use the record as a simple variable, implying that all its components are used. This would be appropriate in the Embedded SQL **select**, **fetch** and **insert** statements. Second, you can use a component of a record to refer to a single element. Of course, this component must be a scalar value (integer, floating-point or character string).

Using a Record as a Collection of Variables

The syntax for referring to a complete record is the same as referring to a simple variable:

:recordname

Syntax Notes:

- The *recordname* can refer to a main or nested record. It can be an element of an array of records. Any variable reference that denotes a record is acceptable. For example:

```
:emprec           -- A simple record
:record_array(i) -- An element of an array of records
:record.minor2.minor3 -- A nested record at level 3
```

- In order to be used as a collection of variables, the final record in the reference must have no nested records or arrays. The preprocessor enumerates all the components of the record and they must have scalar values. The preprocessor generates code as though the program had listed each record component in the order in which it was declared.
- You must not use a record with a variant part as a complete record. The preprocessor generates explicit references to each of its components, including the components of the variant. Because the preprocessor generates references to all variant components but not to discriminants, which it ignores (see the section above on the discriminant constraint), the use of a record with a variant part results in either a "wrong number of values" preprocessor error or a runtime error.

The following example uses the employee.dcl file generated by DCLGEN, to retrieve values into a record.

```
exec sql begin declare section;
    exec sql include 'employee.dcl';
            -- see above for description
exec sql end declare section;
    ...

exec sql select *
        into :emprec
        from employee
        where eno = 123;
```

The example above generates code as though the following statement had been issued instead:

```
exec sql select *
        into :emprec.eno, :emprec.ename, :emprec.age,
        :emprec.job, :emprec.sal, :emprec.dept
        from employee
        where eno = 123;
```

The example below fetches the values associated with all the columns of a cursor into a record.

```
exec sql begin declare section;
    exec sql include 'employee.dcl';
            -- see above for description
exec sql begin declare section;

exec sql declare empcsr cursor for
        select *
        from employee
        order by ename;
        ...
exec sql fetch empcsr into :emprec;
```

The following example inserts values by looping through a locally declared array of records whose elements have been initialized:

```
exec sql begin declare section;
        exec sql declare person table
              (pname    char(30),
               page     integer1,
               paddr    varchar(50));

        type Person_Rec is record
               name:  String(1..30);
               age:   Short_Short_Integer;
               addr:  String(1..50);
        end record;
        person: array(1..10) of Person_Rec;
exec sql end declare section;
                ...

for i in 1..10 loop
    exec sql insert into person
                values (:person(i));
end loop;
```

The **insert** statement in the example above generates code as though the following statement had been issued instead:

```
exec sql insert into person
        values (:person(i).name,:person(i).age,:person(i).addr);
```

Using Record Components

The syntax Embedded SQL uses to refer to a record component is the same as in Ada:

:record.component{.component}

Syntax Notes:

- The last record *component* denoted by the above reference must be a scalar value (integer, floating-point or character string). There can be any combination of arrays and records, but the last object referenced must be a scalar value. Thus, the following references are all legal:

```
-- Assume correct declarations for "employee,"
 -- "person" and other records.
 employee.sal       -- Component of a record
person(3).name
                   -- Component of an element of an array
rec1.mem1.mem2.age -- Deeply nested component
```

- You must fully qualify all record components when referenced. You can shorten the qualification by using the Ada **renames** clause in another declaration to rename some components or nested records.

The following example uses the array of emprec records to load values into the emptable tablefield in empform form.

```
exec sql begin declare section;
        type Employee_Rec is
                record
                        ename:    String(1..20);
                        eage:     Short_Integer;
                        eidno:    Integer;
                        ehired:   String(1..25);
                        edept:    String(1..10);
                        esalary:  Float;
                end record;
          emprec: array(1..100) of Employee_Rec;
    exec sql begin declare section;
    ...

for i in 1..100 loop
        exec frs loadtable empform emptable
            (name = :emprec(i).ename, age = :emprec(i).eage,
            idno = :emprec(i).eidno, hired =
                   :emprec(i).ehired,
            dept = :emprec(i).edept,
            salary =:emprec(i).esalary);
end loop;
```

If you want to shorten the reference to the record, you can use the **renames** clause to rename a particular member of the emprec array, as in the following example:

```
for i in 1..100 loop
    declare
          exec sql begin declare section;
                er: Employee_Rec renames emprec(i);
          exec sql end declare section;
    begin
          exec frs loadtable empform emptable
             (name = :er.ename, age = :er.eage,
              idno = :er.eidno, hired = :er.ehired,
          dept = :er.edept, salary = :er.esalary);
    end;
end loop;
```

Access Variables

An access variable must qualify another object using the dot operator, and using the same syntax as a record component:

:access.reference

Syntax Notes:

- By the time you reference an access variable, you must fully define the type to which it is *pointing*. This is true even for access types that were declared to point at incomplete types.
- The final object denoted by the above reference must be a scalar value (integer, floating-point or character string). There can be any combination of arrays, records or access variables, but the last object referenced must be a scalar value.
- If an access variable is pointing at a scalar-valued type, then the qualification must include the Ada **.all** clause to refer to the scalar value. If you use the **.all** clause, it must be the last component in the qualification. For example:

```
exec sql begin declare section;
    type Access_Integer is access Integer;
    ai: Access_Integer;
exec sql end declare section;
    ...

ai := new Integer'(2);
exec frs sleep :ai.all;
```

In the following example, an access type to an employee record is used to load a linked list of values into the Employee database table.

```
exec sql begin declare section;
        type Employee_Rec;
        type Emp_Link is access Employee_Rec;
        type Employee_Rec is
                record
                        ename: String(1..20);
                        eage:  Short_integer;
                        eidno: Integer;
                        enext: Emp_Link;
                end record;
        elist: Emp_Link;
exec sql end declare section;
    ...

while (elist <= null) loop
    exec sql insert into employee (name, age, idno)
        values (:elist.ename, :elist.eage, :elist.eidno);
    elist := elist.enext;
end loop;
```

Using Indicator Variables

The syntax for referring to an *indicator* variable is the same as for a simple variable, except that an indicator is always associated with a host variable:

:host_variable:indicator_variable

or

:host_variable **indicator** *:indicator variable*

Syntax Notes:

- The indicator variable can be a simple variable, an array element, or a record component that yields a 2-byte integer (**short_integer**). For example:

```
ind: Short_Integer;  -- Indicator variable
ind_arr: array(1..10) of Short_Integer;
                     -- Indicator array
:var_1:ind_var
:var_2:ind_arr(2)
```

- If the host variable associated with the indicator variable is a record, then the indicator variable should be an array of 2-byte integers. In this case the array should *not* be dereferenced with a subscript.
- When you use an indicator array, the first element of the array corresponds to the first component of the record, the second element with the second component, and so on. Indicator array elements begin at subscript 1 regardless of the range with which the array was declared.

The following example uses the employee.dcl file generated by DCLGEN and the empind array to retrieve values and **nulls** into a structure.

```
exec sql include sqlca;
exec sql begin declare section;

    exec sql include 'employee.dcl';
                   -- See above for description
    empind: array(1..10) of short_integer;

exec sql end declare section;

exec sql select *
 into :emprec:empind
 from employee;
```

The above example generates code as though the following statement had been issued:

```
exec sql select *
    into :emprec.eno:empind(1), :emprec.ename:empind(2),
            :emprec.age:empind(3), :emprec.job:empind(4),
            :emprec.sal:empind(5), :emprec.dept:empind(6),
    from employee;
```

Data Type Conversion

An Ada variable declaration must be compatible with the Ingres value it represents. Numeric Ingres values can be set by and retrieved into numeric variables, and Ingres character values can be set by and retrieved into character string variables.

Data type conversion occurs automatically for different numeric types, such as from floating-point database column values into integer Ada variables, and for character strings, such as from varying-length Ingres character fields into fixed-length Ada character string buffers.

Ingres does *not* automatically convert between numeric and character types. You must use the Ingres type conversion operators, the Ingres **ascii** function, or an Ada conversion procedure for this purpose.

The following table shows the default type compatibility for each Ingres data type. Note that some Ada types do not match exactly and, consequently, may go through some runtime conversion.

Ingres Data Types and Corresponding Ada Data Types

Ingres Type	Ada Type
char(*N*)	string(1..*N*)
char(*N*)	array(1..*N*) of character
varchar(*N*)	string(1..*N*)
varchar(*N*)	array(1..*N*) of character
integer1	short_short_integer
smallint	short_integer
integer	integer
float4	float
float4	f_float
float	long_float
float	d_float
date	string(1..25)
money	long_float
table_key	string (1..8)
object_key	string (1..16)
decimal	float
long varchar	string( )

Runtime Numeric Type Conversion

The Ingres runtime system provides automatic data type conversion between numeric-type values in the database and the forms system and numeric Ada variables. The standard type conversion rules (according to standard VAX rules) are followed. For example, if you assign a **float** variable to an integer-valued field, the digits after the decimal point of the variable's value are truncated. Runtime errors are generated for overflow on conversion.

The Ingres **money** type is represented as **long_float**, an 8-byte floating-point value.

Runtime Character and Varchar Type Conversion

Automatic conversion occurs between Ingres character string values and Ada character string variables. The string-valued Ingres objects that can interact with character string variables are:

- Ingres names, such as form and column names
- Database columns of type **character**
- Database columns of type **varchar**
- Form fields of type **character**.
- Database columns of type **long varchar**

Several considerations apply when dealing with character string conversions, both to and from Ingres.

The conversion of Ada character string variables used to represent Ingres names is simple: trailing blanks are truncated from the variables because the blanks make no sense in that context. For example, the string literals "empform " and "empform" refer to the same form.

The conversion of other Ingres objects is a bit more complicated. First, the storage of character data in Ingres differs according to whether the medium of storage is a database column of type **character**, a database column of type **varchar**, or a **character** form field. Ingres pads columns of type **character** with blanks to their declared length. Conversely, it does not add blanks to the data in columns of type **varchar** or **long varchar**, or in form fields.

Second, Embedded SQL assumes that the convention is to blank-pad fixed-length character strings. Character string variables not blank-padded may be storing ASCII nulls or data left over from a previous assignment. For example, the character string "abc" can be stored in an Ada **string(1..5)** variable as the string "abc " followed by two blanks.

When character data is retrieved from a Ingres database column or form field into an Ada character string variable and the variable is longer than the value being retrieved, the variable is padded with blanks. If the variable is shorter than the value being retrieved, the value is truncated. You should always ensure that the variable is at least as long as the column or field in order to avoid truncation of data.

When inserting character data into a Ingres database column or form field from an Ada variable, note the following conventions:

- When you insert data from an Ada variable into a database column of type **character** and the column is longer than the variable, the column is padded with blanks. If the column is shorter than the variable, the data is truncated to the length of the column.
- When you insert data from an Ada variable into a database column of type **varchar** or **long varchar** and the column is longer than the variable, no padding of the column takes place. Furthermore, by default, all trailing blanks in the data are truncated before the data is inserted into the **varchar** column. For example, when a string "abc" stored in an Ada **string(1..5)** variable as "abc " followed by two blanks is inserted into the **varchar** column, the two trailing blanks are removed and only the string "abc" is stored in the database column. To retain such trailing blanks, you can use the Ingres **notrim** function. It has the following syntax:

  notrim(:*stringvar)*

  where *stringvar* is a character string variable. An example demonstrating this feature follows later. If the **varchar** column is shorter than the variable, the data is truncated to the length of the column.
- When you insert data from an Ada variable into a **character** form field and the field is longer than the variable, no padding of the field takes place. In addition, all trailing blanks in the data are truncated before inserting the data into the field. If the field is shorter than the data (even after all trailing blanks have been truncated), the data is truncated to the length of the field.

  When comparing character data in an Ingres database column with character data in an Ada variable, note the following convention:
- When comparing data in **character** or **varchar** database columns with data in a character variable, all trailing blanks are ignored. Initial and embedded blanks are significant.

Note: As described above, the conversion of character string data between Ingres objects and Ada variables often involves the trimming or padding of trailing blanks, with resultant change to the data. If trailing blanks have significance in your application, give careful consideration to the effect of any data conversion. For a complete description of the significance of blanks in string comparisons, see the *SQL Reference Guide*.

The Ingres **date** data type is represented as a 25-byte character string.

The program fragment in the next example demonstrates the **notrim** function and the truncation rules explained above.

```
exec sql include sqlca;
        ...
exec sql begin declare section;
         exec sql declare varychar table
```

```
            (row     integer,
             data   varchar(10));
                -- Note the varchar data type
        row:  Integer;
        data: String(1..7) := (1..7 => ' ');
exec sql end declare section;
    ...

data(1..3):="abc    ";-- Holds "abc" followed by 4 blanks

-- The following insert adds the string "abc"
-- (blanks truncated)

exec sql insert into varychar (row, data)
        values (1, :data);

-- This statement adds the string "abc    ", with 4
-- trailing blanks left intact by using the
-- notrim function.

exec sql insert into varychar (row, data)
        values (2, notrim(:data));

-- This select will retrieve row #2, because the notrim
-- function left trailing blanks in the "data" variable
-- in the last insert statement.

exec sql select row
        into :row
        from varychar
        where length(data) = 7;

put("Row found = ");
put(row);
```

The SQL Communications Area

This section describes the SQL Communications Area (SQLCA) as implemented in Ada.

The Include SQLCA Statement

You must issue the **include sqlca** statement in front of each compilation unit (subprogram specification, subprogram body, package specification, or package body) containing Embedded SQL statements. You cannot issue the **include sqlca** statement inside a compilation unit because the statement causes the preprocessor to generate **with** and **use** clauses, which are not legal in that context.

```
exec sql include sqlca;
package Employees is
    procedure Emp_Util_1 is
            -- Declarations for emp_util_1
    begin
            -- Embedded statements for emp_util_1
    end Emp_Util_1;
```

```
    procedure Emp_Util_2
            -- Declarations for emp_util_2
    begin
            -- Embedded statements for emp_util_2
    end Emp_Util_2;
end Employees;
```

The **include sqlca** statement instructs the preprocessor to generate code that includes references to the SQLCA (SQL Communications Area) record for error handling on database statements. It generates Ada **with** and **use** statements referencing a package that defines the SQLCA record variable. The package specification must first be entered in your Ada program library by the procedure described in Entering Embedded SQL Package Specifications in this chapter.

Whether or not you intend to use the SQLCA for error handling, you must issue an **include sqlca** statement. If you do not issue it, the Ada compiler generates errors about undeclared function names.

Contents of the SQLCA

One of the results of issuing the **include sqlca** statement is the declaration of the SQLCA structure, which you can use for error handling in the context of database statements. The record declaration for the SQLCA is:

```
type IISQL_ERRM is       -- Varying length string.
        record

                sqlerrml: Short_Integer;
                sqlerrmc: String(1..70);
        end record;

type IISQL_ERRD is array(1..6) of Integer;

type IISQL_WARN is       -- Warning structure.
        record
                sqlwarn0: Character;
                sqlwarn1: Character;
                sqlwarn2: Character;
                sqlwarn3: Character;
                sqlwarn4: Character;
                sqlwarn5: Character;
                sqlwarn6: Character;
                sqlwarn7: Character;
        end record;

type IISQLCA is
        record
                sqlcaid: String(1..8);
                sqlcabc: Integer;
                sqlcode: Integer;
                sqlerrm: IISQL_ERRM;
                sqlerrp: String(1..8);
                sqlerrd: IISQL_ERRD;
                sqlwarn: IISQL_WARN;
                sqlext:  String(1..8);
        end record;

sqlca: IISQLCA;
```

The nested record **sqlerrm** is a varying length character string consisting of the two variables **sqlerrml** and **sqlerrmc** described in the *SQL Reference Guide.* For a full description of all the SQLCA structure members, see the *SQL Reference Guide*.

The SQLCA is initialized at load-time. The **sqlcaid** and **sqlcabc** fields are initialized to the string "SQLCA" and the constant 136, respectively.

Note that the preprocessor is not aware of the record declaration. Therefore, you cannot use members of the record in an Embedded SQL statement. For example, the following statement, attempting to **insert** the string "SQLCA" into a table generates an error:

```
exec sql insert into
        employee (ename) -- This statement is illegal
        values (:sqlca.sqlcaid);
```

All modules written in Ada and other Embedded SQL languages share the same SQLCA.

Using the SQLCA for Error Handling

User-defined error, message and dbevent handlers offer the most flexibility for handling errors, database procedure messages, and database events. For more information, see Advanced Processing in this chapter.

However, you can do error handling with the SQLCA implicitly by using **whenever** statements, or explicitly by checking the contents of the SQLCA fields **sqlcode**, **sqlerrd**, and **sqlwarn0**.

Error Handling with the Whenever Statement

The syntax of the **whenever** statement is as follows:

> **exec sql whenever** *condition action;*

condition is **dbevent**, **sqlwarning**, **sqlerror**, **sqlmessage**, or **not found**. *action* is **continue**, **stop**, **goto** a label, **call** an Ada procedure, or **raise** an Ada exception. For a detailed description of this statement, see the *SQL Reference Guide.*

Embedded SQL/Ada provides the **raise** exception action as well as the regular SQL actions. You can use this instead of the less desirable **goto** action. Note that you should *not* declare the named exception in an SQL **declare section**.

For example:

```
exec sql include sqlca;
stmt_error: exception;
...
exec sql whenever sqlerror raise stmt_error;
...
-- Database statements
exception
        when stmt_error =>
              put_line("An error occurred.");
              ...
```

In Ada, all label, exception, and procedure names must be legal Ada identifiers, beginning with an alphabetic character. If the name is an Embedded SQL reserved word, specify it in quotes. Note that the label targeted by the **goto** action and the exception targeted by the **raise** action must be in the scope of all subsequent Embedded SQL statements until you encounter another **whenever** statement for the same action. This is necessary because the preprocessor may generate the Ada statement:

> **if (***condition)* ***then***
>   **goto** *label;* --**raise** *exception*
> **end if;**

after an Embedded SQL statement. If the scope of the label or exception is invalid, the Ada compiler generates an error.

The same scope rules apply to procedure names used with the **call** action. Note that the reserved procedure **sqlprint**, which prints errors or database procedure messages and then continues, is always in the scope of the program. When a **whenever** statement specifies a **call** as the action, the target procedure is called, and after its execution, control returns to the statement following the statement that caused the procedure to be called. Consequently, after handling the **whenever** condition in the called procedure, you may want to take some action, instead of merely issuing an Ada **return** statement. The Ada **return** statement causes the program to continue execution with the statement following the Embedded SQL statement that generated the error.

The following example demonstrates use of the **whenever** statements in the context of printing some values from the Employee table. The comments do not relate to the program but to the use of error handling.

```
-- I/O packages
with text_io; use text_io;

with integer_text_io; use integer_text_io;
with short_integer_text_io; use short_integer_text_io;

exec sql include sqlca;

procedure Db_Test is
        exec sql begin declare section;
                eno:        Short_Integer;
                ename:      String(1..20);
```

```
            age:        String(1..1);
    exec sql end declare section;

    sql_error: Exception;

    exec sql declare empcsr cursor for
        select eno, ename, age
        from employee;

--
-- Clean_Up: error handling procedure (print error
-- and disconnect).

procedure Clean_Up is
    exec sql begin declare section;
        errmsg: String(200);
    exec sql end declare section;

begin -- Clean_Up
    exec sql inquire_sql (:errmsg = errortext);
    put_line( "aborting because of error: " );
    put_line( errmsg );
    exec sql disconnect;

    raise sql_error; -- No return
end Clean_Up;

begin     -- Db_Test
--
-- An error when opening the personnel database
-- will cause the error to be printed and the
-- program to abort.
--

exec sql whenever sqlerror stop;
exec sql connect personnel;

-- Errors from here on will cause the program to
-- clean up.
exec sql whenever sqlerror call Clean_Up;

exec sql open empcsr;

put_line( "Some values from the ""employee""
         table.");

-- When no more rows are fetched, close the cursor.
exec sql whenever not found goto Close_Csr;

--
-- The last executable Embedded SQL statement was an
-- OPEN, so we know that the value of "sqlcode"
-- cannot be SQLERROR or NOT FOUND.
--

while (sqlca.sqlcode = 0) loop
    -- Loop is broken by NOT found
    exec sql fetch empcsr
        into :eno, :ename, :age;

    --
    -- These "put" statements do not execute after
    -- the previous FETCH returns the NOT FOUND
    -- condition.
    --
```

```
        put( eno );
        put( ", " & ename & ", ");
        put( age );
        new_line;
    end loop;
    --
    -- From this point in the file onwards, ignore all
    -- errors. Also turn off the NOT FOUND condition,
    -- for consistency.
    --

    exec sql whenever sqlerror continue;
    exec sql whenever not found continue;
<<Close_Csr>>
    exec sql close empcsr;
    exec sql disconnect;

    --
    -- "Sqlerror" is raised only in Clean_Up, which
    -- has already taken care of the error.
    --

    exception
        when sql_error =>
            null;       -- Just go away quietly
end Db_Test;
```

The Whenever Goto Action in Embedded SQL Blocks

An Embedded SQL block-structured statement is a statement delimited by the **begin** and **end** clauses. For example, the **select** loop and the **unloadtable** loops are both block-structured statements. You can terminate these statements only by the methods specified for the particular statement in the *SQL Reference Guide.* For example, the preprocessor terminates the **select** loop either when all the rows in the database result table have been processed or by an **endselect** statement, and the preprocessor terminates the **unloadtable** loop either when all the rows in the forms table field have been processed or by an **endloop** statement.

Therefore, if you use a **whenever** statement with the **goto** action in an SQL block, you must avoid going to a label outside the block. Such a **goto** causes the block to be terminated without issuing the runtime calls necessary to clean up the information that controls the loop. (For the same reason, you must not issue an Ada **return**, **exit**, **goto**, or **raise** statement that causes control to leave or enter an SQL block.) The target label of the **whenever goto** statement should be a label in the block. If however, it is a label for a block of code that cleanly exits the program, the above precaution need not be taken.

The above information does not apply to error handling for database statements issued outside an SQL block, nor to explicit hard-coded error handling. For an example of hard-coded error handling, see The Table Editor Table Field Application in this chapter.

Explicit Error Handling

The program can also handle errors by inspecting values in the SQLCA structure at various points. For further details, see the *SQL Reference Guide.*

The example on the following page is functionally the same as the previous example, except that the error handling is hard-coded in Ada statements.

```
-- I/O packages
with text_io; use text_io;
with integer_text_io; use integer_text_io;
with short_integer_text_io; use short_integer_text_io;

exec sql include sqlca;

procedure Db_Test is
    exec sql begin declare section;
        eno:   Short_Integer;
        ename: String(1..20);
        age:   String(1..1);
    exec sql end declare section;

    sql_error: Exception;
    not_found: constant := 100;

    exec sql declare empcsr cursor for
        select eno, ename, age
        from employee;

    --
    -- Clean_Up: Error handling procedure (print error
    -- and disconnect).
    --

    procedure Clean_Up( str: in String) is

        exec sql begin declare section;
            err_stmt: String(40) := str;
            errmsg:   String(200);
        exec sql end declare section;

    begin         -- Clean_Up
        exec sql inquire_sql (:errmsg = ERRORTEXT);
        put_line
             ( "Aborting because of error in " &
               err_stmt & ": ");
        put_line( errmsg );
        exec sql disconnect;

        raise sql_error; -- No return
    end Clean_Up;

begin -- Db_Test

    -- Exit if the database cannot be opened.
    exec sql connect personnel;
    if (sqlca.sqlcode < 0) then
        put_line( "Cannot access database.");
        raise sql_error;
    end if;

    -- Errors if cannot open cursor.
    exec sql open empcsr;
    if (sqlca.sqlcode < 0) then
```

```
            Clean_Up( "OPEN ""empcsr""" );
        end if;

        put_line("Some values from the ""employee"" table.");

        --
        -- The last executable Embedded SQL statement was an
        -- OPEN, so we know that the value of "sqlcode"
        -- cannot be SQLERROR or NOT FOUND.
        --

        while (sqlca.sqlcode = 0) loop
                                   -- Loop is broken by NOT FOUND
            exec sql fetch empcsr
                into :eno, :ename, :age;

            -- Do not print the last values twice
            if (sqlca.sqlcode < 0) then
                Clean_Up( "FETCH ""empcsr""" );
            elsif (sqlca.sqlcode <= NOT_FOUND) then
                put( eno );
                put( ", " & ename & ", ");
                put( age );
                new_line;
            end if;
        end loop;

        -- From this point in the file onwards, ignore all
        -- errors.

        exec sql close empcsr;
        exec sql disconnect;

        --
        -- "Sql_error" is raised only in Clean_Up, which has
        -- already taken care of the error, or in opening
        -- the database.
        --

        exception
            when sql_error =>
                null; -- Just go away quietly
    end Db_Test;
```

Determining the Number of Affected Rows

The third element of the SQLCA array **sqlerrd** indicates how many rows were affected by the last row-affecting statement. The following program fragment, which deletes all employees whose employee numbers are greater than a given number, demonstrates how to use **sqlerrd**:

```
procedure Delete_Rows( lower_bound: in Integer ) is
        exec sql begin declare section;
                lower_bound_num: integer := lower_bound;
        exec sql end declare section;

begin
        exec sql delete from employee
                where eno > :lower_bound_num;

        -- Print the number of employees deleted.
        put( sqlca.sqlerrd(3) );
        put_line( " (rows) were deleted.");
```

```
end Delete_Rows;
```

Using the SQLSTATE Variable

You can use the **SQLSTATE** variable in an ESQL/ Ada program to return status information about the last SQL statement that was executed. **SQLSTATE** must be declared in a declaration section and must be in uppercase. Also, it is valid across all sessions, so you only need to declare one **SQLSTATE** per application.

To declare this variable, use:

```
SQLSTATE: String(1..5);
```

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

Dynamic Programming for Ada

Ingres provides Dynamic SQL and Dynamic FRS to allow you to write generic programs. Dynamic SQL allows a program to build and execute SQL statements at runtime. For example, an application can include an expert mode in which the runtime user can type in select queries and browse the results at the terminal. Dynamic FRS allows a program to interact with any form at runtime. For example, an application can load in any form, allowing the runtime user to retrieve new data from the form and insert it into the database.

The Dynamic SQL and Dynamic FRS statements are described in the *SQL Reference Guide* and *Forms-based Application Development Tools User Guide*, respectively. This section discusses the Ada-dependent issues of Dynamic programming. For a complete example of using Dynamic SQL to write an SQL Terminal Monitor application, see The SQL Terminal Monitor Application in this chapter. For an example of using both Dynamic SQL and Dynamic FRS to browse and update a database using any form, see A Dynamic SQL/Forms Database Browser in this chapter.

This chapter is written for VAX/VMS Ada and makes use of the VAX/Ada data type definitions, in particular the **address** data type defined by the SYSTEM package.

The SQLDA Record

The SQLDA (SQL Descriptor Area) is used to pass type and size information about an SQL statement, an Ingres form, or a table field, between Ingres and your program.

To use the SQLDA, you should issue the **include sqlda** statement in front of each compilation unit containing references to the SQLDA. You cannot issue the **include sqlda** statement inside a compilation unit because the statement causes the preprocessor to generate Ada **with** and **use** clauses, which are not legal in that context. The package specified by the **include sqlda** statement is called ESQLDA and contains the SQLDA record type definition. The package does *not* declare an SQLDA record variable; your program must declare a variable of the specified type. You can also code the SQLDA record variable directly instead of using the **include sqlda** statement. When coding the declaration yourself, you can choose any name for the record type.

The definition of the SQLDA record (as specified in package ESQLDA) is:

```
-- IISQ_MAX_COLS - Maximum number of columns
-- returned from Ingres
IISQ_MAX_COLS: constant := 1024;

-- Data Type Codes
IISQ_DTE_TYPE: constant := 3;  -- Date - Output
IISQ_MNY_TYPE: constant := 5;  -- Money - Output
IISQ_DEC_TYPE: constant := 10; -- Decimal - Output
IISQ_CHA_TYPE: constant := 20; -- Char-Input, Output
IISQ_VCH_TYPE: constant := 21; -- Varchar- Input, Output
IISQ_LVCH_TYPE:constant := 22; -- Long Varchar - Output
IISQ_INT_TYPE: constant := 30; -- Integer-Input, Output
IISQ_FLT_TYPE: constant := 31; -- Float-Input, Output
IISQ_OBJ_TYPE: constant := 45; -- 4GL Object: Output
IISQ_HDLR_TYPE:constant := 46; -- Datahandler -Inp/Output
IISQ_TBL_TYPE: constant := 52; -- Table field - Output
IISQ_DTE_LEN:  constant := 25; -- Date length

-- Address constant to avoid SYSTEM requirement
 IISQ_ADR_ZERO: constant ADDRESS := ADDRESS_ZERO;

type IISQL_NAME is -- Varying length name
 record
             sqlnamel: Short_Integer;
             sqlnamec: String(1..34);
 end record;

type IISQL_VAR is
               -- Single element of SQLDA column/variable
  record
            sqltype: Short_Integer;
            sqllen:  Short_Integer;
            sqldata: Address; -- Address of any type
            sqlname: IISQL_NAME;
  end record;

type IISQL_VARS is -- Array of IISQL_VAR elements
  array(Short_Integer range <>) of IISQL_VAR;

-- IISQLDA - SQLDA with varying number of
-- result variables.
-- Default is maximum number (IISQ_MAX_COLS).
type IISQLDA (sqln: Short_Integer := IISQ_MAX_COLS) is
     record
             sqldaid: String(1..8);
             sqldabc: Integer;
             sqld:    Short_Integer;
             sqlvar:  IISQL_VARS(1..sqln);
     end record;
```

```
-- Generic SQL-compatible record layout description.
-- for IISQLDA use
 record
     sqldaid at 0 range 0..63;
                       -- Bytes 0..7 = String(1..8);
     sqldabc at 8 range 0..31;
                      -- Bytes 8..11 = Integer;
     sqln at 12 range 0..15;
                     -- Bytes 12..13 = Short_Integer;
     sqld at 14 range 0..15;
                     -- Bytes 14..15 = Short_Integer;
 end record;

--
-- IISQHDLR - Structure type with function pointer and
-- function argument for the DATAHANDLER
--
type IISQHDLR is
    record
         sqlarg:    Address;
         sqlhdlr:   Address;
    end record;
```

Record Definition and Usage Notes:

- The record type definition of the SQLDA is called IISQLDA. This is done so that an SQLDA variable can be called “SQLDA” without causing an Ada compile-time conflict. You are not required to call your SQLDA record variable “SQLDA.”
- The record type definition includes a discriminant, **sqln**. This discriminant indicates how many elements are allocated in the varying length array, **sqlvar**. The VAX/Ada default is to allocate space for the discriminant at the start of the record. In order to enforce a compatible SQLDA record layout with the Ingres runtime system and other embedded languages, an Ada representation clause is issued. This clause causes the discriminant, **sqln**, to be placed among the record components as defined in the *SQL Reference Guide*. This is described in more detail later.
- The varying length **sqlvar** array, whose length is determined by the discriminant **sqln**, has a default size of IISQ_MAX_COLS (1024) elements. If you declare an SQLDA record variable of type IISQLDA without a discriminant constraint, then the program will have declared a record with IISQ_MAX_COLS elements.
- Note that the **sqlvar** array begins at subscript 1. If you code your own SQLDA record you can specify any number for a lower bound.
- The **sqldata** and **sqlind** record components are declared as addresses. You must set these to point at variables using the Ada **address** attribute. You must set the addresses before using the SQLDA to retrieve or set Ingres data in the database or in a form. Because you can use null indicators, a constant (IISQ_ADR_ZERO) is provided so that you can set **sqlind** to the zero address without including the SYSTEM package.

- If your program defines its own SQLDA record type you must verify that the **internal** record layout is identical to that of the IISQLDA record type, although you can declare a different number of **sqlvar** elements. You need not declare the type with a discriminant, but if you do, you must issue an Ada representation clause to force the 2-byte discriminant to be placed between **sqldabc** and **sqld**. The internal layout of the IISQLDA record type is equivalent to the following pseudo Ada declaration:

  ```
  type IISQLDA_RECORD_LAYOUT is
          record
              sqldaid: String(1..8);
              sqldabc: Integer;
              sqln:    Short_Integer; -- See FOR clause
              sqld:    Short_Integer;
              sqlvar:  IISQL_VARS(1..sqln);
          end record;
  ```

  Consequently, if you declare a record type *without* a discriminant (that is, with a fixed length array of **sqlvar** elements), you should position the **sqln** component as shown above.

- The **sqlname** component is a varying length character string consisting of a length and data area. The **sqlnamec** component contains the name of a result field or column after a **describe** or **prepare into** statement. The length of the name is specified by **sqlnamel**. The characters in **sqlnamec** are padded with blanks. You can also set the **sqlname** component by a program using Dynamic FRS. The program is not required to pad **sqlnamec** with blanks. For more information, see Setting SQLNAME for Dynamic FRS in this chapter.

- The list of type codes represent the types that are returned by the **describe** statement, and the types used by the program when retrieving or setting data with an SQLDA. The type code IISQ_TBL_TYPE indicates a table field and is set by the FRS when describing a form that contains a table field.

Declaring an SQLDA Record Variable

Once you have included (or hard-coded) the SQLDA type definition, the program can declare an SQLDA record variable. You must declare this record variable outside of a **declare section**, as the preprocessor does not understand the special meaning of the SQLDA record or the IISQLDA record type. When you use the variable in the context of a Dynamic SQL or Dynamic FRS statement, the preprocessor accepts *any* object name, and assumes that the variable refers to a legally declared SQLDA record variable, for which storage has been allocated.

If your program requires an SQLDA variable with IISQ_MAX_COLS **sqlvar** elements, you can accomplish this by declaring the variable without a discriminant constraint. Unlike other languages, an Ada program cannot set the value of **sqln**. Because **sqln** is a type discriminant, its value is implicit from the declaration.

For example:

```
exec sql include sqlda;
sqlda: IISQLDA;
         -- Default sets sqlda.sqln to IISQ_MAX_COLS
         -- This is outside of a DECLARE SECTION.
...
exec sql describe s1 into :sqlda;
```

However, when you do not use a discriminant constraint in the record declaration, you cannot later use an Ada **renames** statement as a shorthand into the **sqlvar** array. A shorthand can be desirable over continued long references such as:

```
sqlda.sqlvar(i).sqldata
sqlda.sqlvar(i).sqlname.sqlnamec
```

For example, the above declaration of the SQLDA is equivalent to:

```
exec sql include sqlda;
max_sqlda: IISQLDA(IISQ_MAX_COLS); -- Includes constraint
...
exec sql describe s1 into :max_sqlda;
...
for i in 1..max_sqlda.sqld loop;
    declare
        sqv: IISQL_VAR renames max_sqlda.sqlvar(i);
    begin
        -- Use shorthand sqv instead of
        -- max_sqlda.sqlvar(i)
    end;
end loop;
```

If you require an SQLDA with a *different* number of **sqlvar** elements, then you can use a different discriminant constraint. For example:

```
sqlda_10:IISQLDA(10); -- Implicitly sets sqlda.sqln to 10
```

You can also dynamically allocate an SQLDA with a varying number of **sqlvar** elements. In the following example an SQLDA access variable is declared. Note that when you reference the variable in the **describe** statement the Ada **all** clause is used, as the preprocessor expects a valid SQLDA record variable and not a *pointer* to a record:

```
exec sql include sqlda;

procedure Process_Dynamic_SQL (num_cols: in

       Short_Integer) is

    type SQLDA_PTR is access IISQLDA;

    sp: SQLDA_PTR;

begin

    sp := new SQLDA_PTR(num_cols);

    ...

    exec sql describe s1 INTO :sp.all; -- Note .all

    ...

end Process_Dynamic_SQL;
```

As long as you use the IISQLDA record type, or a derivative of that type, you need not be concerned with the SQLDA record layout. When you code your own SQLDA record type then you must confirm that the internal record layout is identical to that of the IISQLDA record. One reason you might code your own SQLDA record type is to avoid the runtime overhead required to validate offsets into a record variable containing a varying length array, such as **sqlvar**. You may prefer a fixed length record variable without a discriminant. In that case you must declare the **sqln** component in the correct position, and you must *explicitly* set the value of **sqln** in order for the **describe** statement to succeed. For example:

```
max_sq: constant := 50;
type fixed_sqlda_max is -- Layout is correct
    record
         my_sqid:  String(1..8);
         my_sqbc:  Integer;
         my_vars:  Short_Integer;  -- Equivalent to sqln
         res_vars: Short_Integer;  -- and SQLD
         col_vars: IISQL_VARS(1..MAX_SQ);
    end record;

my_sq: FIXED_SQLDA_MAX;

...
my_sq.my_vars := MAX_SQ; -- Size must be set

...

exec sql describe s1 into :my_sq;
```

In the above record type definition, the names of the record components are not the same as those of the IISQLDA record, but their layout is identical.

As shown above there are a variety of ways to declare an SQLDA record variable. Names of record components are not important; internal component layout, however, is critical.

Using the SQLVAR

The *SQL Reference Guide* discusses the legal values of the **sqlvar** array. The **describe** and **prepare into** statements set the type, length, and name information of the SQLDA. This information refers to the result columns of a prepared **select** statement, the fields of a form, or the columns of a table field. When the program uses the SQLDA to retrieve or set Ingres data, it must assign type and length information that now refers to the variables being pointed at by the SQLDA.

Ada Variable Type Codes

The type codes listed above (as Ada constants) are the types that describe Ingres result fields and columns. For example, the SQL types **date**, **decimal**, **long varchar** and **money** do not describe a program variable, but rather data types that are compatible with the Ada character string and numeric data types. When these types are returned by the describe statement, the type code must be changed to a compatible Ada or ESQL/Ada type.

The following table describes the data type codes to use with Ada variables that are *pointed* at by the **sqldata** pointers:

The SQL Type Codes

Ada Type	SQL Type Codes (sqltype)	SQL Length (sqllen)
Short_Short_Integer	IISQ_INT_TYPE	1
Short_Integer	IISQ_INT_TYPE	2
Integer	IISQ_INT_TYPE	4
Float	IISQ_FLT_TYPE	4
Long_Float	IISQ_FLT_TYPE	8
String(1..LEN)	IISQ_CHA_TYPE	LEN
IISQLHDLR	IISQ_HDLR_TYPE	0

As described in Ada Variables and Data Types, all other types are compatible with the above Ada data types. For example, you can retrieve an SQL **date** into an Ada **string** variable, while you can retrieve **money** into a **long_float** variable.

You can specify nullable data types (those variables that are associated with a null indicator) by assigning the negative of the type code to **sqltype**. If the type is negative then you must point at a null indicator by **sqlind**. The type of the null indicator must be a 2-byte integer, **short_integer**, or a derivative of that type. For information on how to declare and use a null indicator in Ada, see Ada Variables and Data Types in this chapter.

Character data and the SQLDA have the same rules as character data in regular Embedded SQL statements. For details of character string processing in SQL, see Ada Variables and Data Types in this chapter.

Pointing at Ada Variables

In order to fill an element of the **sqlvar** array, you must set the type information, and assign a valid address to **sqldata**. The address must be that of a legally declared and allocated variable. If the element is nullable then the corresponding **sqlind** component must point at a legally declared null indicator.

In order to assign addresses to **sqldata** and **sqlind**, you should use the Ada **address** attribute or some other function that yields an address. Because null indicators are not always required, you can sometimes assign **sqlind** a zero-valued address. This can be accomplished by assigning to **sqlind** the constant IISQ_ADR_ZERO, as defined in the ESQLDA package, or the constant ADDRESS_ZERO, if you have included the SYSTEM package.

When assigning addresses, you should be careful to follow the guidelines set by the VAX/VMS Ada. For example, you should not reference a variable whose lifetime has expired, and you should not access storage beyond the allocated amount. You can use the **volatile** pragma when addressing variables local to a subprogram body in order to prevent the compiler from referring to a local copy of a variable. When dynamically allocating result storage variables, you may want to use the **controlled** pragma together with an instantiation of the generic **unchecked_deallocation** procedure. The SQL Terminal Monitor Application and A Dynamic SQL/Forms Database Browser, which use Dynamic SQL and the SQLDA, do not use any of these pragmas, but rely on the rules defined in the *VAX Ada Programmer's Runtime Reference Manual*.

The following example fragment sets the type information of and points at a 4-byte integer variable, an 8-byte nullable floating-point variable, and a character slice (sub-string) whose length is specified by **sqllen**. This example demonstrates how a program can maintain a pool of available variables, such as large arrays of a few different typed variables and a large string space. When a variable is allocated out of the pool the next available spot is incremented:

```
exec sql include SQLDA;
max_pool: constant := 50;
sqlda: IISQLDA(MAX_POOL);
...

-- Numeric and string pool declarations.
ind_store: array(1..MAX_POOL) of
                                  Short_Integer;               -- Indicators
current_ind: Integer := 0;
int4_store: array(1..MAX_POOL) of Integer;  -- Integers
current_int: Integer := 0;
flt8_store: array(1..MAX_POOL) of Long_Float; -- Floats
current_flt: Integer := 0;
char_store: String(1..3000);            -- String buffer
current_chr: Integer := 1;

...
sqlda.sqlvar(1).sqltype := IISQ_INT_TYPE;
                                                  -- 4-byte integer
sqlda.sqlvar(1).sqllen  := 4;
```

```
sqlda.sqlvar(1).sqldata:= int4_store(current_int)'Address;
sqlda.sqlvar(1).sqlind := IISQ_ADR_ZERO;
current_int            := current_int + 1;
                                                    -- Update integer pool

sqlda.sqlvar(2).sqltype := -IISQ_FLT_TYPE;
                                 -- 8-byte nullable float
sqlda.sqlvar(2).sqllen := 8;
sqlda.sqlvar(2).sqldata :=flt8_store(current_flt)'Address;
sqlda.sqlvar(2).sqlind := ind_store(current_ind)'Address;
current_flt          := current_flt + 1; -- Update float
current_ind          := current_ind + 1; -- and indicator
                                          -- pool

--
-- SQLLEN has been assigned by DESCRIBE to be the length
-- of a specific result column. This length is used to
-- pick off a slice out of the large string buffer.
-- The character counter is then updated.
--

sqlda.sqlvar(3).sqltype := IISQ_CHA_TYPE;
sqlda.sqlvar(3).sqldata
                         := char_store(current_chr)'Address;
sqlda.sqlvar(3).sqlind := IISQ_ADR_ZERO;
current_chr := current_chr + sqlda.sqlvar(3).sqllen;
```

Of course, in the above example, you must verify enough pool storage before referencing each cell of the different arrays in order to prevent **sqldata** and **sqlind** from pointing at undefined storage. For demonstrations of this method, see The The SQL Terminal Monitor Application and A Dynamic SQL/Forms Database Browser in this chapter.

You may also set the SQLVAR to point to a datahandler for large object columns.

If you code your own SQLDA and, in place of **sqldata**, you declare a variant record of access types to a subset of different data types you may find that you can use the Ada allocator, **new**, and basic access type assignments. If you confirm that the layout of the record with the variant component is the same as that of IISQLDA, then you can use this type of record as an SQLDA without the need to access object addresses. This approach is not discussed further.

Setting SQLNAME for Dynamic FRS

Using the **sqlvar** with Dynamic FRS statements requires a few extra steps. These extra steps relate to the differences between Dynamic FRS and Dynamic SQL and are described in the *Forms-based Application Development Tools User Guide* and the *SQL Reference Guide.*

When using the SQLDA in a forms input or output **using** clause, you must set the value of **sqlname** to a valid field or column name. If a previous **describe** statement has set the name, it must be retained or reset by the program. If the name refers to a hidden column in a table field, the program must set **sqlname** directly. If your program sets **sqlname** directly, it must also set **sqlnamel** and **sqlnamec**. The name portion need not be padded with blanks.

For example, a dynamically named table field has been described, and the application always initializes any table field with a hidden 6-byte character column called "rowid." The code used to retrieve a row from the table field including the hidden column and **_state** variable must construct the two named columns:

```
...
rowid:    String(1..6);

rowstate: Integer;
...
exec frs describe table :formname :tablename into :sqlda;

...
sqlda.sqld := sqlda.sqld + 1;
col_num := sqlda.sqld;

-- Set up to retrieve rowid
sqlda.sqlvar(col_num).sqltype          := IISQ_CHA_TYPE;
sqlda.sqlvar(col_num).sqllen           := 6;
sqlda.sqlvar(col_num).sqldata          := rowid'Address;
sqlda.sqlvar(col_num).sqlind           := IISQ_ADR_ZERO;
sqlda.sqlvar(col_num).sqlname.sqlnamel := 5;
sqlda.sqlvar(col_num).sqlname.sqlnamec(1..5) := "rowid";

sqlda.sqld := sqlda.sqld + 1;
col_num := sqlda.sqld;

-- Set up to retrieve _STATE
sqlda.sqlvar(col_num).sqltype := IISQ_INT_TYPE;
sqlda.sqlvar(col_num).sqllen := 4;
sqlda.sqlvar(col_num).sqldata := rowstate'Address;
sqlda.sqlvar(col_num).sqlind := IISQ_ADR_ZERO;
sqlda.sqlvar(col_num).sqlname.sqlnamel := 6;
sqlda.sqlvar(col_num).sqlname.sqlnamec(1..6) := "_state";

...
exec frs getrow :formname :tablename using descriptor :sqlda;
```

Advanced Processing

This section describes user-defined handlers. It includes information about user-defined error, dbevent, and message handlers as well as data handlers for large objects.

User-Defined Error, DBevent, and Message Handlers

You can use user-defined handlers to capture errors, messages, or events during the processing of a database statement. Use these handlers instead of the **sql whenever** statements with the SQLCA when you want to do the following:

- Capture more than one error message on a single database statement.
- Capture more than one message from database procedures fired by rules.
- Trap errors, events, and messages as the DBMS raises them. If an event is raised when an error occurs during query execution, the WHENEVER mechanism detects only the error and defers acting on the event until the next database statement is executed.

User-defined handlers offer you flexibility. If, for example, you want to trap an error, you can code a user-defined handler to issue an **inquire_sql** to get the error number and error text of the current error. You can then switch sessions and log the error to a table in another session; however, you must switch back to the session from which the handler was called before returning from the handler. When the user handler returns, the original statement continues executing. User code in the handler cannot issue database statements for the session from which the handler was called.

The handler must be declared to return an integer. However, the preprocessor ignores the return value.

Syntax Notes:

The following syntax describes the three types of handlers:

```
exec sql set_sql (errorhandler   = error_routine|0);
exec sql set_sql (dbeventhandler = event_routine|0);
exec sql set_sql (messagehandler = message_routine|0);
```

Errorhandler, dbeventhandler, and messagehandler denote a user-defined handler to capture errors, events, and database messages respectively, as follows:

- error_routine is the name of the function the Ingres runtime system calls when an error occurs.
- event_routine is the name of the function the Ingres runtime system calls when an event is raised.
- message_routine is the name of the function the Ingres runtime system calls whenever a database procedure generates a message.

  Errors that occur in the error handler itself do not cause the error handler to be re-invoked. You must use **inquire_sql** to handle or trap any errors that may occur in the handler.

- Unlike regular variables, the handler must not be declared in an ESQL declare section; therefore, do not use a colon before the handler argument. (However, you must declare the handler to the compiler.)
- If you specify a zero (0) instead of a name, the zero will unset the handler.

User-defined handlers are also described in the *SQL Reference Guide.*

Declaring and Defining User-Defined Handlers

The following example shows how to declare a handler for use in the **set_sql errorhandler** statement for ESQL/Ada:

```
exec sql include sqlca;

package Error_Trap is
        function Error_Func return Integer;
        pragma export_function (Error_Func);
end Error_Trap;

with text_io; use text_io;
with integer_text_io; use integer_text_io;

package body Error_Trap is
        function Error_Func return Integer is
        exec sql begin declare section;
                errnum : Integer;
        exec sql end declare section;
        begin
                exec sql inquire_sql(:errnum = ERRORNO);
                put ("Error number is: ");
                put (Errnum);
        end Error_Func;
end;

with Error_Trap; use Error_Trap;
procedure TEST is
begin
        exec sql connect dbname;

        exec sql set_sql (ERRORHANDLER = Error_Func);
          --
          -- ESQL will generate
          -- IILQshSetHandler ( 1, Error_Func'Address );
          --
          . . .
end;
```

User-Defined Data Handlers for Large Objects

You can use user-defined datahandlers to transmit large object column values to or from the database a segment at a time. For more details on Large Objects, the **datahandler** clause, the **get data** statement and the **put data** statement, see the *SQL Reference Guide* and the *Forms-based Application Development Tools User Guide*.

ESQL/Ada Usage Notes

- The datahandler, and datahandler argument, should not be declared in an ESQL declare section. Therefore do not use a colon before the datahandler or its argument.
- You must ensure that the datahandler argument is a valid Ada variable address. ESQL will not do any syntax or datatype checking of the argument.
- The datahandler must be declared to return an integer. However, the actual return value will be ignored.

DATAHANDLERS and the SQLDA

You may specify a user-defined datahandler as an SQLVAR element of the SQLDA, to transmit large objects to or from the database. The eqsqlda.h file included with the **include sqlda** statement defines an IISQLDHDLR type which may be used to specify a datahandler and its argument. It is defined:

```
--
-- IISQLHDLR - Structure type with function pointer and
--             function argument for the DATAHANDLER.
--
type IISQLHDLR is
    record
       sqlarg:  Address;-- Optional argument to pass
       sqlhdlr: Address;--User-defined datahandler function
    end record;
```

The file does not declare an IISQLHDLR variable; the program must declare a variable of the specified type and set the values:

```
-- Declare argument to be passed to datahandler
      hdlr_arg:     Hdlr_Rec;

-- Declare IISQLHDLR
      data_handler: IISQLHDLR;
-- Declare Get_Handler function to return an integer
function Get_Handler(info: Hdlr_Rec) return Integer
        data_handler.sqlhdlr = Get_Handler'Address;
        data_handler.sqlarg       = hdlr_arg'Address;
```

The **sqltype**, **sqlind** and **sqldata** fields of the SQLVAR element of the SQLDA should then be set as follows:

```
sqlda.sqlvar(i).sqltype := IISQ_HDLR_TYPE;
sqlda.sqlvar(i).sqldata := data_handler'Address;
sqlda.sqlvar(i).sqlind  := indvar'Address'
```

Sample Programs

The programs in this section are examples of how to declare and use user-defined datahandlers in an ESQL/Ada program. There are examples of a handler program, a put handler program, a get handler program and a dynamic SQL handler program.

Handler Program

This example assumes that the book table was created with the statement:

```
exec sql create table book (chapter_num integer,
     chapter_name char(50), chapter_text long varchar);
```

This program inserts a row into the book table using the data handler Put_Handler to transmit the value of column chapter_text from a text file to the database in segments. Then it selects the column chapter_text from the table book using the data handler Get_Handler to retrieve the chapter_text column a segment at a time:

```
package DataHdlrPkg is
    type Hdlr_Rec is

    record
      argstr:   String(1..100);
      argint:   Integer;  -- 4-byte integers
    end record;

    function Put_Handler(info: Hdlr_Rec) return Integer;
    function Get_Handler(Info: Hdlr_Rec) return Integer;
    pragma export_function(Put_Handler);
    pragma export_function(Get_Handler);

end DataHdlrPkg;

with DataHdlrPkg;      use DataHdlrPkg;

procedure handler is

    exec sql include sqlca;

-- Do not declare the datahandlers nor the datahandler
-- argument to the ESQL preprocessor

    hdlr_arg: Hdlr_Rec;

-- Null indicator for datahandler must be declared to
-- ESQL

--

    exec sql begin declare section;
      indvar:     Short_integer;
      chapter_num: Integer;
    exec sql end declare section;

-- Insert a long varchar value chapter_text into the
-- table book using the datahandler Put_Handler
-- The argument passed to the datahandler is the
-- address of the record hdlr_arg
```

```
    ...

    exec sql insert into book (chapter_num, chapter_name,
                     chapter_text)
         values (5, 'One Dark and Stormy Night',
            Datahandler(Put_handler(hdlr_arg)));

-- Select the column chapter_num and the long varchar
-- column chapter_text from the table book.
-- The datahandler (Get_Handler) will be invoked
-- for each non-null value of the column chapter_text
-- retrieved. For null values the indicator
-- variable will be set to "-1" and the datahandler
-- will not be called.
    ...
    exec sql select chapter_text into
        :chapter_num,
        datahandler (Get_Handler(hdlr_arg)) :indvar
        from book;

    exec sql begin;
        process row ...
    exec sql end;
    ...

end handler;
```

Put Handler

This example shows how to read the long varchar chapter_text from a text file and insert it into the database a segment at a time:

```
function Put_Handler(info: Hdlr_Rec) return Integer is
    exec sql begin declare section;
            seg_buf:   String(1..1000);
            seg_len:   Integer;
            data_end:  Integer;
    exec sql end declare section;

    process information passed in via the info
            record...
    open file ...

    data_end := 0;

    while (not end-of-file) loop

            read segment of less than 1000 chars from
           file into seg_buf...

            if (end-of-file) then
                data_end := 1;
            end if;

        exec sql put data (segment = :seg_buf,
                    segmentlength = :seg_len,
                         dataend = :data_end);

    end loop;

    ...
    close file ...
    set info record to return appropriate values...
    ...
```

```
    return 0;
end Put_Handler;
```

Get Handler

This example shows how to get the long varchar chapter_text from the database and write it to a text file:

```
function Get_Handler(info: Hdlr_Rec) return Integer is
    exec sql begin declare section;
            seg_buf:      String(1..100);
            seg_len:      Integer;
            data_end:     Integer;
            max_len:      Integer;
    exec sql end declare section;

    ...
    process information passed in via the
                info record....
    open file...

  -- Set a maximum segment length of 2000 bytes
  data_end := 0;

    while (data_end = 0) loop
            exec sql get data (:seg_buf = segment,
                              :seg_len =segmentlength,
                             :data_end = dataend)
                         with maxlength = :max_len;

        write segment to file ...
    end loop;

      . . .
      set info record to return appropriate values...

      . . .

      return 0;
end Get_Handler;
```

Dynamic SQL Handler Program

The following is an example of a dynamic SQL handler program. This program fragment shows the declaration and usage of a datahandler in a dynamic SQL program, using the SQLDA. It uses the datahandler Get_Handler and the HDLR_PARAM structure described in the previous example:

```
with DataHdlrPkg;      use DataHdlrPkg;

procedure Dynamic_hdlr

        exec sql include sqlca;
        exec sql include sqlda;

-- Do not declare the datahandlers nor the datahandler
-- argument to the ESQL preprocessor.

-- Declare argument to be passed to datahandler
```

```
        hdlr_arg:     Hdlr_Rec;

-- Declare SQLDA and IISQLHDLR

     sqlda:   IISQLDA(IISQ_MAX_COLS);
     data_handler: IISQLHDLR;

        col_num:      Integer;
        base_type:    Integer;

-- Declare null indicator to ESQL

        exec sql begin declare section;
              indvar:   Short_Integer;
                stmt_buf: String(100);
        exec sql end declare section;

        ...

--   Set the IISQLHDLR structure with the appropriate
-- datahandler and datahandler argument.

        data_handler.sqlhdlr = Get_Handler'Address;
        data_handler.sqlarg  = hdlr_arg'Address;

-- Describe the statement into the SQLDA.
        stmt_buf := "select * from book";
        exec sql prepare stmt from :stmt_buf;
        exec sql describe stmt into sqlda;
        . . .

-- Determine the base_type of the SQLDATA variables.
        for col_num in 1..sqlda.sqld loop

            if (sqlda.sqlvar(col_num).sqltype > 0) then
                base_type := sqlda.sqlvar(col_num).sqltype;

            else
                base_type := -sqlda.sqlvar(col_num).sqltype;
            end if;

-- Set the sqltype, sqldata and sqlind for each column
-- The long varchar column chapter_text will be set
-- to use a datahandler.

            if (base_type = IISQ_LVCH_TYPE) then
                 sqlda.sqlvar(col_num).sqltype := IISQ_HDLR_TYPE;
                 sqlda.sqlvar(col_num).sqldata :=
                            data_handler'Address;
                 sqlda.sqlvar(col_num).sqlind  := indvar'Address'
            else
            . . .

            end if;

        end loop;

-- The Datahandler (Get_Handler) will be invoked for
-- each non-null value of column chapter_text
-- retrieved. For null values the indicator variable
-- will be set to "-1" and the datahandler will not
-- be called.

    . . .
```

```
    exec sql execute immediate :stmt_buf using :sqlda;
    exec sql begin;

        process row....
        exec sql end;
        . . .

end Dynamic_hdlr;
```

Preprocessor Operation

This section describes the operation of the Embedded SQL preprocessor for Ada and the steps required to create, compile, and link an Embedded SQL program.

Include File Processing

The Embedded SQL **include** statement provides a means to include external packages and source files in your program's source code. Its syntax is:

exec sql include *filename;*

where *filename* is a quoted string constant specifying a file name or a logical name that points to the file name. If you do not specify an extension to the file name (or to the file name pointed at by the logical name), the default Ada input file extension ".sa" is assumed.

This statement is used to include variable declarations or package specifications. For more details on the **include** statement, see the *SQL Reference Guide.*

Including and Processing Variable Declarations

If issued in a declaration section, the **include** statement can only be used to include variable declarations. The included file is preprocessed, and Ada output is generated into the parent file.

For example, a file called "employee.dcl" containing a record declaration generated by DCLGEN can be included into the source code as follows:

```
exec sql begin declare section;
       exec sql include 'employee.dcl';
    -- more declarations
exec sql end declare section;
```

The employee.dcl file is preprocessed into the parent output file.

Including and Processing Package Specifications

If issued outside a declaration section, the **include** statement can only be used to include package specifications. The preprocessor reads the specified file, processing all variables declared in the package, and generates the Ada **with** and **use** clauses using the last component of the file name (excluding the file extension) as the package name. If the last component of the file name has a trailing underscore, as is standard in VAX/VMS Ada package specification files, then the preprocessor removes that trailing underscore in the generated context clauses. The preprocessor does not generate an output file, because it is assumed that the package specification has already been compiled.

Note:

- Each package must be in a separate source file.
- Nothing but the package should be in that source file (no other variable declarations, etc).
- There are no limitations on what can be in a package (you may define types, etc).

The following example demonstrates the **include** statement. Assume that the specification of package “employee” is in a employee_.sa file and that a procedure “empentry” is in the empentry.sa file:

Contents of employee_.sa:

```
package employee is
    exec sql begin declare section;
                ename:   String(1..20);
                eage:    Integer;
                esalary: Float;
    exec sql end declare section;
end employee;
```

Contents of empentry.sa:

```
exec sql include '[joe.neil.empfiles]employee_.sa';
procedure empentry is
begin
    -- Statements using variables in package
    -- "employee"
end empentry;
```

The Embedded SQL/Ada preprocessor modifies the **include** line to the Ada **with** and **use** clauses by extracting the last component of the file name:

```
with employee;    use employee;
```

The above two clauses appear in the empentry.ada output file. The preprocessor does not generate an output file for “employee_.sa,” and the employee package must have already been compiled in order to compile the empentry.ada file.

Assuming that the employee_.sa and empentry.sa files appear as shown above, you should execute the following sequence of VMS commands in order to compile “empentry.ada”:

```
esqla employee_.sa
esqla empentry.sa
ada employee_.ada
ada empentry.ada
```

You must still follow the Ada rules specifying the order of compilation. The Embedded SQL preprocessor does not affect these compilation rules.

Coding Requirements for Writing Embedded SQL Programs

The following sections describe coding requirements for writing Embedded SQL Programs.

Comments Embedded in Ada Output

Each Embedded SQL statement generates one comment and a few lines of Ada code. You may find that the preprocessor translates 50 lines of Embedded SQL into 200 lines of Ada. This can confuse the program developer trying to debug the original source code. To facilitate debugging, each group of Ada statements associated with a particular Embedded SQL statement is delimited by a comment corresponding to the original Embedded SQL source. Each comment is one line in length and informs the reader of the file name, line number, and type of statement in the original source file.

Embedded SQL Statements that Do Not Generate Code

The following Embedded SQL declarative statements do not generate any Ada code:

declare cursor
declare statement
declare table
whenever

These statements must not contain labels. Also, they must not be coded as the only statements in Ada constructs that do not allow *empty* statements. For example, coding a **declare cursor** statement as the only statement in an Ada **if** statement causes compiler errors:

```
if (using_database) then
        exec sql declare empcsr cursor for
                select ename from employee;
else
        put_line("You have not accessed the database.");
end if;
```

The preprocessor generates the code:

```
if (using_database) then
else
        put_line("You have not accessed the database.");
end if;
```

This is an illegal use of the Ada **if-then-else** statement.

Command Line Operations

The following sections describe how to turn an embedded SQL/Ada source program into an executable program. The commands that preprocess, compile, and link a program are also described.

The Embedded SQL Preprocessor Command

The Embedded SQL/Ada preprocessor is invoked by the following command line:

esqla {*flags*} {*filename*}

where *flags* are

Flag	Description
-d	Adds debugging information to the runtime database error messages generated by Embedded SQL. The source file name, line number, and statement in error are displayed with the error message.
-f[*filename]*	Writes preprocessor output to the named file. If you do not specify a *filename*, the output is sent to standard output, one screen at a time.
-l	Writes preprocessor error messages to the preprocessor's listing file, as well as to the terminal. The listing file includes preprocessor error messages and your source text in a file named *filename**.lis***, where *filename* is the name of the input file.
-lo	Acts like the **-l** flag, but the listing file also includes the generated Ada code.
-?	Shows what command line options are available for Embedded SQL/Ada.

Flag	Description
-s	Reads input from standard input and generates Ada code to standard output. This is useful for testing unfamiliar statements. If you specify the **-l** option with this flag, the listing file is called "stdin.lis." To terminate the interactive session, type **Ctrl Z**.
-sqlcode	Indicates the file declares an integer variable named SQLCODE to receive status information from SQL statements. That declaration need not be in an exec sql begin/end declare section. This feature is provided for ISO Entry SQL92 conformity However, the ISO Entry SQL92 specification describes **SQLCODE** as a "deprecated feature" and recommends using the **SQLSTATE** variable.
-**nosqlcode**	Tells the preprocessor not to assume the existence of a status variable named **SQLCODE**. The flag **-nosqlcode** is the default.
-w	Prints warning messages.
-wopen	This flag is identical to **-wsql=open**. However, **-wopen** is supported only for backwards capability. Refer to **-wsql=open** below for more information.
-**wsql=entry_ SQL92**	Causes the preprocessor to flag any usage of syntax or features that do not conform to the ISO Entry SQL92 entry level standard. (This is also known as the "FIPS flagger" option.)
-**wsql**=**open**	Use *open* only with OpenSQL syntax. **-wsql = open** generates a warning if the preprocessor encounters an Embedded SQL statement that does not conform to OpenSQL syntax. (For OpenSQL syntax, see the *OpenSQL Reference Guide*.) This flag is useful if you intend to port an application across different Enterprise Access products. The warnings do not affect the generated code and the output file may be compiled. This flag does not validate the statement syntax for any Enterprise Access product whose syntax is more restrictive than that of OpenSQL.

The Embedded SQL/Ada preprocessor assumes that input files are named with the extension ".sa". You can override this default by specifying the file extension of the input file(s) on the command line. The output of the preprocessor is a file of generated Ada statements with the same name and the extension ".ada".

If you enter the command without specifying any flags or a filename, a list of flags available for the command is displayed.

The following table presents the options available with Embedded SQL/Ada.

Esqla Command Examples

Command	Comment
esqla file1	Preprocesses "file1.sa" to "file1.ada"
esqla file2.xa	Preprocesses "file2.xa" to "file2.ada"
esqla -l file3	Preprocesses "file3.sa" to "file3.ada" and creates the listing "file3.lis"
esqla -s	Accepts input from standard input
esqla -ffile4.out file4	Preprocesses "file4.sa" to "file4.out"
esqla	Displays a list of flags available for this command.

The ACS Environment and the Ada Compiler

The preprocessor generates Ada code. You can then use the VMS **ada** command to compile this code into your Ada program library.

The following sections describe the Ada program library and Embedded SQL programs.

Entering Embedded SQL Package Specifications

Once you have set up an Ada program library, you must add four Embedded SQL units to your library. The units are package specifications that describe to the Ada compiler all the calls that the preprocessor generates. The source for these units is in the files:

```
ii_system:[ingres.files]eqdef.ada
ii_system:[ingres.files]eqsqlca.ada
ii_system:[ingres.files]eqsqlda.ada
```

Once you have defined your current program library using the **acs set library** command, you should enter the three units into your program library by issuing the following commands:

```
copy ii_system:[ingres.files]eqdef.ada, eqsqlca.ada,-
  eqsqlda.ada []
ada eqdef.ada,eqsqlca.ada,eqsqlda.ada
delete eqdef.ada.,eqsqlca.ada.,eqsqlda.ada
```

You do not have to take the last step if you intend to compile the *closure* of a particular program from the source files at a later date. However, you should not modify an Embedded SQL definition file if it is left in your directory.

You need only enter the four Embedded SQL units once into your program library. Of course, if a new release of Embedded SQL/Ada includes modifications to the files "eqdef.ada," "esqlca.ada," or "eqsqlda.ada," you should copy and recompile the files.

If you display the new unit information, you will find the four unit names "ESQL," "ESQLDA," "EQUEL," and "EQUEL_FORMS" in the library. For example, by issuing:

```
acs dir esql*,equel*
```

the three unit names will be displayed.

Defining Long Floating-point Storage

The storage representation format of long floating-point variables must be **d_float** because the Ingres runtime system uses that format for floating-point conversions. If your Embedded SQL program has **long_float** variables that interact with the Embedded SQL runtime system, you must make sure they are stored in the **d_float** format. The default Ada format is **g_float**. A convenient way to control the format of all long float variables is to issue the **acs set pragma** program command. For example, by issuing the following command, you redefine the program library characteristics for **long_float** from the default to **d_float**:

```
acs set pragma/long_float=d_float
```

A second remedy to this particular problem is to issue the statement:

```
pragma long_float(d_float)
```

in the source file of each compilation unit that uses floating-point variables. You can also explicitly declare the Embedded SQL variables with type **d_float**, as defined in package SYSTEM.

The following example is a typical command file that sets up a new Ada program library with the Embedded SQL package specifications and the **d_float** numerical format. The name of the new program library is passed in as a parameter:

```
acs create library [.'p1']
acs set library [.'p1']
acs set pragma/long_float=d_float
copy ii_system:[ingres.files]eqdef.ada,eqsqlca.ada, -
  eqsqlda.ada []
ada eqdef.ada,eqsqlca.ada,eqsqlda.ada
delete eqdef.ada.,eqsqlca.ada.,eqsqlda.ada
exit
```

The Ada Compiler

After you enter the Embedded SQL packages into the Ada program library, you can compile the Ada file generated by the preprocessor.

The following example preprocesses and compiles the file "test1." Note that both the Embedded SQL/Ada preprocessor and the Ada compiler assume the default extensions.

```
esqla test1
ada/list test1
```

Note: Refer to the Readme file for any operating system specific information on compiling and linking ESQL/Ada programs.

VMS

As of Ingres II 2.0/0011 (axm.vms/00) Ingres uses member alignment and IEEE floating-point formats. Embedded programs must be compiled with member alignment turned on. In addition, embedded programs accessing floating-point data (including the MONEY data type) must be compiled to recognize IEEE floating-point formats.

Linking an Embedded SQL Program

Embedded SQL/Ada programs require procedures from several VMS shared libraries in order to run. After you preprocess and compile an Embedded SQL/Ada program, you can link it. For example, if your program unit is called "dbentry," you can use the following link command:

```
acs link dbentry -
 ii_system:[ingres.files]esql.opt/opt
```

Note that the Embedded SQL runtime library is not written in Ada, and therefore is specified as a *foreign* object file.

Assembling and Linking Precompiled Forms

The technique of declaring a precompiled form to the Forms Runtime System is discussed in the *SQL Reference Guide.* To use such a form in your program, you must also follow the steps described here.

In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a file in your directory describing the form in the VAX MACRO language. After you create the file this way, you can assemble it into linkable object code with the VMS command:

macro *filename*

The output of this command is a file with the extension ".obj". You can then link this object file with your program by specifying it in the link command as in the following example for the program unit "formentry," which includes two compiled forms:

```
acs link formentry -
  empform.obj, deptform.obj, -
  ii_system:[ingres.files]esql.opt/opt
```

Linking an Embedded SQL Program without Shared Libraries

While the use of shared libraries in Embedded SQL programs is recommended for optimal performance and ease of maintenance, non-shared versions of the Embedded SQL runtime libraries have been included in case you require them. Non-shared libraries required by Embedded SQL are listed in the esql.noshare options file. The options file must be included in your link command *after* all local modules. Libraries must be specified in the order given in the options file.

The following example demonstrates the link command of the Embedded SQL program unit "dbentry," which has been preprocessed and compiled:

```
acs link dbentry -
 ii_system:[ingres.files]esql.noshare/opt
```

Embedded SQL/Ada Preprocessor Errors

To correct most errors, you may wish to run the Embedded SQL/Ada preprocessor with the listing (-l) option on. The listing should be sufficient for locating the source and reason for the error.

For preprocessor error messages specific to Ada, see Preprocessor Error Messages in this chapter.

Preprocessor Error Messages

The following is a list of error messages specific to Ada.

E_DC000A

"Table 'employee' contains column(s) of unlimited length."

Explanation: Character string(s) of zero length have been generated. This causes a compile-time error. You must modify the output file to specify an appropriate length.

E_E60001 "The Ada variable '%0c' is an array and must be subscripted."

Explanation: A variable declared as an array must be subscripted when referenced. The preprocessor does not confirm that you use the correct number of subscripts. A variable declared as a 1-dimensional array of characters must not be subscripted as it refers to a character string.

E_E60002 "The Ada variable '%0c' is not an array and must not be subscripted."

Explanation: A variable not declared as an array cannot be subscripted. You cannot subscript string variables in order to refer to a single character or a slice of a string (sub-string).

E_E60003 "The Ada identifier '%0c' is not a declared type."

Explanation: The identifier was used as an Ada type name in an object or type declaration. This identifier has not yet been declared to the preprocessor and is not a preprocessor-predefined type name.

E_E60004 "The Ada CHARACTER variable '%0c' must be a 1-dimensional array."

Explanation: Variables of type CHARACTER can only be declared as 1-dimensional arrays. You cannot use a single character or a multi-dimensional array of characters as an Ingres string. Note that you can use a multidimensional array of type STRING.

E_E60005 "The Ada DIGITS clause '%0c' is out of the range 1..16."

Explanation: Embedded Ada supports D_FLOAT floating-point variables. Consequently, all DIGITS specifications must be in the specified range.

E_E60006 "Statement '%0c' is embedded in INCLUDE file package specification."

Explanation: Preprocessor INCLUDE files may only be used for Ada package specifications. The preprocessor generates an Ada WITH clause for the package. No executable statements may be included in the file because the code generated will not be accepted by the Ada compiler in a package specification.

E_E60007 "Too many names (%0c) in Ada identifier list. Maximum is %1c."

Explanation: Ada identifier lists cannot have too many names in the comma-separated name list. The name specified in the error message caused the overflow, and the remainder of the list is ignored. Rewrite the declaration so that there are fewer names in the list.

E_E60008 "The Ada identifier list has come up short."

Explanation: The stack used to store comma separated names in Ada declarations has been corrupted. Try rearranging the list of names in the declaration.

E_E60009 "The Ada CONSTANT declaration of '%0c' must be initialized."

Explanation: CONSTANT declarations must include an initialization clause.

E_E6000A "The Ada identifier '%0c' is either a constant or an enumerated literal."

Explanation: The named identifier was used to retrieve data from Ingres. A constant, an enumerated literal and a formal parameter with the IN mode are all considered illegal for the purpose of retrieval.

E_E6000B "The Ada variable '%0c' with '.ALL' clause is illegal."

Explanation: The ADA .ALL clause, as specified with access objects, can be used only if the variable is an access object pointing at a single scalar-valued type. If the type is not scalar valued, or if the access object is pointing at a record or array, then the use of .ALL is illegal.

E_E6000C "The Ada variable '%0c' with '.ALL' clause is not a scalar type."

Explanation: The ADA .ALL clause, as specified with access objects, can be used only if the variable is an access object pointing at a single scalar-valued type. If the type is not scalar valued, or if the access object is pointing at a record or array, then the use of .ALL is illegal.

E_E6000D "Last component in Ada record qualification '%0c' is illegal."

Explanation: The last component referenced in a record qualification is not a member of the record. If this component was supposed to be declared as a record, the following components will cause preprocessor syntax errors.

E_E6000E "In ADA RENAMES statement, '%0c' must be a constant or a variable."

Explanation: The target object of a RENAMES statement must be a constant or a variable, and the item being declared is used a synonym for the target object.

E_E6000F "In ADA RENAMES statement, object is incompatible with type."

Explanation: The type of the target object in the RENAMES statement must be compatible in base type, size and array dimensions with the type name specified in the declaration.

E_E60010	"Only one name may be declared in an ADA RENAMES statement." **Explanation:** One object can rename only one other object.
E_E60012	"The Ada variable '%0c' has not been declared." **Explanation:** The named identifier was used where a variable must be used to set or retrieve Ingres data. The variable has not yet been declared.
E_E60013	"The ADA type %0c is not supported." **Explanation:** Some Ada types are not supported because they are not compatible with the Ingres runtime system.
E_E60014	"The ADA variable '%0c' is a record, not a scalar value." **Explanation:** The named variable qualification refers to a record. It was used where a variable must be used to set or retrieve Ingres data. This error may also cause syntax errors on record component references.
E_E60016	"The ADA statement %0c is not supported." **Explanation:** Statements that modify the internal representation of variables that interact with Ingres are not supported.

Sample Applications

This section contains sample applications.

The Department-Employee Master/Detail Application

This application uses two database tables joined on a specific column. This typical example of a department and its employees demonstrates how to process two tables as a master and a detail.

The program scans through all the departments in a database table, in order to reduce expenses. Based on certain criteria, the program updates department and employee records. The conditions for updating the data are the following:

Departments:

- If a department has made less than $50,000 in sales, the department is dissolved.

Employees:

- If an employee was hired since the start of 1985, the employee is terminated.
- If the employee's yearly salary is more than the minimum company wage of $14,000 and the employee is not nearing retirement (over 58 years of age), the employee takes a 5% pay cut.
- If the employee's department is dissolved and the employee is not terminated, the employee is moved into a state of limbo to be resolved by a supervisor.

This program uses two cursors in a master/detail fashion. The first cursor is for the Department table, and the second cursor is for the Employee table. Both tables are described in **declare table** statements at the start of the program. The cursors retrieve all the information in the tables, some of which is updated. The cursor for the Employee table also retrieves an integer date interval whose value is positive if the employee was hired after January 1, 1985.

Each row that is scanned from both the Department table and the Employee table is recorded into the system output file. This file serves as a log of the session and as a simplified report of the updates.

Each section of code is commented for the purpose of the application and also to clarify some of the uses of the Embedded SQL statements. The program illustrates table creation, multi-statement transactions, all cursor statements, direct updates, and error handling.

```
-- Create package for Long_Float I/O so as not to conflict with
-- the default G_FLOAT format. This example assumes that the ACS
-- SET PRAGMA command has been issued.

with text_io;
package long_float_text_io is new text_io.float_io(long_float);

-- I/O utilities
with text_io;                   use text_io;
with integer_text_io;           use integer_text_io;
with short_integer_text_io;     use short_integer_text_io;
with short_short_integer_text_io; use short_short_integer_text_io;
with float_text_io;             use float_text_io;
with long_float_text_io;        use long_float_text_io;

exec sql include sqlca;

-- The department table
exec sql declare dept table
      (name        char(12) not null,      -- Department name
       totsales    money not null,         -- Total sales
       employees   smallint not null);      -- Number of employees

-- The employee table
exec sql declare employee table
      (name        char(20)     not null,    -- Employee name
       age         integer1 not null,        -- Employee age
       idno        integer not null,         -- Unique employee id
       hired       date not null,            -- Date of hire
```

```
       dept         char(12) not null,        -- Department of work
       salary       money not null);          -- Yearly salary

-- "State-of-Limbo" for employees who lose their department
exec sql declare toberesolved table
      (name         char(20) not null,        -- Employee name
       age          integer1 not null,        -- Employee age
       idno         integer not null,         -- Unique employee id
       hired        date   not null,          -- Date of hire
       dept         char(12) not null,        -- Department of work
       salary       money not null);          -- Yearly salary

-- Procedure: Process_Expenses -- MAIN
-- Purpose:   Main body of the application. Initialize the
--            database, process each department and terminate
--            the session.
-- Parameters:
--            None

procedure Process_Expenses is

      log_file: File_type;            -- Log file to write to.
      sql_error: exception;
--
-- Procedure:    Init_Db
-- Purpose:      Initialize the database.
--               Connect to the database and abort on error.
--               Before processing departments and employees,
--               create the table for employees who
--               lose their departments, "toberesolved".
-- Parameters:
--             None
--

procedure Init_Db is

begin
    exec sql whenever sqlerror stop;
    exec sql connect personnel;

    put_line(log_file,
      "Creating ""To_Be_Resolved"" table.");
    exec sql create table toberesolved
         (name      char(20) not null,
          age       integer1 not null,
          idno      integer not null,
          hired     date not null,
          dept      char(12) not null,
          salary    money not null);

end Init_Db;

-- Procedure: End_Db
-- Purpose:   Commit the multi-statement transaction and
--            end access to the database.
-- Parameters:
--            None

procedure End_Db is
begin
    exec sql commit;
    exec sql disconnect;
end End_Db;

--
-- Procedure: Close_Down
```

```
-- Purpose: Error handler called any time after Init_Db has been
--          successfully completed. In all cases, print the cause
--          of the error and abort the transaction, backing out
--          changes. Note that disconnecting from the database
--          will implicitly close any open cursors.
-- Parameters: None.
--

procedure Close_Down is

      exec sql begin declare section;
             errbuf: String(1..200);
      exec sql end declare section;

begin
      -- Turn off error handling here
      exec sql whenever sqlerror continue;
      exec sql inquire_sql (:errbuf = ERRORTEXT);
      put_line( "Closing Down because of database error.");
      put_line( errbuf );

      exec sql rollback;
      exec sql disconnect;

      raise sql_error; -- No return
end Close_Down;

--
-- Procedure: Process_Employees
-- Purpose:   Scan through all the employees for a particular
--            department.Based on given conditions, the employee
--            may be terminated or given a salary reduction:
--            1. If an employee was hired since 1985, the
--            employee is terminated.
--            2. If the employee's yearly salary is more
--            than the minimum company wage of $14,000 and
--            the employee is not close to retirement
--            (over 58 years of age), the employee takes
--            a 5% salary reduction.
--            3. If the employee's department is dissolved
--            and the employee is not terminated, then
--            the employee is moved into the
--            "toberesolved" table.
-- Parameters:
--            dept_name    - Name of current department.
--            deleted_dept - Is department dissolved?
--            emps_term - Set locally to record how many employees
--                     were terminated for the current department.
procedure Process_Employees
        (dept_name:      in String;
         deleted_dept:   in Boolean;
         emps_term:      in out Integer) is

    exec sql begin declare section;
      -- Emp_Rec corresponds to the "employee" table
      type Emp_Rec is
          record
              name:           String(1..20);
              age:            Short_Short_Integer;
              idno:           Integer;
              hired:          String(1..25);
              salary:         Float;
              hired_since_85: Integer;
          end record;
      erec: Emp_Rec;
      salary_reduc: constant Float := 0.95;
```

```
      dname: String(1..12) := dept_name;
    exec sql end declare section;

    min_emp_salary: constant Float := 14000.00;
    nearly_retired: constant Short_Short_Integer := 58;
    title:       String(1..12); -- Formatting values
    descript:    String(1..25);

    -- Note the use of the Ingres function to find out
    -- who has been hired since the start of 1985.

    exec sql declare empcsr cursor for
          select name, age, idno, hired, salary,
          int4(interval('days', hired-date('01-jan-1985')))
          from employee
          where dept = :dname
          for direct update OF name, salary;

begin                    -- Process Employees

    -- All errors from this point on close down the application
    exec sql whenever sqlerror call Close_Down;
    exec sql whenever not found goto Close_Emp_Csr;

    exec sql open empcsr;

    emps_term := 0;             -- Record how many
    while (sqlca.sqlcode = 0) loop
           exec sql fetch empcsr into :erec;

           if (erec.hired_since_85 > 0) then
                exec sql delete from employee
                        where current of empcsr;
                title := "Terminated: ";
                descript := "Reason: Hired since 1985.";
                emps_term := emps_term + 1;

           -- Reduce salary if not nearly retired
           elsif (erec.salary > min_emp_salary) then
                if (erec.age < nearly_retired) then
                    exec sql update employee
                        set salary = salary * :salary_reduc
                        where current of empcsr;
                    title := "Reduction: ";
                    descript := "Reason: Salary. ";
                else
                    -- Do not reduce salary
                    title := "No Changes: ";
                    descript :=
                    "Reason: Retiring. ";
                end if;

           -- Else leave employee alone
           else
                  title := "No Changes: ";
                  descript := "Reason: Salary. ";
           end if;

           -- Was employee's department dissolved?
           if (deleted_dept) then
                exec sql insert into toberesolved
                      select *
                      from employee
                      where idno = :erec.idno;
                exec sql delete from employee
                      where current of empcsr;
```

```
          end if;

          -- Log the employee's information
          put(log_file, " " & title & " ");
          put(log_file, erec.idno, 6);
          put(log_file, ", " & erec.name & ", ");
          put(log_file, erec.age, 3);
          put(log_file, ", ");
          put(log_file, erec.salary, 8, 2, 0);
          put_line(log_file, " ; " & descript);
    end loop;

<<Close_Emp_Csr>>
    exec sql whenever not found continue;
    exec sql close empcsr;

end Process_Employees;

-- Procedure:Process_Depts
-- Purpose: Scan through all the departments, processing each
--          one. If the department has made less than $50,000 in
--          sales, dissolve the department. For each department,
--          process all employees (they may even be moved to
--          another database table). If an employee wa
--          terminated, update the department's employee counter.
-- Parameters:
--          None

procedure Process_Depts is

    exec sql begin declare section;
        -- Dept_Rec corresponds to the "dept" table
        type Dept_Rec is
              record
                    name:       String(1..12);
                    totsales:   Long_Float;
                    employees:  Short_Integer;
              end record;
        dept: Dept_Rec;

        -- Employees terminated
        emps_term: Integer := 0;
    exec sql end declare section;

    min_tot_sales: constant := 50000.00;
    deleted_dept: Boolean; -- Was the dept deleted?
    dept_format: String(1..20); -- Formatting value

    exec sql declare deptcsr cursor for
          select name, totsales, employees
          from dept
          for direct update of name, employees;

begin
    -- All errors from this point on close down the application
    exec sql whenever sqlerror call Close_Down;
    exec sql whenever not found goto Close_Dept_Csr;

    exec sql open deptcsr;

    while (sqlca.sqlcode = 0) loop
        exec sql fetch deptcsr into :dept;

        -- Did the department reach minimum sales?
        if (dept.totsales < min_tot_sales) then
            exec sql delete from dept
```

```
                where current of deptcsr;
            deleted_dept := TRUE;
            dept_format := " -- DISSOLVED --";
        else
            deleted_dept := FALSE;
            dept_format := (1..20 = > ' ');
        end if;

        -- Log what we have just done
        put(log_file,
            "Department: " & dept.name &
            ", Total Sales: ");
        put(log_file, dept.totsales, 12, 3, 0);
        put_line(log_file, dept_format);
        -- Now process each employee in the department
        Process_Employees(dept.name,
           deleted_dept, emps_term);

        -- If employees were terminated, record the fact
        if (emps_term > 0 and not deleted_dept) then
            exec sql update dept
                set employees = :dept.employee - :emps_term
                where current of deptcsr;
        end if;
    end loop;

<<Close_Dept_Csr>>
    exec sql whenever not found continue;
    exec sql close deptcsr;
end Process_Depts;

begin -- MAIN program
      put_line("Entering application to process expenses.");
      create(log_file, out_file, "expenses.log");
      Init_Db;
      Process_Depts;
      End_Db;
      close(log_file);
      put_line("Completion of application.");

      exception
           when sql_error =>
               null; -- Just go away quietly
end Process_Expenses;
```

The Table Editor Table Field Application

This application edits the Person table in the Personnel database. It is a forms application that allows the user to update a person's values, remove the person, or add new persons. Various table field utilities are provided with the application to demonstrate how they work.

The objects used in this application are:

Object	Description
personnel	The program's database environment.
person	A table in the database, with three columns:

Object	Description
	name (**char(20)**) age (**smallint**) number (**integer**) Number is unique.
personfrm	The VIFRED form with a single table field.
persontbl	A table field in the form, with two columns: name (**char(20)**) age (**integer**) When initialized, the table field includes the hidden column number (**integer**).

At the beginning of the application, the program opens a database cursor to load the table field with data from the Person table. After loading the table field, the user can browse and edit the displayed values. The user can add, update, or delete entries. When finished, the values are unloaded from the table field and, in a multi-statement transaction, the updates are transferred back into the Person table.

```
-- I/O utilities
with text_io; use text_io;

exec sql include sqlca;

exec sql declare person table
   (name     char(20),         -- Person name
    age      smallint,         -- Age
    number   integer);         -- Unique id number
procedure Table_Edit is

      exec sql begin declare section;
          -- Table field row states
          type Row_States is (
              row_undef,       -- Empty or undefined row
              row_new,         -- Appended by user
              row_unchange,   -- Loaded by program, but not updated
              row_change,      -- Loaded by program and updated
              row_delete       -- Deleted by program
          );
          not_found: constant := 100; -- SQLCA value for no rows
          -- Person information corresponds to "person" table
          pname:   String(1..20);  -- Full name
          page:    Short_Integer;  -- Age
          pnumber: Integer;        -- Unique person number
          pmaxid:  Integer;        -- Maximum person id number
          -- Table field entry information
          state: Row_States;       -- State of data set row
          recnum,                  -- Record number
          lastrow: Integer;        -- Last row in table field
          -- Utility buffers
          search:  String(1..20);  -- Name to find in search loop
          password: String(1..13); -- Password buffer
          msgbuf:  String(1..100); -- Message buffer
          respbuf: String(1..1);   -- Response buffer
```

```
exec sql end declare section;

-- Error handling variables for database updates
update_error: Boolean; -- Error in updates?
update_commit: Boolean; -- Commit updates

-- Load the information from the "person" table into the
-- person variables. Also, save the maximum person id
-- number.

function Load_Table return Integer is
     exec sql begin declare section;
            -- Person information
            pname:   String(1..20); -- Full name
            page:    Short_Integer; -- Age
            pnumber: Integer;        -- Unique person number
            maxid:   Integer;    -- Maximum person id number
      exec sql end declare section;
      exec sql declare loadtab cursor for
            select name, age, number
            from person;

      -- Set up error handling for loading procedure
      exec sql whenever sqlerror goto Load_End;
      exec sql whenever not found goto Load_End;

begin                      -- Load_Table
      exec frs message 'Loading Person Information . . .';

      -- Fetch the maximum person id number for later use
      exec sql select max(number)
            into :maxid
            from person;

      exec sql open loadtab;

      while (sqlca.sqlcode = 0) loop
          -- Fetch data into record and load table field
          exec sql fetch loadtab into
                          :pname, :page, :pnumber;

          exec frs loadtable personfrm persontbl
                (name = :pname, age = :page,
                             number = :pnumber);
      end loop;

<<Load_End>>
    exec sql whenever sqlerror continue;
    exec sql close loadtab;

    return maxid;
end Load_Table;

begin -- Table_Edit
      -- Set up error handling for main program
      exec sql whenever sqlwarning continue;
      exec sql whenever not found continue;
      exec sql whenever sqlerror STOP;

      -- Start up Ingres and the FORMS system

      exec sql connect 'personnel';

      exec frs forms;

      -- Verify that the user can edit the "person" table
```

```
exec frs prompt noecho ('Password for table editor: ',
                         :password);
if (password /= "MASTER_OF_ALL") then
    exec frs message 'No permission for task.
                                  Exiting . . .';
    exec frs endforms;
    exec sql disconnect;
    return;
end if;

exec frs message 'Initializing Person Form . . .';
exec frs forminit personfrm;

-- Initialize "persontbl" table field with a data set
-- in FILL mode, so that the runtime user can append rows.
-- To keep track of events occurring to original
-- rows loaded into the table field, hide the unique
-- person number.

exec frs inittable personfrm persontbl FILL
                  (number = integer);

pmaxid := Load_Table;

-- Display the form and allow runtime editing

exec frs display personfrm update;
exec frs initialize;
exec frs begin;
    -- Provide menu items, as well as the system FRS
    -- key, to scroll to both extremes of the table field.
    exec frs scroll personfrm persontbl to 1;
exec frs end;

exec frs activate menuitem 'Top';
exec frs begin;
    exec frs scroll personfrm persontbl TO 1; -- Backward
exec frs end;

exec frs activate menuitem 'Bottom';
    exec frs begin;
    exec frs scroll personfrm persontbl to end; -- Forward
exec frs end;

exec frs activate menuitem 'Remove';
exec frs begin;
    -- Remove the person in the row the user's cursor
    -- is on. If there are no persons, exit operation
    -- with message. Note that this check cannot
    -- really happen, as there is always at least one
    -- UNDEFINED row in FILL mode.

    exec frs inquire_frs table personfrm
        (:lastrow = lastrow(persontbl));
    if (lastrow = 0) then
        exec frs message 'Nobody to Remove';
        exec frs sleep 2;
        exec frs resume field persontbl;
    end if;

exec frs deleterow personfrm persontbl;  -- Recorded for
                                         -- later
exec frs end;

exec frs activate menuitem 'Find';
exec frs begin;
```

```
        -- Scroll user to the requested table field entry.
        -- Prompt the user for a name, and if one is typed
        -- in, loop through the data set searching for it.

        search := (1..20 => ' ');
            exec frs prompt ('Person''s name : ', :search);
        if (search(1) = ' ') then
            exec frs resume field persontbl;
        end if;

            exec frs unloadtable personfrm persontbl
                 (:pname = name, :recnum = _record,
                  :state = _state);
            exec frs begin;
                -- Do not compare with deleted rows
               if (state /= ROW_DELETE and pname = search) then
                 exec frs scroll personfrm persontbl TO :recnum;
                    exec frs resume field persontbl;
                end if;
         exec frs end;

        -- Fell out of loop without finding name. Issue error.
        msgbuf := (1..100 => ' ');
        msgbuf(1..62) := "Person '" & search &
              "' not found in table [HIT RETURN] ";
        exec frs prompt noecho (:msgbuf, :respbuf);
    exec frs end;

    exec frs activate menuitem 'Exit';
    exec frs begin;
        exec frs validate field persontbl;
        exec frs breakdisplay;
    exec frs end;
    exec frs finalize;

-- Exit person table editor and unload the table field.
-- If any updates, deletions or additions were made,
-- duplicate these changes in the source table. If the
-- user added new people, assign a unique id to each person
-- before adding the person to the table. To do this,
-- increment the previously-saved maximum id number with
-- each insert.
-- Do all the updates in a multi-statement transaction
exec sql savepoint savept;

update_commit := TRUE;

-- Hard code the error handling in the UNLOADTABLE loop,
-- so as to cleanly exit the loop.

exec sql whenever sqlerror continue;

exec frs message 'Exiting Person Application . . .';

exec frs unloadtable personfrm persontbl
    (:pname = name, :page = age,
     :pnumber = number, :state = _state);
exec frs begin;

    case (state) is
        when row_new =>
            -- Filled by user. Insert with new unique id.
            pmaxid := pmaxid + 1;
            exec sql insert into person (name, age, number)
               values (:pname, :page, :pmaxid);
```

```
            when row_change =>
                -- Updated by user. Reflect in table.
                exec sql update person set
                    name = :pname, age = :page
                    where number = :pnumber;

            when row_delete =>
                -- Deleted by user, so delete from table.
                -- Note that only original rows, not rows
                -- appended at runtime, are saved by the
                -- program.
                exec sql delete from person
                    where number = :pnumber;

            when others =>
                -- Else UNDEFINED or UNCHANGED -
                -- No updates required.
                null;
        end case;

        -- Handle error conditions -
        -- If an error occurred, abort the transaction.
        -- If no rows were updated, inform user and
        -- prompt for continuation.
        if (sqlca.sqlcode < 0) then -- Error
            exec sql inquire_sql (:msgbuf = errortext);
            exec sql rollback to savept;
            update_error := TRUE;
            update_commit := FALSE;
            exec frs endloop;
        elsif (sqlca.sqlcode = NOT_FOUND) then
            msgbuf := (1..100 => ' ');
            msgbuf(1..62) :=
                "Person '" & pname &
                "' not updated. Abort all updates? ";
            exec frs prompt noecho (:msgbuf, :respbuf);
            if (respbuf = "Y" or respbuf = "y") then
                update_commit := FALSE;
                exec sql rollback to savept;
                exec frs endloop;
            end if;
        end if;
    exec frs end;
    if (update_commit) then
        exec sql commit;              -- Commit the updates
    end if;

    exec frs endforms; -- Terminate FORMS and Ingres
    exec sql disconnect;

    if (update_error) then
       put_line( "Your updates were aborted because of error:" );
        put_line( msgbuf );
    end if;

end Table_Edit;
```

The Professor-Student Mixed Form Application

This application lets the user browse and update information about graduate students who report to a specific professor. The program is structured in a master/detail fashion, with the professor being the master entry, and the students the detail entries. The application uses two forms—one to contain general professor information and another for detailed student information.

The objects used in this application are shown in the following table:

Object	Description
personnel	The program's database environment.
professor	A database table with two columns: pname (**char(25)**) pdept (**char(10)**). See its **declare table** statement in the program for a full description.
student	A database table with seven columns: sname (**char(25)**) sage (**integer1**) sgpa (**char(25)**) sgpa (**float4**) sidno (**integer**) scomment (**varchar(200)**) sadvisor (**char(25)**) See its **declare table** statement for a full description. The sadvisor column is the join field with the pname column in the Professor table.
masterfrm	The main form has the pname and pdept fields.
studenttbl	A table field in "masterfrm" with two columns "sname" and "sage." When initialized, it also has five hidden columns corresponding to information in the student table.
studentfrm	The detail form, with seven fields, which correspond to information in the student table. Only the sgpa, scomment, and sadvisor fields are updatable. All other fields are display-only.

Object	Description
grad	A global structure, whose members correspond in name and type to the columns of the Student database table, the studentfrm form, and the studenttbl table field.

The program uses the "masterfrm" as the general-level master entry, in which data can only be retrieved and browsed, and the studentfrm as the detailed screen, in which specific student information can be updated. The user can enter a name in the pname field and then select the **Students** menu operation. The operation fills the Studenttbl table field with detailed information of the students reporting to the named professor. This is done by the studentcsr database cursor in the **Load_Students** procedure. The program assumes that each professor is associated with exactly one department.

The user can then browse the table field (in **read** mode), which displays only the names and ages of the students. More information about a specific student can be requested by selecting the **Zoom** menu operation. This operation displays the studentfrm form (in **update** mode). The fields of studentfrm are filled with values stored in the hidden columns of "studenttbl." The user can make changes to three fields (sgpa, scomment, and sadvisor). If validated, these changes are written back to the Database table (based on the unique student id), and to the table field's data set. The user can repeat this process for different professor names.

```
    -- Master and student compiled forms (imported objects)
    package Compiled_Forms is
          exec sql begin declare section;
               masterfrm, studentfrm: Integer;
          exec sql end declare section;
          pragma import_object( masterfrm );
          pragma import_object( studentfrm );
    end Compiled_Forms;

exec sql include sqlca;

exec sql declare student table   -- Graduate student table
      (sname char(25),            -- Name
       sage integer1,             -- Age
       sbdate char(25),           -- Birth date
       sgpa float4,               -- Grade point average
       idno integer,              -- Unique student number
       scomment varchar(200),     -- General comments
       sadvisor char(25));        -- Advisor's name
exec sql declare professor table  -- Professor table
      (pname char(25),            -- Professor's name
       pdept char(10));           -- Department
        with Compiled_Forms;      use Compiled_Forms;
        with Text_Io;             use Text_Io;
        with Integer_Text_Io;     use Integer_Text_Io;
-- Procedure: Prof_Student
-- Purpose:   Main body of "'Professor Student" Master-Detail
--            application.

procedure Prof_Student is

    exec sql begin declare section;
```

```
  -- Graduate student record maps to "student" database table
    type Student_Rec is
      record
          sname:      String(1..25);
          sage:       Short_Short_Integer;
          sbdate:     String(1..25);
          sgpa:       Float;
          sidno:      Integer;
          scomment:   String(1..200);
          sadvisor:   String(1..25);
      end record;
    grad: Student_Rec;
exec sql end declare section;

--
-- Procedure: Load_Students
-- Purpose: Given an advisor name, load into the "studenttbl"
--          table field all the graduate students who report
--          to the professor with that name.
-- Parameters: advisor - User-specified professor name.
--         Uses the global student record "grad."
--

procedure Load_Students( adv : in String ) is

    exec sql begin declare section;
         advisor : String(1..25) := adv;
    exec sql end declare section;

    exec sql declare studentcsr cursor for
      select sname, sage, sbdate, sgpa,
            sidno, scomment, sadvisor
      from student
      where sadvisor = :advisor;

begin
    --
    -- Clear previous contents of table field. Load the table
    -- field from the database table based on the advisor
    -- name. Columns "sname" and "sage" will be displayed,
    -- and all others will be hidden.
    --

    exec frs message 'Retrieving Student Information . . .';
    exec frs clear field studenttbl;
    exec frs redisplay; -- Refresh for query

    exec sql whenever sqlerror goto Load_End;
    exec sql whenever not found goto Load_End;

    exec sql open studentcsr;

    --
    -- Before we start the loop, we know that the OPEN
    -- was successful and that NOT FOUND was not set.
    --

    while (sqlca.sqlcode = 0) loop
        exec sql fetch studentcsr into :grad;

        exec frs loadtable masterfrm studenttbl
           (sname = :grad.sname,
            sage = :grad.sage,
            sbdate = :grad.sbdate,
            sgpa = :grad.sgpa,
            sidno = :grad.sidno,
```

```
                scomment = :grad.scomment,
                sadvisor = :grad.sadvisor);
        end loop;

<<Load_End>>         -- Clean up on an error, and close cursors
        exec sql whenever not found continue;
        exec sql whenever sqlerror continue;
        exec sql close studentcsr;

    end Load_Students;


    --
    -- Function: Student_Info_Changed
    -- Purpose: Allow the user to zoom in on the details of a
    --        selected student. Some of the data can be
    --        updated by the user. If any updates are made,
    --        incorporate them into the database table.
    --        The procedure returns TRUE if any changes are
    --        made.
    -- Parameters:
    --        None
    -- Returns:
    --       TRUE/FALSE - Changes were made to the database.
    --       Sets the global "grad" record with the new data.
    --

    function Student_Info_Changed return Boolean is

        exec sql begin declare section;
            changed: Integer;         -- Changes made to the form?
            valid_advisor: Integer;  -- Is the advisor name valid?
        exec sql end declare section;

    begin

        -- Local error handler just prints error and continues
        exec sql whenever sqlerror call sqlprint;
        exec sql whenever not found continue;

        -- Display the detailed student information
        exec frs display studentfrm fill;
        exec frs initialize
           (sname = :grad.sname,
            sage = :grad.sage,
            sbdate = :grad.sbdate,
            sgpa = :grad.sgpa,
            sidno = :grad.sidno,
            scomment = :grad.scomment,
            sadvisor = :grad.sadvisor);

        exec frs activate menuitem 'Write';
        exec frs begin;

            --
            -- If changes were made, then update the
            -- database table. Only bother with the
            -- fields that are not read-only.
            --
            exec frs inquire_frs form (:changed = change);

            if (changed = 1) then
             exec frs validate;
             exec frs message 'Writing changes to database. . .';

                exec frs getform
```

```
                (:grad.sgpa = sgpa,
                 :grad.scomment = scomment,
                 :grad.sadvisor = sadvisor);

            -- Enforce integrity of professor name
            valid_advisor := 0;
            exec sql select 1 into :valid_advisor
                from professor
                where pname = :grad.sadvisor;

             if (valid_advisor = 0) then
                exec frs message 'Not a valid advisor name';
                exec frs sleep 2;
                exec frs resume field sadvisor;
             else
                exec sql update student set
                    sgpa = :grad.sgpa,
                    scomment = :grad.scomment,
                    sadvisor = :grad.sadvisor
                    where sidno = :grad.sidno;
             end if;
        end if;
        exec frs breakdisplay;
    exec frs end; -- 'Write'

    exec frs activate menuitem 'Quit';
    exec frs begin;
        -- Quit without submitting changes
        changed := 0;
        exec frs breakdisplay;
    exec frs end; -- 'Quit'

    exec frs finalize;

    return (changed = 1);

end Student_Info_Changed;

--
-- Procedure: Master
-- Purpose:   Drive the application, by running "masterfrm"
--         and allowing the user to "zoom" in on a selected
-- student. Parameters: None - Uses the global student "grad"
-- record.

procedure Master is

    exec sql begin declare section;
        -- Professor record maps to "professor" database table
        type Prof_Rec is
              record
                  pname: String(1..25);
                  pdept: String(1..10);
              end record;
        prof: Prof_Rec;

        -- Useful forms runtime information
        lastrow,           -- Lastrow in table field
        istable: Integer;  -- Is a table field?

        -- Utility buffers
        msgbuf:      String(1..100);    -- Message buffer
        respbuf:     String(1..1);      -- Response buffer
        old_advisor: String(1..25); -- Old advisor before ZOOM
  exec sql end declare section;
```

```
begin -- Master
    --
    -- Initialize "studenttbl" with a data set in READ mode.
    -- Declare hidden columns for all the extra fields that
    -- the program will display when more information is
    -- requested about a student. Columns "sname" and "sage"
    -- are displayed. All other columns are hidden, to be
    -- used in the student information form.
    --

    exec frs inittable masterfrm studenttbl read
          (sbdate = char(25),
           sgpa = float4,
           sidno = integer,
           scomment = char(200),
           sadvisor = char(20));

    -- Drive the application, by running "masterfrm" and
    -- allowing the user to "zoom" in on a selected student.
    exec frs display masterfrm update;

    exec frs initialize;
    exec frs begin;
          exec frs message 'Enter an Advisor name . . .';
          exec frs sleep 2;
    exec frs end;

    exec frs activate menuitem 'Students', field 'pname';
    exec frs begin;
          -- Load the students of the specified professor
          exec frs getform (:prof.pname = pname);

          -- If no professor name is given, then resume
          if (prof.pname(1) = ' ') then
               exec frs resume field pname;
          end if;

          --
          -- Verify that the professor exists. If not,
          -- print a message and continue. Assume that
          -- each professor has exactly one department.
          --

          exec sql whenever sqlerror call sqlprint;
          exec sql whenever not found continue;

          prof.pdept := (1..10 => ' ');
          exec sql select pdept
              into :prof.pdept
              from professor
              where pname = :prof.pname;

          -- If no professor, report error
          if (prof.pdept(1) = ' ') then
              msgbuf := (1..100 => ' ');
              msgbuf(1..59) :=
                 "No professor with name '" &
                  prof.pname & "' [RETURN]";
              exec frs prompt noecho (:msgbuf, :respbuf);
              exec frs clear field all;
              exec frs resume field pname;
          end if;

          -- Fill the department field and load students
          exec frs putform (pdept = :prof.pdept);
          Load_Students( prof.pname );
```

```
    exec frs resume field studenttbl;
exec frs end;          -- 'Students'

exec frs activate menuitem 'Zoom';
exec frs begin;
   --
   -- Confirm that user is in "studenttbl" and that
   -- the table field is not empty. Collect data from
   -- the row and zoom for browsing and updating.
   --

   exec frs inquire_frs field masterfrm
       (:istable = table);
   if (istable = 0) then
         exec frs prompt noecho
       ('Select from the student table [RETURN]', :respbuf);
         exec frs resume field studenttbl;
   end if;

   exec frs inquire_frs table masterfrm
         (:lastrow = lastrow);
   if (lastrow = 0) then
        exec frs prompt noecho
            ('There are no students [RETURN]', :respbuf);
        exec frs resume field pname;
   end if;

    -- Collect all data on student into graduate record
    exec frs getrow masterfrm studenttbl
       (:grad.sname = sname,
        :grad.sage = sage,
        :grad.sbdate = sbdate,
        :grad.sgpa = sgpa,
        :grad.sidno = sidno,
        :grad.scomment = scomment,
        :grad.sadvisor = sadvisor);

    --
    -- Display "studentfrm," and, if any changes were made,
    -- make the updates to the local table field row.
    -- Only make updates to the columns corresponding to
    -- writable fields in "studentfrm." If the student
    -- changed advisors, then delete the row from the
    -- display.
    --

    old_advisor := grad.sadvisor;
    if (Student_Info_Changed) then
        if (old_advisor <= grad.sadvisor) then
             exec frs deleterow masterfrm studenttbl;
        else
            exec frs putrow masterfrm studenttbl
                (sgpa = :grad.sgpa,
                 scomment = :grad.scomment,
                 sadvisor = :grad.sadvisor);
        end if;
    end if;

exec frs end;                           -- 'Zoom' ;

exec frs activate menuitem 'Exit';
exec frs begin;
     exec frs breakdisplay;
exec frs end;                          -- 'Exit' ;
```

```
      exec frs finalize;

   end Master;

begin                              -- Prof_Student

    -- Start up Ingres and the FORMS system
    exec frs forms;

    exec sql whenever sqlerror stop;
    exec frs message 'Initializing Student Administrator . . .';
    exec sql connect personnel;

    exec frs addform :masterfrm;
    exec frs addform :studentfrm;

    Master;

    exec frs clear screen;
    exec frs endforms;
    exec sql disconnect;

end Prof_Student;
```

The SQL Terminal Monitor Application

This application executes SQL statements that are read in from the terminal. The application reads statements from input and writes results to output. Dynamic SQL is used to process and execute the statements.

When the application starts, it prompts the user for the database name. The program then prompts for an SQL statement. The preprocessor does not accept SQL comments and statement delimiters. The SQL statement is processed using dynamic SQL, and results and SQL errors are written to output. At the end of the results, the program displays an indicator of the number of rows affected. The loop is then continued and the program prompts you for another SQL statement. When end-of-file is typed in, the application rolls back any pending updates and disconnects from the database.

The user's SQL statement is prepared using **prepare** and **describe**. If the SQL statement is not a **select** statement, then it is run using **execute** and the number of rows affected is printed. If the SQL statement is a **select** statement, a dynamic SQL cursor is opened, and all the rows are fetched and printed. The routines that print the results do not try to tabulate the results. A row of column names is printed, followed by each row of the results.

Keyboard interrupts are not handled. Fatal errors, such as allocation errors, and boundary condition violations are handled by rolling back pending updates and disconnecting from the database session.

```
--
-- I/O utilities
-- This example assumes package Long_Float_Text_IO
-- has been instantiated to use the D_FLOAT format.
--
with text_io;                 use text_io;
```

```
with integer_text_io;         use integer_text_io;
with short_integer_text_io;   use short_integer_text_io;
with long_float_text_io;      use long_float_text_io;

-- Declare the SQLCA and the SQLDA records
exec sql include sqlca;
exec sql include sqlda;

-- Dynamic SQL statement and cursor
exec sql declare stmt statement;
exec sql declare csr cursor for stmt;

--
-- Program: SQL_Monitor
-- Purpose: Main entry of SQL Monitor application.
--

procedure SQL_Monitor is

     exec sql begin declare section;
        dbname: String(1..50) := (1..50 => ' '); -- Database name
        dblen:  Natural;
       dbrun:  Boolean := false;               -- connected to db
     exec sql end declare section;

     -- Global SQLDA. Discriminant SQLN is implicitly set
     -- to IISQ_MAX_COLS
     sqlda: IISQLDA(IISQ_MAX_COLS);

     --
     -- Constants and types needed to declare global storage for
     -- SELECT results.
     --

     -- Length of large string pool from which slices will
     -- be allocated
     MAX_STRING: constant := 3000;

     -- Different numeric types for result variables
     type Numerics is
         record
             n_int: Integer;       -- 4-byte integers
             n_flt: Long_Float;    -- 8-byte floating-points
             n_ind: Short_Integer; -- 2-byte null indicators
         end record;

     type Numerics_Array is array(Short_Integer range <>)
                        of Numerics;

     -- Large string pool from which to allocate slices
     type Strings is
             record
                s_len: Integer;                 -- Length used
                s_data: String(1..MAX_STRING);  -- and data area
           end record;

      -- Record of numerics and strings
      type Results is
              record
                    nums: Numerics_Array(1..IISQ_MAX_COLS);
                    str:  Strings;
              end record;

      --
      -- Global result storage area - set up by Print_Header,
      -- filled when executing the FETCH statement, and
```

```
-- displayed by Print_Row.
--
res: Results;

-- Forward defined procedures and functions

-- Main body of monitor
procedure Run_Monitor;

-- Execute dynamic SELECT statements
function Execute_Select return Integer;

-- Print the column headers for a dynamic SELECT
function Print_Header return Boolean;

-- Print a result row for a dynamic SELECT
procedure Print_Row;

-- Print an error message
procedure Print_Error;

-- Read a statement from input
procedure Read_Stmt(stmt_num: in Integer; stmt_buf:
                      in out String);

--
-- Procedure: Run_Monitor
-- Purpose: Run the SQL monitor. Initialize the global
--          SQLDA with the number of SQLVAR elements.
--          Loop while prompting
--          the user for input and processing the SQL
--          statement; if end-of-file is typed then control
--          is returned to the main program exception
--          handler from Read_Stmt.
--
--          If the statement is not a SELECT statement
--          then execute it, otherwise open a cursor and
--          process a dynamic select statement
--          (using Execute_Select).
--

procedure Run_Monitor is

exec sql begin declare section;

  stmt_buf: String(1..1000); -- SQL statement input buffer
  stmt_num: Integer;         -- SQL statement number
  rows: Integer;             -- # of rows affected

exec sql end declare section;

begin                               -- Run_Monitor

      -- Now we are set for input
      stmt_num := 0;

      -- Loop till end-of-file is detected.
      loop

            --
            -- Prompt and read the next statement. If Read_Stmt
            -- end-of-file was detected then end_error is signaled
            -- and control is returned to the main program.
            --
            stmt_num := stmt_num + 1;
            Read_Stmt(stmt_num, stmt_buf);
```

```
            -- Handle database errors
            exec sql whenever sqlerror goto Stmt_Err;

            --
            -- PREPARE and DESCRIBE the statement. If the
            -- statement is not a SELECT then EXECUTE it,
            -- otherwise inspect the
            -- contents of the SQLDA and call Execute_Select.
            --
            exec sql prepare stmt from :stmt_buf;
            exec sql describe stmt into :sqlda;

            --
            -- If SQLD = 0 then this is not a SELECT
            -- statement. Otherwise call Execute_Select to process
            -- a dynamic cursor.
            if (sqlda.sqld = 0) then

                exec sql execute stmt;
                rows := sqlca.sqlerrd(3);

            else

                rows := Execute_Select;

            end if;                     -- If SELECT or not

            exec sql whenever sqlerror continue;

            <<Stmt_Err>>
            --
            -- Only display error message if we arrived here
            -- because of the SQLERROR condition. Otherwise print
            -- the rows processed and continue with the loop.
            if (sqlca.sqlcode < 0) then
                    Print_Error;
            else
                  put("[");
                  put(rows, 1);
                  put_line(" row(s)]");
            end if;

            end loop; -- While reading statements

end Run_Monitor;
--
-- Function: Execute_Select
-- Purpose:  Run a dynamic SELECT statement. The SQLDA has
--           already been described. This routine calls
--           Print_Header to print column headers
--           and set up result storage information.
--           A Dynamic SQL cursor is then opened
--           and each row is fetched and printed by Print_Row.
--           Any error causes the cursor to be closed.
-- Returns:
--           Number of rows fetched from the cursor.
--

function Execute_Select return Integer is

            rows: Integer := 0;             -- Counter of rows fetched

begin                       -- Execute_Select

    --
```

```
    -- Print result column names and set up the result data types
    -- and variables. Print_Header returns FALSE if the dynamic
    -- set-up fails.
    --
    if (Print_Header) then

          exec sql whenever sqlerror goto Select_Error;
          -- Open the dynamic cursor
          exec sql open csr for readonly;

          -- Fetch and print each row
          rows := 0;
          while (sqlca.sqlcode = 0) loop

              exec sql fetch csr using descriptor :sqlda;
              if (sqlca.sqlcode = 0) then
                  rows := rows + 1; -- Count the rows
                  Print_Row;
              end if;

          end loop;                  -- While there are more rows
        <<Select_Error>>
          -- Display error message if SQLERROR condition is set.
          if (sqlca.sqlcode < 0) then
                     Print_Error;
          end if;

                         exec sql whenever sqlerror continue;
                         exec sql close csr;

        end if; -- If Print_Header

        return rows;

end Execute_Select;

--
-- Function: Print_Header
-- Purpose: A statement has just been described so set up the
--          SQLDA for result processing. Print all the column
--          names and allocate result variables for retrieving
--          data. The result variables are allocated out of
--          a pool of numeric variables (integers, floats and
--          2-byte indicators) and a large character buffer.
--          The SQLDATA and SQLIND fields are pointed at the
--          addresses of the result variables.
-- Returns:
--          TRUE if successfully set up the SQLDA for result
--          variables, FALSE if an error occurred.
--

function Print_Header return Boolean is

      nullable: Boolean;     -- Null indicator required
      chlen: Short_Integer;  -- Current string length

begin                        -- Print_Header

    --
    -- Verify that there are enough result variables.
    -- If not print an error and return.
    --
    if (sqlda.sqld >= sqlda.sqln) then
        put("SQL Error: SQLDA requires ");
        put(sqlda.sqld, 1);
        put(" variables, but has only ");
```

```
    put(sqlda.sqln, 1);
    put_line(".);
    return FALSE;
end if; -- If enough result variables

--
-- For each column print the number and title. For example:
--    [1] name [2] age [3] salary
-- While processing each column determine the column type
-- and to where SQLDATA and SQLIND must point in order to
-- retrieve type-compatible results.
--

res.str.s_len := 1;               -- No string space used yet

for col in 1 .. sqlda.sqld loop    -- For each column

    declare

                 sqv: IISQL_VAR renames sqlda.sqlvar(col); -- Shorthand

    begin

        -- Print column name and number
        put("[");
        put(col, 1);
        put("] ");
        put(sqv.sqlname.sqlnamec(1..Integer
                    (sqv.sqlname.sqlnamel)));
        if (col < sqlda.sqld) then
             put(" ");      -- Separator space
        end if;

        --
        -- Process the column for type and length
        -- information. Use
        -- result storage area from which variables can
        -- be allocated.

        if (sqv.sqltype < 0) then
                      -- Null indicator handled later
            nullable := TRUE;
        else
            nullable := FALSE;
        end if;

        case (abs(sqv.sqltype)) is

            - Integers - use 4-byte integer
            when IISQ_INT_TYPE =>
                  sqv.sqltype := IISQ_INT_TYPE;
                  sqv.sqllen := 4;
                  sqv.sqldata := res.nums(col).n_int'Address;
            -- Floating points - use 8-byte float
            when IISQ_MNY_TYPE | IISQ_FLT_TYPE =>

                  sqv.sqltype := IISQ_FLT_TYPE;
                  sqv.sqllen := 8;
                  sqv.sqldata := res.nums(col).n_flt'Address;

            -- Character strings
       when IISQ_DTE_TYPE | IISQ_CHA_TYPE | IISQ_VCH_TYPE =>

              --
              -- Determine the length of the slice required
              -- from the large character buffer. If we have
```

```
                -- enough space left then point at the start of
                -- the corresponding slice, otherwise print an
                -- error and return.
                    --
                    -- Note that for DATE types we must set the
                    -- length.
                    --
                    if (abs(sqv.sqltype) = IISQ_DTE_TYPE) then
                        chlen := IISQ_DTE_LEN;
                    else
                        chlen := sqv.sqllen;
                    end if;

                    -- Enough room in large string buffer ?
                if (res.str.s_len + Integer(chlen) > MAX_STRING)
                        then
                        new_line;
                      put_line("SQL Error: Character result data "
                                 & "overflow.");
                        return FALSE;
                    end if;

                --
                -- Allocate space out of the large character
                -- buffer and keep track of the amount of space
                -- used so far.
                --
                    sqv.sqltype := IISQ_CHA_TYPE;
                    sqv.sqllen  := chlen;
                    sqv.sqldata :=
                         res.str.s_data(res.str.s_len)'Address;
                res.str.s_len := res.str.s_len + Integer(chlen);
              -- Bad data type
              when others =>

                    new_line;
                put("SQL Error: Unknown data type returned: ");
                    put(sqv.sqltype, 1);
                    put_line(".);
                    return FALSE;

          end case;                 -- Of data types

          -- If nullable then point at null indicator and
          -- toggle type id
          if (nullable) then
                sqv.sqltype := -sqv.sqltype;
                sqv.sqlind := res.nums(col).n_ind'Address;
          else
                sqv.sqlind := IISQ_ADR_ZERO;
          end if;

      end;                             -- Declare (rename) block

    end loop;                          -- For processing columns

    new_line;                          -- Print separating line
    put_line("---------------------------------------");

    return TRUE;

end Print_Header;

--
--  Procedure: Print_Row
--  Purpose:  For each element inside the SQLDA, print the value.
```

```
--            Print its column number too in order to identify it
--            with the column name printed earlier. If the value
--            is NULL print "N/A". This routine prints the values
--            using very basic formats and does not try to
--             tabulate the results.
--

procedure Print_Row is

    chlen: Short_Integer; -- Index into string slices

begin                   -- Print_Row

    --
    -- For each column, print the column number and the data. The
    -- number identifies the column with the column name printed
    -- in Print_Header. NULL columns are printed as "N/A".
    --
    res.str.s_len := 1;       -- No characters printed yet

    for col in 1 .. sqlda.sqld loop

        declare

            sqv: IISQL_VAR renames sqlda.sqlvar(col); -- Shorthand

        begin

            put("[");              -- Print column number
            put(col, 1);
            put("] ");

            -- If nullable and is NULL then print "N/A"
            if (sqv.sqltype < 0) and (res.nums(col).n_ind = -1) then
                put("N/A");

            else

                --
                -- Using the base type set up in Print_Header
                -- determine how to print the results. All types
                -- are printed using
                -- very basic formats.
                --

                case (abs(sqv.sqltype)) is
                                         when IISQ_INT_TYPE =>

                                         put(res.nums(col).n_int, 1);

                                         when IISQ_FLT_TYPE =>

                                         put(res.nums(col).n_flt, 1, 4, 0);

                                         when IISQ_CHA_TYPE =>

                        -- Use a current-length slice from the
                        -- large character buffer, as allocated
                        -- in Print_Header.
                        -- Track number of characters printed.
                        chlen := sqv.sqllen;
                        put(res.str.s_data(res.str.s_len ..
                             res.str.s_len + Integer(chlen) - 1));
                        res.str.s_len :=
                               res.str.s_len + Integer(chlen);
```

```
                                        when others => -- Bad data type
                        put("<type = ");
                        put(sqv.sqltype, 1);
                        put(">");

                end case;                 -- Of data types

            end if;                       -- If null or not

        end;                              -- Declare (rename) block

        if (col < sqlda.sqld) then       -- Add trailing space
             put(" ");
        end if;

    end loop;                             -- For processing columns

    new_line;                             -- Print end of line
end Print_Row;
--
-- Procedure:  Print_Error
-- Purpose:    SQLCA error detected. Retrieve the error message
--             and print it.
--

procedure Print_Error is

      exec sql begin declare section;
            error_buf: String(1..200); -- SQL error text retrieval
      exec sql end declare section;

begin

      exec sql inquire_sql (:error_buf = ERRORTEXT);
      put_line("SQL Error:");
      put_line(error_buf);

end Print_Error;

--
-- Procedure: Read_Stmt
-- Purpose:   Reads a statement from standard input. This routine
--          issues a prompt with the current statement number,
--          and reads the response into the parameter "stmt_buf".
--          No special scanning is done to look for terminators,
--          string delimiters or line continuations.
--
--          On eof-of-file end_error is raised and processed in
--          the main program.
--
--       The routine can be extended to allow line continuations,
--       SQL-style comments and a semicolon terminator.
-- Parameters:
--            stmt_num - Statement number for prompt.
--            stmt_buf - Buffer to fill from input.
--

procedure Read_Stmt (stmt_num: in Integer; stmt_buf: in out String) is

        slen: Natural;

begin                     -- Read_Stmt

      stmt_buf := (1..stmt_buf'length => ' ');
      slen := 0;
```

```
      while (slen = 0) loop          -- Ignore empty lines
             put(stmt_num, 3);
             put("> ");
             get_line(stmt_buf, slen);
      end loop;

end Read_Stmt;

--
-- Program: SQL_Monitor Main
-- Purpose: Main entry of SQL Monitor application. Prompt for
--          database name and connect to the database. Run the
--          monitor and disconnect from the database. Before
--          disconnecting roll
--          back any pending updates.
--

begin                                 -- Main Program

      put("SQL Database: ");           -- Prompt for database name
      get_line(dbname, dblen);
      if (dblen = 0) then
           return;
      end if;

      put_line("-- SQL Terminal Monitor --");

      -- Treat connection errors as fatal errors
      exec sql whenever sqlerror stop;
      exec sql connect :dbname;

      dbrun := TRUE;

      Run_Monitor;

      exec sql whenever sqlerror continue;

      exception
          when others =>       -- exit on EOF and other errors
              if (dbrun) then
                    put_line("SQL: Exiting monitor program.");
                    exec sql rollback;
                    exec sql disconnect;
              end if;

end SQL_Monitor;
```

A Dynamic SQL/Forms Database Browser

This program lets the user browse data from and insert data into any table in any database, using a dynamically defined form. The program uses Dynamic SQL and Dynamic FRS statements to process the interactive data. You should already have used VIFRED to create a Default Form based on the database table that you want to browse. VIFRED will build a form with fields that have the same names and data types as the columns of the specified database table.

When run, the program prompts the user for the name of the database, the table, and the form. The form is profiled using the **describe form** statement, and the field name, data type, and length information is processed. From this information, the program fills in the SQLDA data and null indicator areas and builds two Dynamic SQL statement strings to **select** data from and **insert** data into the database.

The **Browse** menu item retrieves the data from the database using an SQL cursor associated with the dynamic **select** statement, and displays that data using the dynamic **putform** statement. A **submenu** allows the user to continue with the next row or return to the main menu. The **Insert** menu item retrieves the data from the form using the dynamic **getform** statement, and adds the data to the database table using a prepared **insert** statement. The **Save** menu item commits the changes and, because prepared statements are discarded, reprepares the **select** and **insert** statements. When the **Quit** menu item is selected, all pending changes are rolled back and the program is terminated.

```
-- Declare the SQLCA and SQLDA records
exec sql include sqlca;
exec sql include sqlda;

--
-- Program:
--              Dynamic_FRS
-- Purpose:
--              Main entry of Dynamic SQL forms application
--

procedure Dynamic_FRS is

    -- Global SQLDA. Discriminant SQLN is implicitly
    -- set to IISQ_MAX_COLS
    sqlda: IISQLDA(IISQ_MAX_COLS);

    -- String object maximums
    MAX_NAME: constant := 40;       -- Input name size
    MAX_STRING: constant := 3000;  -- Large string buffer size
    MAX_STMT: constant := 1000;    -- SQL statement string size

    --
    -- Result storage pool for Dynamic SQL and FRS statements.
    -- This result pool consists of arrays of 4-byte integers,
    -- 8-byte floating-points, 2-byte indicators, and a large
    -- string buffer from which slices will be allocated. Each
    -- SQLDA SQLVAR sets its SQLDATA and SQLIND address pointers
    -- to variables from this pool.
```

```
--
integers:array(1..IISQ_MAX_COLS) of Integer; -- 4-byte integer
floats: array(1..IISQ_MAX_COLS) of Long_Float; -- 8-byte float
indicators:array(1..IISQ_MAX_COLS) of Short_Integer; -- 2-byte
                                                   -- indicator
characters: String(1..MAX_STRING);              -- String pool

exec sql begin declare section;
    dbname:   String(1..MAX_NAME); -- Database name
    formname: String(1..MAX_NAME); -- Form name
    tabname:  String(1..MAX_NAME); -- Database table name
    sel_buf:  String(1..MAX_STMT); -- Prepared SELECT and
    ins_buf:  String(1..MAX_STMT); -- INSERT statements
    err:      Integer;             -- Error status
    ret:      String(1..1);        -- Prompt error buffer
exec sql end declare section;

--
-- Function:
--        Describe_Form
-- Purpose:
--        Profile the specified form for name and data type
--        information.Using the DESCRIBE FORM statement, the
--        SQLDA is loaded with field information from the
--        form. This procedure processes this information to
--        allocate result storage, point at storage for
--        dynamic FRS data
--        retrieval and assignment, and build SQL statements
--        strings for subsequent dynamic SELECT and
--        INSERT statements. For example, assume the form
--        (and table) 'emp' has the following fields:
--
--           Field Name Type        Nullable?
--           ---------- ----        ---------
--            name      char(10)      No
--            age       integer4      Yes
--            salary    money       Yes
--
--   Based on 'emp', this procedure will construct the SQLDA.
--   The procedure allocates variables from a result variable
--   pool (integers, floats and a large character
--   string space). The SQLDATA and SQLIND fields are pointed
--   at the addresses of the result variables in the pool.
--   The following SQLDA is built:
--
--              sqlvar(1)
--                   sqltype = IISQ_CHA_TYPE
--                   sqllen  = 10
--                   sqldata = pointer into characters array
--                   sqlind  = null
--                   sqlname = 'name'
--              sqlvar(2)
--                   sqltype = -IISQ_INT_TYPE
--                   sqllen  = 4
--                   sqldata = address of integers(2)
--                   sqlind  = address of indicators(2)
--                   sqlname = 'age'
--              sqlvar(3)
--                   sqltype = -IISQ_FLT_TYPE
--                   sqllen  = 8
--                   sqldata = address of floats(3)
--                   sqlind  = address of indicators(3)
--                   sqlname = 'salary'
--
-- This procedure also builds two dynamic SQL statements
-- strings. Note that the procedure should be extended to
```

```
-- verify that the statement strings do fit into the
-- statement buffers (this was not done in this example).
-- The above example would construct the following
-- statement strings:
--
--     'SELECT name, age, salary FROM emp ORDER BY name'
--     'INSERT INTO emp (name, age, salary) VALUES (?, ?, ?)'
--
-- Parameters (globals):
--   formname - (in)  Name of form to profile.
--   tabname  - (in)  Name of database table.
--   sel_buf  - (out) Buffer to hold SELECT statement string.
--   ins_buf  - (out) Buffer to hold INSERT statement string.
-- Returns:
--       TRUE/FALSE - Success/failure - will fail on error
--                    or upon finding a table field.
--

function Describe_Form return Boolean is

    names:    String(1..MAX_STMT); -- Names for SQL statements
    name_cur: Integer;             -- Current name length
    name_cnt: Integer;             -- Bytes used in 'names'
    marks:    String(1..MAX_STMT); -- Place holders for INSERT
    mark_cnt: Integer;             -- Bytes used in 'marks
    nullable: Boolean;            -- Is nullable (SQLTYPE < 0)
    char_cnt: Integer;            -- Total string length
    char_cur: Integer;             -- Current string length

begin                 -- Describe_Form

    --
    -- DESCRIBE the form - if we cannot fully describe the
    -- form (our SQLDA is too small) then report an error and
    -- return.
    exec frs describe form :formname all into :sqlda;
    exec frs inquire_frs frs (:err = ERRORNO);
    if (err > 0) then
          return FALSE;              -- Error already displayed
    elsif (sqlda.sqld > sqlda.sqln) then
        exec frs prompt noecho
          ('SQLDA is too small for form :', :ret);
        return FALSE;
    elsif (sqlda.sqld = 0) then
        exec frs prompt noecho
           ('There are no fields in the form :', :ret);
        return FALSE;
    end if;

    --
    -- For each field determine the size and type of the
    -- result data area. This data area will be allocated out
    -- of the result variable pool (integers, floats and
    -- characters) and will be pointed at by SQLDATA and
    -- SQLIND.
    --
    -- If a table field type is returned then issue an error.
    --
    -- Also, for each field add the field name to the 'names'
    -- buffer and the SQL place holders '?' to the 'marks'
    -- buffer, which will be used to build the final SELECT
    -- and INSERT statements.
    --
    char_cnt := 1;            -- No strings used yet

    for i in 1 .. sqlda.sqld loop
```

```
  declare

    sqv: IISQL_VAR renames sqlda.sqlvar(i); -- Shorthand
    col: Integer := Integer(i);

  begin

   --
   -- Collapse all different types into Integers, Floats
   -- or Characters.
   --
if (sqv.sqltype < 0) then --Null indicator handled later
         nullable := TRUE;
    else
        nullable := FALSE;
    end if;

    case (abs(sqv.sqltype)) is

        -- Integers - use 4-byte integer
        when IISQ_INT_TYPE =>
             sqv.sqltype := IISQ_INT_TYPE;
             sqv.sqllen := 4;
             sqv.sqldata := integers(col)'Address;

        -- Floating points - use 8-byte floats
        when IISQ_MNY_TYPE | IISQ_FLT_TYPE =>
            sqv.sqltype := IISQ_FLT_TYPE;
            sqv.sqllen := 8;
            sqv.sqldata := floats(col)'Address;

        -- Character strings
        when
        IISQ_DTE_TYPE | IISQ_CHA_TYPE | IISQ_VCH_TYPE =>
            --
            -- Determine the length of the slic
            -- required from the large character buffer.
            -- If we have enough space left then point
            -- at the start of
            -- the corresponding slice, otherwise print
            -- an error and return.
            --
            -- Note that for DATE types we must set
            -- the length.
            --
            if (abs(sqv.sqltype) = IISQ_DTE_TYPE) then
                char_cur := IISQ_DTE_LEN;
            else
                char_cur := Integer(sqv.sqllen);
            end if;

            -- Enough room in large string buffer ?
            if (char_cnt + char_cur >
             characters'length) then
                exec frs prompt noecho
             ('Character pool buffer overflow :', :ret);
                return FALSE;
            end if;

            -- Allocate slice out of buffer
            sqv.sqltype = IISQ_CHA_TYPE;
            sqv.sqllen  = Short_Integer(char_cur);
            sqv.sqldata = characters(char_cnt)'Address;
            char_cnt    = char_cnt + char_cur;
```

```
            when IISQ_TBL_TYPE =>
                exec frs prompt noecho
                    ('Table field found in form :', :ret);
                return FALSE;

            when others =>
                exec frs prompt noecho
                     ('Invalid field type :', :ret);
                return FALSE;

        end case;                    -- Of data types

      -- Assign pointers to null indicators and toggle type
        if (nullable) then
             sqv.sqltype := -sqv.sqltype;
             sqv.sqlind := indicators(col)'Address;
        else
             sqv.sqlind := IISQ_ADR_ZERO;
        end if;

       --
       -- Store field names and place holders (separated by
       -- commas) for the SQL statements.
       --
        if (col = 1) then
            name_cnt = 1;
            mark_cnt := 1;
        else
            names(name_cnt) = ',';
            name_cnt := name_cnt + 1;
            marks(mark_cnt) = ',';
            mark_cnt := mark_cnt + 1;
        end if;
        name_cur := Integer(sqv.sqlname.sqlnamel);
        names(name_cnt..name_cnt+name_cur-1) :=
               sqv.sqlname.sqlnamec(1..name_cur);
        name_cnt := name_cnt + name_cur;
        marks(mark_cnt) := '?';
        mark_cnt := mark_cnt + 1;

    end;            -- Declare (renames) block

end loop;           -- While processing columns

--
-- Create final SELECT and INSERT statements. For the
-- SELECT statement ORDER BY the first field.
--
sel_buf := (1..sel_buf'length => ' ');
ins_buf := (1..ins_buf'length => ' ');
name_cur := Integer(sqlda.sqlvar(1).sqlname.sqlnamel);
sel_buf(1..7 + name_cnt-1 + 6 + tabname'length +
                10 + name_cur)
            := "SELECT " & names(1..name_cnt-1) &
              " FROM " & tabname &
              " ORDER BY " &
              sqlda.sqlvar(1).sqlname.sqlnamec(1..name_cur);
ins_buf(1..12 + tabname'length + 2 + name_cnt-1 + 10 +
        mark_cnt-1 + 1)
          := "INSERT INTO " & tabname & " (" &
               names(1..name_cnt-1) & ") VALUES (" &
               marks(1..mark_cnt-1) & ")";

      return TRUE;

  end Describe_Form;
```

```
--
-- Program:
--            Dynamic_FRS Main
-- Purpose:
--         Main body of Dynamic SQL forms application. Prompt for
--         database, form and table name. Call Describe_Form
--         to obtain a profile of the form and set up the SQL
--        statements. Then allow the user to interactively browse
--        the database table and append new data.
--

begin                          -- Dynamic_FRS Main

    exec sql declare sel_stmt statement;        -- Dynamic SQL
                                                -- SELECT statement
    exec sql declare ins_stmt statement;        -- Dynamic SQL
                                                -- INSERT statement
    exec sql declare csr cursor for sel_stmt;   -- Cursor for
                                                -- SELECT statement

    exec frs forms;

    -- Prompt for database name - will abort on errors
    exec sql whenever sqlerror stop;
    exec frs prompt ('Database name: ', :dbname);
    exec sql connect :dbname;

    exec sql whenever sqlerror call sqlprint;

    --
    -- Prompt for table name - later a Dynamic SQL SELECT
    -- statement will be built from it.
    --
    exec frs prompt ('Table name: ', :tabname);

    --
    -- Prompt for form name. Check forms errors reported
    -- through INQUIRE_FRS.
    --
    exec frs prompt ('Form name: ', :formname);
    exec frs message 'Loading form ...';
    exec frs forminit :formname;
    exec frs inquire_frs frs (:err = ERRORNO);
    if (err > 0) then
          exec frs message 'Could not load form. Exiting.';
          exec frs endforms;
          exec sql disconnect;
          return;
    end if;

    -- Commit any work done so far - access of forms catalogs
    exec sql commit;

    -- Describe the form and build the SQL statement strings
    if (not Describe_Form) then
        exec frs message 'Could not describe form. Exiting.';
        exec frs endforms;
        exec sql disconnect;
        return;
    end if;

    --
    -- PREPARE the SELECT and INSERT statements that correspond
    -- to the menu items Browse and Insert. If the Save menu item
    -- is chosen the statements are reprepared.
```

```
--
exec sql prepare sel_stmt from :sel_buf;
err := sqlca.sqlcode;
exec sql prepare ins_stmt from :ins_buf;
if ((err < 0) or (sqlca.sqlcode < 0)) then
    exec frs message 'Could not prepare SQL statements. Exiting.';
    exec frs endforms;
    exec sql disconnect;
    return;
end if;

--
-- Display the form and interact with user, allowing browsing
-- and the inserting of new data.
--
exec frs display :formname fill;
exec frs initialize;
exec frs activate menuitem 'Browse';
exec frs begin;
    --
    -- Retrieve data and display the first row on the form,
    -- allowing the user to browse through successive rows.
    -- If data types from the database table are not
    -- consistent with data descriptions obtained from the
    -- form, a retrieval error
    -- will occur. Inform the user of this or other errors.
    --
    -- Note that the data will return sorted by the first
    -- field that was described, as the SELECT statement,
    -- sel_stmt, included an ORDER BY clause.
    --
    exec sql open csr;

    -- Fetch and display each row
    while (sqlca.sqlcode = 0) loop

        exec sql fetch csr using descriptor :sqlda;
        if (sqlca.sqlcode <= 0) then
            exec sql close csr;
            exec frs prompt noecho ('No more rows :', :ret);
            exec frs clear field all;
            exec frs resume;
        end if;

        exec frs putform :formname using descriptor :sqlda;
        exec frs inquire_frs frs (:err = ERRORNO);
        if (err > 0) then
           exec sql close csr;
           exec frs resume;
        end if;

        -- Display data before prompting user with submenu
        exec frs redisplay;

        exec frs submenu;
        exec frs activate menuitem 'Next', frskey4;
        exec frs begin;
            -- Continue with cursor loop
            exec frs message 'Next row ...';
            exec frs clear field all;
        exec frs end;
        exec frs activate menuitem 'End', frskey3;
        exec frs begin;
            exec sql close csr;
            exec frs clear field all;
            exec frs resume;
```

```
            exec frs end;

        end loop;                  -- While there are more rows
    exec frs end;

    exec frs activate menuitem 'Insert';
    exec frs begin;
        exec frs getform :formname using descriptor :sqlda;
        exec frs inquire_frs frs (:err = ERRORNO);
        if (err > 0) then
              exec frs clear field all;
              exec frs resume;
        end if;
        exec sql execute ins_stmt using descriptor :sqlda;
        if ((sqlca.sqlcode < 0) or (sqlca.sqlerrd(3) = 0)) then
             exec frs prompt noecho ('No rows inserted :', :ret);
        else
             exec frs prompt noecho ('One row inserted :', :ret);
        end if;
    exec frs end;

    exec frs activate menuitem 'Save';
    exec frs begin;
        --
        -- COMMIT any changes and then re-PREPARE the SELECT and
        -- INSERT statements as the COMMIT statements discards
        -- them.
        --
        exec sql commit;
        exec sql prepare sel_stmt FROM :sel_buf;
        err := sqlca.sqlcode;
        exec sql prepare ins_stmt FROM :ins_buf;
        if ((err < 0) or (sqlca.sqlcode < 0)) then
            exec frs prompt noecho
                ('Could not reprepare SQL statements :', :ret);
            exec frs breakdisplay;
        end if;
    exec frs end;

    exec frs activate menuitem 'Clear';
    exec frs begin;
         exec frs clear field all;
    exec frs end;

    exec frs activate menuitem 'Quit', frskey2;
    exec frs begin;
        exec sql rollback;
        exec frs breakdisplay;
    exec frs end;
    exec frs finalize;

    exec frs endforms;
    exec sql disconnect;

    exception
        when others =>
            exec frs prompt noecho
                 ('Unexpected exception encountered :', :ret);
            raise;

end Dynamic_FRS;
```

Chapter 6: Embedded SQL for BASIC

This chapter describes the use of Embedded SQL with the BASIC programming language.

Embedded SQL Statement Syntax for BASIC

This section describes the language-specific issues inherent in embedding SQL database and forms statements in a BASIC program. An Embedded SQL database statement has the following general syntax:

[*margin*] **exec sql** *SQL_statement*

The syntax of an Embedded SQL/FORMS statement is almost identical:

[*margin*] **exec frs** *SQL/FORMS_statement*

For information on SQL statements, see the *SQL Reference Guide*. For information on SQL/FORMS statements, see the *Forms-based Application Development Tools User Guide*.

The sections below describe the various syntactical elements of these statements as implemented in BASIC.

Margin

In general, Embedded SQL statements in BASIC require no special margins. The **exec** keyword can begin anywhere on the source line. Host declarations can also begin on any column.

BASIC Line Numbers

The BASIC line number, while not required, can occur at the beginning of any Embedded SQL statement. For example:

```
100     EXEC SQL DROP TABLE emp
```

In most instances, the preprocessor outputs any BASIC line number that precedes an Embedded SQL statement. However, in a few cases the preprocessor ignores a BASIC line number and does not include it in the code it generates. For example, line numbers occurring on Embedded SQL statements that produce no BASIC code are ignored by the preprocessor. It is an error to put a line number on a continuation line for an Embedded SQL statement or declaration.

The preprocessor never generates line numbers of its own. Thus, if you prefix an Embedded SQL statement with a line number and the preprocessor translates that statement into several BASIC statements, the line number will appear before the first BASIC statement only. Subsequent BASIC statements will be unnumbered. The BASIC line number, if present, must be the first item on the line. It can be preceded only by spaces or tabs.

Note that the BASIC language does require a line number on the first line of a program or subprogram. The Embedded SQL preprocessor does not verify that this line number exists.

Terminator

There is no terminator for Embedded SQL/BASIC. Following the end of an Embedded SQL statement in BASIC, only comments and white space (blanks and tabs) are allowed to the end of the line.

The preprocessor allows, but does not require, a semicolon as a statement terminator for Embedded SQL statements. It does not write the semicolon to the output file of BASIC code. The terminating colon can be convenient when entering source code directly from the terminal, using the **-s** flag on the preprocessor command line to test the syntax of a particular statement (see Advanced Processing in this chapter).

Labels

Like BASIC statements, Embedded SQL statements can have a label prefix. The label must begin with an alphabetic character and the remaining characters, if present, can be any combination of alphabetic and numeric characters and underscores. Note that dollar signs ($) and periods (.) are not permitted in labels preceding Embedded SQL statements, even though the BASIC compiler accepts these characters. The label must be separated from the statement it labels with a colon. For example:

```
Close_Csr: exec sql close cursor1
```

The label can appear before any Embedded SQL statement. As with line numbers, in most instances the preprocessor outputs any BASIC label that precedes an Embedded SQL statement. However, in a few cases the preprocessor ignores a BASIC label and does not include it in the code it generates. For example, the preprocessor ignores labels occurring on Embedded SQL statements that do not produce BASIC code. It is an error to put a label on a continuation line for an Embedded SQL statement.

A label can be preceded by a BASIC line number. For example:

```
100     Close_down:     exec sql disconnect
```

Line Continuation

Embedded SQL statements and variable declarations can be continued over multiple lines. The line continuation rules are the same as those for BASIC statements. The ampersand (&) character followed immediately by a newline character indicates to the preprocessor that the current statement or declaration is to be continued. For example, the following **select** statement is continued over four lines:

```
exec sql select ename           &
        into :namevar           &
        from employee           &
        where eno = :numvar
```

Blank lines can be included between Embedded SQL statement lines and do not require a continuation indicator. If a line continuation character is missing from the end of a line containing an Embedded SQL statement to be continued, the preprocessor generates the error message:

```
"Syntax error on terminator or missing BASIC continuation indicator."
```

The preprocessor does not enforce strict line continuation rules in declaration sections.

Comments

You can include a comment field or line in an Embedded SQL statement by typing the exclamation point (!) at the beginning of the comment field. The following example shows the use of a comment field on the same line as an Embedded SQL statement:

```
exec sql open empcsr            ! Process employees
```

The next example shows the use of a comment field embedded in an SQL statement:

```
exec sql select ename                    &
            into :namevar                &
            from employee                &
! Confirm that "eno" is the same as
! the current value chosen
            where eno = :currentval
```

In both cases, the preprocessor ignores the comment field. Note that a comment field terminates with the newline. A comment field cannot be continued over multiple lines.

A comment line can appear anywhere in an Embedded SQL program that a blank line is allowed, with the following exceptions:

- In string constants. Such a comment would be interpreted as part of the string constant.
- In parts of statements that are dynamically defined. For example, a comment in a string variable specifying a form name is interpreted as part of the form name.
- Between component lines of Embedded SQL block-type statements. All block-type statements (such as **activate** and **unloadtable**) are compound statements that include a statement section delimited by **begin** and **end**. Comment lines must not appear between the statement and its section. The preprocessor would interpret such comments as BASIC host code, causing preprocessor syntax errors. (Note, however, that the comment begun by the exclamation point can appear on the same line as the statement.) For example, the following statement would cause a syntax error on the first comment:

```
exec frs unloadtable empform employee (:namevar = ename)
    ! Illegal comment before statement body.
        exec frs begin
    ! Comment legal here
            exec frs message :namevar ! And legal here too
        exec frs end
```

- Statements that are made up of more than one compound statement, such as the **display** statement, which typically consists of the **display** clause, an **initialize** section, **activate** sections, and a **finalize** section, cannot have comments between any of the components. These comments would be translated as host code and would cause syntax errors on subsequent statement components.

A comment line can also begin with the BASIC **rem** keyword.

The SQL comment delimiter "--" acts just like the "!" delimiter; it indicates that the rest of the line is a comment.

String Literals

Embedded SQL string literals are delimited by single quotes. For example:

```
exec sql update employee        &
        set salary = 30000.00   &
        where name = 'Newman'
```

Quotes cannot be embedded in a string literal. If you want to use a quote in a character string in an Embedded SQL statement, assign the string into a string variable or a BASIC string constant and use the string variable or constant in the SQL statement. For example:

```
comm_str = "Doesn't seem to relax"
exec sql update employee                &
        set comments = :comm_str        &
        where eno = :numvar
```

You can also declare a BASIC string constant. Following BASIC rules, you cannot continue string literals over more than one line.

Integer Literals

You can use the optional trailing percent sign (%) with Embedded SQL integer literals. The preprocessor always adds the percent sign to the integer literals that it generates.

The Create Procedure Statement

As mentioned in the *SQL Reference Guide*, the **create procedure** statement has language-specific syntax rules for line continuation, string literal continuation, comments, and the final terminator. These syntax rules follow the rules discussed in this section—for example, the ampersand is used to continue lines. Regardless of the number of statements inside the procedure body, the preprocessor treats the **create procedure** statement as a single statement and, as an Embedded SQL/BASIC statement, it has no final terminator. However, you must terminate all statements in the body of the procedure with a semicolon.

The following example shows a **create procedure** statement that follows the Embedded SQL/BASIC syntax rules:

```
exec sql                                        &
    create procedure proc (parm integer) as     &
    declare  &
        var integer;
    begin &
        ! Use BASIC comment field (no need to continue here)
        if parm  10 then &
            message 'BASIC strings cannot continue over lines';&
            insert into tab VALUES (:parm); &
        endif; &
end ! No terminator in BASIC
```

Decimal Literals

The preprocessor distinguishes between decimal and floating-point literals in SQL and Forms Runtime System (FRS) statements according to the following rules:

- A literal containing a decimal point with no E notation is a decimal literal.
- A literal with E notation is a floating-point literal.

For example:

```
exec sql insert
         into mytable (salary) values (23000.12)
exec sql insert
         into mytable (number) values (1.4E4)
```

In addition, the preprocessor treats integer literals greater than MAXINT as decimals. This allows host programs to input large integer values.

Ingres will treat "23000.00" as a decimal literal and "1.4E2" as a float literal.

However, applications will continue to use host language rules for interpreting literals appearing in host declarations. For example:

```
exec sql begin declare section
        integer2 i (1.234)
 exec sql end declare section
```

The literal '1.234' is interpreted according to the BASIC compiler rules.

This is consistent with the Ingres convention of interpreting SQL statements according to SQL rules and host statements according to host language compiler rules.

BASIC Variables and Data Types

This section describes how to declare and use BASIC program variables in Embedded SQL.

Variable Declarations

The following sections describe variable declarations.

Embedded SQL Variable Declaration Sections

Embedded SQL statements use BASIC variables to transfer data to and from the database or a form into the program. BASIC constants can also be used in those SQL statements transferring data from the program into the database. You must declare BASIC variables, constants, and structure definitions to SQL before using them in any Embedded SQL statements. The preprocessor does not allow implicit variable declarations. For this reason, the "%" and "$" suffixes cannot be used with variable names. BASIC variables are declared to SQL in a *declaration section*. This section has the following syntax:

exec sql begin declare section
BASIC variable declarations
exec sql end declare section

Embedded SQL variable declarations are global to the program file from the point of declaration onwards. Multiple declaration sections can be incorporated into a single file, as would be the case when a few different BASIC subprograms issue embedded statements using local variables. Each subprogram can have its own declaration section. For a discussion of the declaration of variables and types that are local to BASIC subprograms, see The Scope of Variables in this chapter.

Reserved Words in Declarations

All Embedded SQL keywords are reserved. Therefore, you cannot declare variables with the same name as ESQL keywords. You can only use them in quoted string literals. These words are:

byte	**decimal**	**external**	**record**
case	**dim**	**integer**	**single**
com	**dimension**	**long**	**string**
common	**double**	**map**	**variant**
constant	**dynamic**	**real**	**word**

The Embedded SQL preprocessor does not distinguish between uppercase and lowercase in keywords. In generating BASIC code, it converts any uppercase letters in keywords to lowercase.

Data Types

The Embedded SQL preprocessor accepts the following elementary BASIC data types. The table below maps these types to their corresponding Ingres type categories. For a description of exact type mapping, see Data Type Conversion in this chapter.

BASIC Data Types and Corresponding Ingres Types

BASIC Type	Ingres Type
string	character
integer	integer
long	integer
word	integer
byte	integer
real	float
single	float
double	float
double	decimal

Because BASIC supports the packed decimal datatype, the Ingres decimal type is mapped to it. For example, the BASIC packed decmial declarations:

```
declare decimal pack1
declare decimal (p,s) pack2
```

correspond to the Ingres **decimal types**:

```
    decimal (15, 2)
    decimal (p,s)
```

In addition, the preprocessor accepts the BASIC record type in variable declarations, providing the record has been predefined in an Embedded SQL declaration section.

The data types **gfloat** and **hfloat** are illegal and will cause declaration errors.

Neither the preprocessor nor the runtime support routines support **gfloat** or **hfloat** floating-point arithmetic. Consequently, the precision of floating-point data is less than that which is available in VMS BASIC programs. You should not compile the BASIC source code with the command line qualifiers **gfloat** or **hfloat** if you intend to pass those floating-point values to or from Ingres objects.

The following sections discuss the variable declarations and the use of variables in Embedded SQL statements.

The String Data Type

The Embedded SQL preprocessor accepts both fixed-length and dynamic string declarations. Strings can be declared using any of the declarations listed later. Note that you can indicate string length only for non-dynamic strings, that is, for string declarations appearing in common, map, or record declarations. For example,

```
common (globals) string ename = 30
```

is acceptable, but

```
declare string bad_str_var = 30 ! length is illegal
```

will generate an error.

The reference to an uninitialized BASIC dynamic string variable in an embedded statement that assigns the value of that string to Ingres will result in a runtime error because that restriction does not apply to the retrieval of data into an uninitialized dynamic string variable.

The Integer Data Type

Embedded SQL/BASIC accepts all BASIC integer data type sizes. It is important that the preprocessor know about **integer** size because it generates code to load data in and out of program variables. The preprocessor assumes that integer size is four bytes by default. However, you can inform the preprocessor of a non-default integer size by using the **-i** flag on the preprocessor command line. For detailed information on this flag, see Advanced Processing in this chapter.

You can explicitly override the default size or the preprocessor **-i** command-line flag by using the BASIC subtype words **byte**, **word**, or **long** in the variable declaration, as these examples illustrate:

```
declare byte one_byte_int
common (globals) word two_byte_int
external long four_byte_int
```

These declarations instruct the preprocessor to create integer variables of one, two, and four bytes respectively, regardless of the default setting.

You can use an integer variable with any numeric-valued object to assign or receive numeric data. For example, you can use such a variable to set a field in a form or to select a column from a database table. It can also specify simple numeric objects, such as table field row numbers.

The Real Data Type

As with the **integer** data type, the preprocessor must know the size of real data variables so that these variables can interact with Ingres correctly at runtime. The preprocessor accepts two sizes of real data: 4-byte variables (the default) and 8-byte variables. Again, you can change the default size with a flag on the preprocessor command line—in this case, the **-r** flag. For detailed information on this flag, see Advanced Processing in this chapter.

You can explicitly override the default size by using the BASIC subtype words **single** or **double** in a variable declaration. For example, the following two declarations:

```
declare single four_byte_real
map (myarea) double eight_byte_real
```

create real variables of four and eight bytes, respectively, regardless of the default setting.

A real variable can be used in Embedded SQL statements to assign or receive numeric data (both real and integer) to and from database columns, form fields, and table field columns. It cannot be used to specify numeric objects, such as table field row numbers.

The Decimal Data Type

The preprocessor accepts variable declarations of the **decimal** data type. Note that because the current implementation of Ingres does not store data in packed decimal format, Ingres converts the contents of a decimal variable to and from a double at runtime. Therefore, although decimal variables can interact with Ingres, the movement of data at runtime, both before and after database manipulation, can lead to some loss of precision.

Decimal variables can be used in Embedded SQL statements to transmit numeric values to and from database columns, form fields, and table field columns. You cannot, however, use decimal variables with Ingres integer objects, such as table field row numbers.

The default scale and precision for both **decimal variables** and **decimal symbolic constants** in EQUEL/BASIC is the BASIC default of (15,2). The preprocessor does not support the BASIC compile flag /decimal_size. Compiling with the flag will not change the default precision and scale of decimal variables as far as the preprocessor is concerned. You should always specify the precision and scale when declaring a decimal variable or constant. For example:

```
declare decimal (10.4) constant = 1.2345 - Preferred declaration

declare decimal constant = 1.234         - Will use default (15,2) thus
                                           scale is truncated to two places.
```

The Record Data Type

The Embedded SQL preprocessor supports the declaration and use of user-defined record variables. You can declare a variable of type **record** if you have already defined the record in an Embedded SQL declaration section. Later sections discuss the syntax of record declarations and their use in Embedded SQL statements.

Variable and Constant Declaration Syntax

Embedded SQL/BASIC variables and constants can be declared in a variety of ways when those declarations are in a declare section. The following sections enumerate these declaration statements and describe their syntax.

The Declare Statement

The **declare** statement for an Embedded SQL/BASIC variable has the following syntax:

declare *type identifier* [**(***dimensions)*] {**,** [*type*] *identifier* [**(***dimensions)*]}

The **declare** statement for an Embedded SQL/BASIC constant has the syntax:

declare ***t****ype* ***constant i****dentifier* = *literal* {**, *i****dentifier* = *literal}*

Syntax Notes:

- If you specify the word **constant**, the declared constants cannot be targets of Ingres assignments.
- The *type* must be a BASIC type acceptable to the preprocessor (see previous section) or, in the case of variables only, a **record** type already defined in the Embedded SQL declaration section. Note that the type is mandatory for Embedded SQL/BASIC declarations, because the preprocessor has no notion of a default type. You need only specify the type once when declaring a list of variables of the same type.
- The *dimensions* of an array specification are not parsed by the preprocessor. Consequently, the preprocessor does not check bounds. Note also that the preprocessor will accept an illegal dimension, such as a non-numeric value, but this will later cause BASIC compiler errors.

The following example illustrates the use of the **declare** statement:

```
exec sql begin declare section
        declare integer enum, eage, string ename
        declare single constant minsal = 12496.62
        declare real esal(100)
        declare word null_ind        ! Null indicator
exec sql end declare section
```

The Dimension Statement

The **dimension** statement can be used to declare arrays to the preprocessor. Its syntax is:

dimension | **dim** *type identifier**(dimensions)*** **{,** [*type*] *identifier* **(***dimensions***)}**

Syntax Notes:

- The *type* must be a BASIC type acceptable to the preprocessor (see previous section) or a record already defined in the Embedded SQL declaration section. Note that the type is mandatory for Embedded SQL/BASIC declarations because the preprocessor has no notion of a default type. You need only specify the type once when declaring a list of variables of the same type.
- The *dimensions* of an array specification are not parsed by the preprocessor. Consequently, the preprocessor does not check bounds. Note also that the preprocessor will accept an illegal dimension, such as a non-numeric value, but it will later cause BASIC compiler errors. Furthermore, the preprocessor does not distinguish between executable and declarative **dimension** statements. If you have used the **dimension** statement to declare an executable array to Embedded SQL/BASIC, subsequent executable dimension statements of the same array in a declaration section will cause a redeclaration error.

The following example illustrates the use of the **dimension** statement:

```
exec sql begin declare section
    dim string employee_names(100,20)
                ! declarative DIM statement
    dimension long emp_id(100,2,2)
    dimension double expenses(numdepts)
                ! executable DIM statement

exec sql end declare section
```

Static Storage Variable Declarations

Embedded ESQL/BASIC supports the BASIC **common** and **map** variable declarations. The syntax for a **common** variable declaration is as follows:

> **common** | **com** [**(***com_name)*]
> *type identifier* [**(***dimensions)*] [**=** ***str_length***]
> {**,** [*type*] *identifier* [**(***dimensions)*] [**=** ***str_length***]}

The syntax for a **map** variable declaration is as follows:

> **map** | **map dynamic (***map_name)*
> *type identifier* [**(***dimensions)*] [**=** ***str_length***]
> {**,** [*type*] *identifier* [**(***dimensions)*] [**=** ***str_length***]}

Syntax Notes:

- The *type* must be a BASIC type acceptable to the preprocessor (see previous section) or a **record** type already defined to Embedded SQL/BASIC. Note that the type is mandatory for Embedded SQL/BASIC declarations because the preprocessor has no notion of a default type. You need only specify the type once when declaring a list of variables of the same type.
- The *dimensions* of an array specification are not parsed by the preprocessor. Consequently, the preprocessor does not check bounds. Note also that the preprocessor will accept an illegal dimension, such as a non-numeric value, but it will later cause BASIC compiler errors.
- The string length, if present, must be a simple integer literal.
- The *com_name* or *map_name* clause is not parsed by the preprocessor. Consequently, the preprocessor will accept common and map areas of the same name in a single declaration section. It will also accept a **map dynamic** statement whose *com_name* has not appeared in another **map** statement. Either of these situations will later cause BASIC compiler errors.

The following example uses the **common** and **map** variable declarations:

```
exec sql begin declare section
    common (globals) string address = 30, integer zip
    map (ebuf) byte eage, string
                ename = 20, single emp_num
    common (globals) integer empid (200)

exec sql end declare section
```

The External Statement

You can inform Embedded SQL/BASIC of variables and constants declared in an external module. The syntax for a variable is as follows:

> **external** *type identifier {, identifier}*

The syntax for a constant is as follows:

> **external** *type constant identifier {, identifier}*

Syntax Note:

Embedded SQL/BASIC applies the same restrictions on *type* as VAX-11 BASIC.

```
exec sql begin declare section
    external integer empform, infoform
    external single constant emp_minsal

exec sql end declare section
```

Record Type Definitions

Embedded SQL/BASIC accepts BASIC record definitions. The syntax of a record definition is:

> **record** *identifier*
> *record_component*
> {*record_component*}
> **end record [*i***dentifier*]

where *record_component* can be any of the following:

> *type identifier* [**(***dimensions)*] [**= *s***tr_length*]
> {**,** [*type*] *identifier* [**(***dimensions)*] [**=** *str_length*]}
>
> *group_clause*
>
> *variant_clause*

In turn, the syntax of a *group_clause* is:

> **group *i***dentifier* [**(***dimensions)*]
> *record_component*
> {*record_component*}
> **end group** [*identifier*]

The syntax of a *variant_clause* is:

> **variant**
> *case_clause*
> {*case_clause*}
> **end variant**

where *case_clause* consists of:

> **case**
> *record_component*

Syntax Notes:

- The *type* must be a BASIC type acceptable to the preprocessor (see previous section) or a **record** type already defined in the declaration section. Note that the type is mandatory for Embedded SQL/BASIC declarations because the preprocessor has no notion of a default type. You need only specify the type once when declaring a list of variables of the same type.
- Use the *str_length* clause only with record components of type **string**.
- Record definitions must appear before declarations using that record type.

The following example contains record type definitions:

```
exec sql begin declare section
    record emp_history
            string ename = 30
            group prev_employers(10)
                string comp_name = 30
                real salary
                integer num_years
            end group prev_employers
    end record emp_history
    record emp_sports
            string ename = 30
            variant
                case
                        group golf
                                integer handicap
                                string club_name
                        end group golf
                case
                        group baseball
                                integer batting_avg
                                string team_name
                        end group baseball
                case
                        group tennis
                                integer seed
                                string club_name
                        end group tennis
            end variant
    end record emp_sports

exec sql end declare section
```

Indicator Variables

An *indicator variable* is a 2-byte integer variable. There are three possible ways to use them in an application:

- In a statement that retrieves data from Ingres, you can use an indicator variable to determine if its associated host variable was assigned a null value.
- In a statement that sets data to Ingres, you can use an indicator variable to assign a null to the database column.
- In a statement that retrieves character data from Ingres, you can use the indicator variable as a check that the associated host variable was large enough to hold the full length of the returned character string. You can use **SQLSTATE** to do this. Although you can use **SQLCODE** as well, it is preferable to use **SQLSTATE** because **SQLCODE** is a deprecated feature.

You can declare an indicator using the integer **word** subtype or, if you used the **-i2** preprocessor command line flag, you can declare an indicator as an **integer**. The following example declares two indicator variables, one a single variable and the other an array of indicators:

```
declare word ind, ind_arr(10)
```

When using an indicator variable with a BASIC record, you must declare the indicator variable as an array of 2-byte integers. In the above example, you can use the variable "ind_arr" as an indicator array with a record assignment.

The DCLGEN Utility

DCLGEN (Declaration Generator) is a record-generating utility that maps the columns of a database table into a record that can be included in a declaration section.

Use the following command to invoke DCLGEN from the operating system level:

dclgen *language dbname tablename filename recordname*

where

- *language* is the Embedded SQL host language, in this case, "basic."
- *dbname* is the name of the database containing the table.
- *tablename* is the name of the database table.
- *filename* is the output file into which the record declaration is placed.
- *recordname* is the name of the BASIC record variable that the command generates. The command generates a record definition named *recordname* followed by an underscore character (_) and a declaration for a record variable of *recordname*.

This command creates the declaration file *filename*, containing a record corresponding to the database table. The file also includes a **record** statement for the record variable, as well as a **declare table** statement that serves as a comment and identifies the database table and columns from which the record was generated.

Once the file has been generated, you can use an Embedded SQL **include** statement to incorporate it into the variable declaration section. The following example demonstrates how to use DCLGEN in a BASIC program.

Assume the Employee table was created in the Personnel database as:

```
exec sql create table employee
        (eno    smallint not null,
         ename  char(20) not null,
         age    integer1,
         job    smallint,
         sal    decimal not null,
         dept   smallint)
```

and the DCLGEN system-level command is:

```
dclgen basic personnel employee employee.dcl emprec
```

The employee.dcl file created by this command contains a comment and three statements. The first statement is the **declare table** description of "employee" which serves as a comment. The second statement is a definition of the BASIC record "emprec_". The last statement is a **declare** statement for the record "emprec". The contents of the employee.dcl file are:

```
!   Description of table employee from database personnel
    exec sql declare employee table
           (eno smallint not null,              &
            ename        char(20) not null,     &
            age          integer1,              &
            job         smallint,               &
            sal         decimal not null,       &
            dept        smallint)

    record emprec_
            word        eno
            string      ename = 20
            byte        age
            word        job
            double      sal
            word        dept
    end record
    declare emprec_ emprec
```

This file should be included, by means of the Embedded SQL **include** statement, in an Embedded SQL declaration section:

```
exec sql begin declare section
        exec sql include 'employee.dcl'
exec sql end declare section
```

You can then use the emprec record in a **select**, **fetch**, or **insert** statement.

DCLGEN and Large Objects

You can use DCLGEN to generate an appropriate **declare table** statement with Ada variables for tables that contain **long varchar** columns. For columns that have a limited length, the variables generated will be identical to the variables generated for the Ingres **varchar** datatype. For columns with unlimited length, such as:

create table long_obj_table(blob_col long varchar);

DCLGEN will issue an error message and generate a character string variable with zero length. You can modify the length of the generated variable before attempting to use the variable in an application.

For example the following table definition:

```
create tablelongobj_table
    (long_column    long varchar));
```

results in the following DCLGEN generated output for BASIC compilers that support structures:

```
exec sql declare long_obj_table table     &
    (long_column   long varchar)

record blobs_rec_
    string long column = 0
end record blobs_rec_
declare blobs_rec_ blobs_rec
```

Assembling and Declaring External Compiled Forms

You can pre-compile your forms in the Visual Forms Editor (VIFRED). This saves the time that would be otherwise required at runtime to extract the form's definition from the database forms catalogs. When you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED prompts you for the name of the file with the MACRO description. After you have created the file, you can use the following VMS command to assemble it into a linkable object module:

macro *filename*

This command produces an object file containing a global symbol with the same name as your form. Before the Embedded SQL/FORMS statement **addform** can refer to this global object, the object must be declared in an Embedded SQL declaration section with the following syntax:

external integer *formname*

Syntax Notes:

- The *formname* is the actual name of the form. VIFRED gives this name to the address of the global object. The *formname* is also used as the title of the form in other Embedded SQL/FORMS statements.
- The **external** statement associates the object with the external form definition.

The example below shows a typical form declaration and illustrates the difference between using the form's object definition and the form's name.

```
exec sql begin declare section
    external integer empform
    ...

exec sql end declare section
    ...
 exec frs addform :empform       ! The global object
exec frs display empform         ! The name of the form
    ...
```

Concluding Example

The following example demonstrates some simple Embedded SQL/BASIC declarations:

```
exec sql include sqlca
exec sql begin declare section
    declare byte                d_byte ! variables of each data type
    declare word                d_integer2
    declare long                d_integer4
    declare integer             d_integer_def
    declare single              d_real4
    declare double              d_real8
    declare real                d_real_def
    declare decimal(6,2)        d_decimal
    declare string              d_string
    declare integer constant num_depts = 10 ! constant
    common(globs) real e_raise ! static storage variables
    map (ebuf) string ename = 20
    dim string                  emp_names(100,30) ! array declarations
    declare integer     dept_id(10)
    common(globs) string e_address(40) = 30
    record person ! Variant record
        byte age
        long flags
        variant
            case
                group emp_list
                        string full_name = 30
                end group
            case
                group emp_directory
                        string firstname = 12
                        string lastname = 8
                end group
        end variant
    end record

declare person p_table(100)            ! Array of records

exec sql include 'employee.dcl'        ! From DCLGEN

external integer empform, deptform     ! Compiled forms
dim word indicators(10)                ! Array of null indicators

exec sql end declare section
```

The Scope of Variables

All variables declared in an Embedded SQL declaration section can be referenced, and are accepted by the preprocessor, from the point of declaration to the end of the file. This may not be true for the BASIC compiler, which only allows variables to be referred to in the scope of the program unit in which they were declared. If you have two unrelated subprograms in the same file, each of which contains a variable with the same name to be used by Embedded SQL, you should *not* redeclare the variable to Embedded SQL. The preprocessor will use the data type information supplied the first time you declared the variable.

In the following program fragment, the variable "dbname" is passed as a parameter between two subroutines. In the first subroutine, the variable is a local variable. In the second subroutine, the variable is a formal parameter passed as a string to be used with the **connect** statement. In both subroutines, the preprocessor uses the data attributes from the variable's declaration in the first subroutine.

```
100 sub Scopes
    exec sql include sqlca
    exec sql begin declare section
        declare string dbname
    exec sql end declare section
    ! Prompt for and read database name
    print 'Database: '
    input dbname
    call open_db(dbname)
    ...

    end sub

200 sub Open_Db(string dbname)

    exec sql include sqlca
    exec sql whenever sqlerror stop
    exec sql connect :dbname
            ! Declared to SQL in first subroutine
    ...
    end sub
```

Special care should be taken when using variables in a **declare cursor** statement. The variables used in such a statement must also be valid in the scope of the **open** statement for that same cursor. The preprocessor actually generates the code for the **declare** at the point that the **open** is issued and, at that time, evaluates any associated variables. For example, in the following program fragment, even though the variable "number" is valid to the preprocessor at the point of both the **declare cursor** and **open** statements, it is not an explicitly declared variable name for the BASIC compiler at the point that the **open** is issued, possibly resulting in a runtime error. Because BASIC allows implicit variable declarations (although Embedded SQL does not), the compiler itself will not, however, generate an error message.

```
100 sub Init_Csr ! This example contains an error
    exec sql include sqlca
    exec sql begin declare section
        declare integer number ! a local variable
    exec sql end declare section
    exec sql declare cursor1 cursor for &
        select ename, age &
        from employee &
        where eno = :number
                ! initialize "number" to a particular value
        ...
    end sub

200         sub process_csr
    exec sql include sqlca
    exec sql begin declare section
        declare string ename
        declare integer eage
    exec sql end declare section
    exec sql open cursor1
             ! illegal evaluation of "number"
    exec sql fetch cursor1 into :ename, :eage

end sub
```

Note that you must issue **include sqlca** statement in each subprogram containing Embedded SQL statements.

Variable Usage

BASIC variables declared in an Embedded SQL declaration section can substitute for most non key-word elements of Embedded SQL statements. Of course, the variable and its data type must make sense in the context of the element. When you use a BASIC variable in an Embedded SQL statement, you must precede the variable with a colon. You must further verify that the statement using the variable is in the scope of the variable's declaration. As an example, the following **select** statement uses the variables "namevar" and "numvar" to receive data, and the variable "idno" as an expression in the **where** clause:

```
exec sql select ename, eno              &
    into :namevar, :numvar              &
    from employee                       &
    where eno = :idno
```

Various rules and restrictions apply to the use of BASIC variables in Embedded SQL statements. The sections below describe the usage syntax of different categories of variables and provide examples of such use.

Simple Variables

A simple scalar-valued variable (integer, real or character string) is referred to by the syntax:

:simplename

Syntax Notes:

- If you use the variable to send values to Ingres, it can be any scalar-valued variable or constant.
- If you use the variable to receive values from Ingres, it can only be a scalar-valued variable.
- The reference to an uninitialized BASIC dynamic string variable in an embedded statement that assigns the value of that string to Ingres results in a runtime error because an uninitialized dynamic string points at a zero address. This restriction does not apply to the retrieval of data into an uninitialized dynamic string variable.

The following program fragment demonstrates a typical message-handling routine that has two scalar valued variables, "buffer" and "seconds."

```
100     sub message_handle
        exec sql include sqlca
        exec sql begin declare section
            declare string buffer = 50
            declare integer seconds
        exec sql end declare section
            ...
        exec frs message :buffer
        exec frs sleep :seconds
            ...
        end sub
```

Array Variables

An array variable is referred to by the syntax:

:arrayname **(***subscripts)*

Syntax Notes:

- You must subscript the variable, because only scalar-valued elements (integers, reals, and character strings) are legal SQL values.
- When you declare the array, the Embedded SQL preprocessor does not parse the array bounds specification. Consequently, the Embedded SQL preprocessor will accept illegal bounds values. Also, when an array is referenced, the subscript is not parsed. The preprocessor confirms only the use of an array subscript with an array variable. You must ensure that the subscript is legal and that the correct number of indices is used.
- Arrays of null indicator variables used with structure assignments should not include subscripts when referenced.

In the following example, a variable is used as a subscript and need not be declared in the declaration section, as it is not parsed.

```
exec sql begin declare section
    declare string formnames(3)
 exec sql end declare section

data 'empform', 'deptform', 'helpform'
 declare integer i

for i = 1 to 3
    read formnames(i)
    exec frs forminit :formnames(i)
 next i
```

Record Variables

You can use a record variable in two different ways. First, you can use the record as a simple variable, implying that all its members are used. This would be appropriate in the Embedded SQL **select**, **fetch**, and **insert** statements. Second, you can use a member of a record to refer to a single element. Of course, this member must be a scalar value (integer, real or character string).

Using a Record as a Collection of Variables

The syntax for referring to a complete record is the same as referring to a simple variable:

:recordname

Syntax Notes:

- The *recordname* can refer to a main or nested record. It can be an element of an array of records. Any variable reference that denotes a record is acceptable. For example:

  ```
  :emprec                     ! A simple record
  :rec_array(i)               ! An element of an array of records
  :rec::minor2::minor3        ! A nested record at level 3
  ```

- To be used as a collection of variables, the final record in the reference must have no nested records, groups, or arrays. The preprocessor will enumerate all the members of the record. The members must have scalar values. The preprocessor generates code as though the program had listed each record member in the order in which it was declared.

The following example uses the employee.dcl file generated by DCLGEN to retrieve values into a record.

```
exec sql begin declare section
    exec sql include 'employee.dcl'
    ! see above for description
exec sql end declare section

exec sql select *                        &
    into :emprec                         &
    from employee                        &
    where eno = 123
```

The example above generates code as though the following statement had been issued instead:

```
exec sql select * &
     into :emprec::eno, :emprec::ename, :emprec::age, &
          :emprec::job, :emprec::sal, :emprec::dept &
    from employee &
    where eno = 123
```

The example below fetches the values associated with all the columns of a cursor into a record.

```
exec sql begin declare section
    exec sql include 'employee.dcl'
    ! see above for description
exec sql end declare section

exec sql declare empcsr cursor for &
    select *                       &
    from employee                  &
    order by ename
      ...

exec sql fetch empcsr into :emprec
```

The following example inserts values by looping through a locally declared array of records whose elements have been initialized:

```
exec sql begin declare section
    exec sql declare person table       &
    (pname char(30),                    &
     page integer1,                     &
     paddr varchar(50))
 record person_
    string  name = 30
        word    age
        string  addr = 50
end record

declare person_ person(10)
 declare word i

exec sql end declare section
...

for i=1 to 10
     exec sql insert into person &
        values (:person(i))
 next i
```

The **insert** statement in the above example generates code as though the following statement had been issued instead:

```
exec sql insert into person &
    values (:person(i)::name, :person(i)::age,
            :person(i)::addr)
```

Using a Record Member

The syntax Embedded SQL uses to refer to a record member is the same as in BASIC:

:record::member{::member}

Syntax Notes:

- The record member denoted by the above reference must be a scalar value (integer, real or character string). There can be any combination of arrays and records, but the last object referenced must be a scalar value. Thus, the following references are all legal:

```
:employee::sal              ! Member of a record
:person(3)::name            ! Member of an element of an array
:rec1::mem2::mem3::age      ! Deeply nested member
```

- All record components must be fully qualified when referenced. Elliptical references, such as references that omit group names, are not allowed.

The following example uses the record "emprec", similar to the record generated by DCLGEN, to put values into the form "empform".

```
exec sql begin declare section
    record emprec_
        long            idno
        string          ename = 20
        word            age
        string          hired = 25
        double          salary
        string          dept = 10
    end record
    declare emprec_ emprec

exec sql end declare section
    ...

exec frs putform empform &
    (eno = :emprec::idno, ename = :emprec::ename, &
     age = :emprec::age, hired = :emprec::hired, &
     sal = :emprec::salary, dept = :emprec::dept)
```

Using Indicator Variables

The syntax for referring to an *indicator* variable is the same as for a simple variable, except that an indicator variable is always associated with a host variable:

:host_variable:indicator_variable

or

:host_variable indicator :indicator_variable

Syntax Notes:

- The indicator variable can be a simple variable, an array element or a record member that yields a 2-byte integer (the **word** subtype). For example:

```
dcl word ind_var, ind_arr(5)
    :var_1:ind_var
    :var_2:ind_arr(2)
```

- If the host variable associated with the indicator variable is a record, the indicator variable should be an array of 2-byte integers. In this case the array should *not* be dereferenced with a subscript.
- When you use an indicator array, the first element of the array corresponds to the first member of the record, the second element with the second member, and so on. Indicator array elements generated by the preprocessor begin at subscript 1 and not at subscript 0.

The following example uses the employee.dcl file generated by DCLGEN, to retrieve values into a record and null values into the array "empind".

```
exec sql include sqlca

exec sql begin declare section
    exec sql include 'employee.dcl'
        ! see above for description
    declare word empind(10)

exec sql end declare section

exec sql select *                                &
     into :emprec:empind                         &
     from employee
```

The above example generates code as though the following statement had been issued:

```
exec sql select *                                                &
into :emprec::eno:empind(1), :emprec::ename:empind(2),           &
     :emprec::age:empind(3), :emprec::job:empind(4),             &
     :emprec::sal:empind(5), :emprec::dept:empind(6),            &
from employee
```

Note that there are three different types of *colon qualifiers*. The first colon indicates that a host variable is used. The second double-colon indicates that a structure member is used. The third colon is the indicator variable colon.

Data Type Conversion

A BASIC variable declaration must be compatible with the Ingres value it represents. Numeric Ingres values can be set by and retrieved into numeric variables, and Ingres character values can be set by and retrieved into string variables.

Data type conversion occurs automatically for different numeric types, as follows:

- From floating-point Ingres database column values into integer BASIC variables
- From integer to decimal
- From decimal to integer
- For different length character strings, such as from varying-length Ingres character fields, into static BASIC string variables

Ingres does not automatically convert between numeric and character types. You must use the Ingres type conversion functions, the Ingres **ascii** function, or a BASIC conversion procedure for this purpose.

The following table shows the default type compatibility for each Ingres data type. Note that some BASIC types do not match exactly and, consequently, can go through some runtime conversion.

Ingres and BASIC Data Type Compatibility

Ingres Type	BASIC Type
char(*N*)	string (dynamic)
char(*N*)	string (static with length clause of *N*)
varchar(*N*)	string (dynamic)
varchar(*N*)	string (static with length clause of *N*)
integer1	integer byte
smallint	integer word
integer	integer long
float4	real single
float	real double
date	string (dynamic)
date	string (static with length clause of 25)
money	real double

Ingres Type	BASIC Type
table_key	string (*dynamic*)
table_key	string (static with length clause of 8)
object_key	string (*dynamic*)
object_key	string (*static with length clause of 16*)
decimal	real double
long varchar	string (dynamic)

Runtime Numeric Type Conversion

The Ingres runtime system provides automatic data type conversion between numeric-type values in the database and forms system and numeric BASIC variables. The standard type conversion rules (according to standard VAX rules) are followed. For example, if you assign a **real** variable to an integer-valued field, the digits after the decimal point of the variable's value are truncated. Runtime errors are generated for overflow on conversion when assigning Ingres numeric values into BASIC variables. Overflow caused by assigning BASIC numeric variables into Ingres numeric objects is likely to result in inconsistent data, but does not by default generate a runtime error. Using the **-x** flag on the Ingres statement changes this behavior by generating errors at runtime.

The BASIC **decimal** data type is converted to **real double** using BASIC assignment statements generated by the preprocessor. Variables of **decimal** data type can be converted twice at runtime, depending on the final Ingres type being set or retrieved from. The standard BASIC arithmetic conversion rules hold for all these generated assignment statements, with a potential loss of precision. For further information, see The Decimal Data Type in this chapter.

The Ingres **money** type is represented as **real double**, an 8-byte floating-point value.

Runtime Character and Varchar Type Conversion

Automatic conversion occurs between Ingres character string values and BASIC string variables. There are four string-valued Ingres objects that can interact with string variables. These are:

- Ingres names, such as form and column names
- Database columns of type **character**
- Database columns of type **varchar**

- Form fields of type **character**
- Database columns of type **long varchar**

Several considerations apply when dealing with string conversions, both to and from Ingres.

The conversion of BASIC string variables used to represent Ingres names is simple: trailing blanks are truncated from the variables because the blanks make no sense in that context. For example, the string literals "empform " and "empform" refer to the same form.

The conversion of other Ingres objects is a bit more complicated. First, the storage of character data in Ingres differs according to whether the medium of storage is a database column of type **character**, a database column of type **varchar**, or a **character** form field. Ingres pads columns of type **character** with blanks to their declared length. Conversely, it does not add blanks to the data in columns of type **varchar** or **long varchar** in form fields.

Second, the BASIC convention is to blank-pad static character strings. For example, the character string "abc" can be stored in a BASIC static string variable of length 5 as the string "abc " followed by two blanks.

When retrieving character data from an Ingres database column or form field into a BASIC variable, take note of the following conventions:

- When character data is retrieved from Ingres into a BASIC static string variable and the variable is longer than the value being retrieved, the variable is padded with blanks. If the variable is shorter than the value being retrieved, the value is truncated. You should always ensure that the variable is at least as long as the column or field, in order to avoid truncation of data.
- When character data is retrieved into a BASIC dynamic string variable, the variable's new length will exactly match the length of the data retrieved. Ingres manipulates dynamic strings in exactly the same way as BASIC does, creating and modifying storage requirements as necessary. For example, when zero-length **varchar** data is retrieved into a BASIC dynamic string variable, storage will not be created for the string.

When inserting character data into an Ingres database column or form field from a BASIC variable, note the following conventions:

- When you insert data from a BASIC variable into a database column of type **character** and the column is longer than the variable, the column is padded with blanks. If the column is shorter than the variable, the data is truncated to the length of the column. When you insert data from a BASIC variable into a database column of type **varchar** or **long varchar** and the column is longer than the variable, no padding of the column takes place. Furthermore, by default, all trailing blanks in the data are truncated before the data is inserted into the **varchar** column. For example, when a string "abc" stored in a BASIC static string variable of length 5 as "abc " (see above) is inserted into the **varchar** column, the two trailing blanks are removed and only the string "abc" is stored in the database column. To retain such trailing blanks, use the Ingres **notrim** function. It has the following syntax:

  notrim(:*stringvar*)

  where *stringvar* is a character string variable. An example demonstrating this feature follows later. If the **varchar** column is shorter than the variable, the data is truncated to the length of the column When you insert data from a BASIC variable into a **character** form field and the field is longer than the variable, no padding of the field takes place. In addition, all trailing blanks in the data are truncated before the data is inserted into the field. If the field is shorter than the data (even after all trailing blanks have been truncated), the data is truncated to the length of the field.

  You cannot use zero-length or uninitialized BASIC dynamic strings in **insert** or **update** statements. This is because an uninitialized dynamic string has no storage allocated for it and Ingres treats it as a non-existent variable.

  When comparing character data in an Ingres database column with character data in a BASIC variable, note the following convention:

- When comparing data in **character** or **varchar** database columns with data in a character variable, all trailing blanks are ignored. Initial and embedded blanks are significant.

Note: As described above, the conversion of character string data between Ingres objects and BASIC variables often involves the trimming or padding of trailing blanks, with resultant change to the data. If trailing blanks have significance in your application, give careful consideration to the effect of any data conversion. For a complete description of the significance of blanks in string comparisons, see the ***SQL Reference Guide***.

The Ingres **date** data type is represented as a 25-byte string.

The program fragment in the example demonstrates the **notrim** function and the truncation rules explained above.

```
exec sql include sqlca

exec sql begin declare section
    exec sql declare textchar table                      &
         (row integer,                                   &
          data varchar(10)) ! Note the varchar type
    declare word                 row
    common string                sdata = 7       ! static string
    declare string               ddata           ! dynamic string
exec sql end declare section

sdata = 'abc   '  ! Holds "abc " (with 4 blanks)
 ddata = 'abc'    ! Holds "abc"
! This insert adds string "abc" (blanks truncated)
 exec sql insert into textchar values (1, :sdata)

! This insert adds string "abc" (never had blanks)
 exec sql insert into textchar values (2, :ddata)

! This insert adds string "abc ", with tailing blanks
! left intact by using the notrim function.
 exec sql insert into textchar values (3, notrim(:sdata))

! This select retrieves rows #1 and #2, because trailing
! blanks were suppressed when those rows were inserted.
 exec sql select row into :row from textchar
        where length(data) = 3
exec sql begin
        print 'Row found =', row
exec sql end
! This select retrieves row #3, because the notrim
! function left trailing blanks in the "sdata"
! variable in the last insert statement.
 exec sql select row into :row from textchar
        where length(data) = 7
exec sql begin
        print 'row found =', row
exec sql end
```

The SQL Communications Area

This section describes the SQL Communications Area (SQLCA) as implemented in BASIC.

The Include SQLCA Statement

You should issue the **include sqlca** statement in your main program module as well as in each subprogram of your BASIC file that includes Embedded SQL statements. If the file is made up of one main program and a few subprograms, **include sqlca** should be the first Embedded SQL statement in each of the program units. For example:

```
10  ! main program
    exec sql include sqlca
    . . .
    end ! main

100 sub emp_sub
    exec sql include sqlca
    . . .
    end sub ! Emp_sub

200 function integer emp_func
    exec sql include sqlca
    . . .
    end func ! Emp_Func
```

The **include sqlca** statement instructs the preprocessor to generate code to call Ingres runtime libraries. It generates a BASIC **%include** statement to make all the calls generated by the preprocessor acceptable to the compiler. The **include sqlca** statement also generates a BASIC **%include** directive to define the SQLCA (SQL Communications Area) **common** block, which is used for error handling.

Whether or not you intend to use the SQLCA for error handling, you must issue an **include sqlca** statement in each program unit containing Embedded SQL statements. If you do not, the BASIC compiler may complain about undeclared functions. Furthermore, the program will abort at runtime because program memory will be overwritten. This occurs because, with no explicit declaration of the SQLCA using the **include sqlca** statement, the BASIC compiler implicitly declares all references (including preprocessor-generated references) to the SQLCA as the default data type (the default set by the BASIC environment or by the system). Therefore, to help detect runtime errors due to missing **include sqlca** statements, you may want to use the qualifier **type_default=explicit** with the BASIC compiler command. By doing so, you can ensure that the compiler generates a warning upon encountering a reference to an undeclared SQLCA.

Contents of the SQLCA

One of the results of issuing the **include sqlca** statement is the declaration of the SQLCA (SQL Communications Area), which you can use for error handling in the context of database statements. As mentioned above, you should issue the statement in your main program and in each subprogram that contains Embedded SQL statements. The declaration for the SQLCA is:

```
common (sqlca) string sqlcaid = 8,  &
               long   sqlcabc,       &
               long   sqlcode,       &
               word   sqlerrml,      &
               string sqlerrmc = 70, &
               string sqlerrp = 8,   &
               long   sqlerrd(5),     &
               string sqlwarn0 = 1,  &
               string sqlwarn1 = 1,  &
               string sqlwarn2 = 1,  &
               string sqlwarn3 = 1,  &
               string sqlwarn4 = 1,  &
               string sqlwarn5 = 1,  &
               string sqlwarn6 = 1,  &
               string sqlwarn7 = 1,  &
               string sqlext = 8
```

Note that the error diagnostic array, **sqlerrd**, is declared with 5 elements. This is because the BASIC compiler implicitly inserts element number zero before the declared array, so that there are really 6 array elements, as described in the *SQL Reference Guide*. A later section discusses the significance of **sqlerrd** for determining the number of rows affected by the last SQL statement.

The SQLCA is initialized at load-time. The fields **sqlcaid** and **sqlabc** are initialized to the string "SQLCA " and the constant 136, respectively.

Note that the preprocessor is not aware of the SQLCA declaration. Therefore, you cannot use SQLCA fields in an Embedded SQL statement. For example, the following statement, attempting to insert the error code **sqlcode** into a table, would generate an error:

```
! This statement is illegal
exec sql insert into employee (eno) &
 values (:sqlcode)
```

All modules written in BASIC and other Embedded SQL languages share the same SQLCA.

Using the SQLCA for Error Handling

Error handling with the SQLCA can be done implicitly by using **whenever** statements or explicitly by checking the contents of the SQLCA fields **sqlcode**, **sqlerrd(2)**, and **sqlwarn0**.

Error Handling with the Whenever Statement

The syntax of the **whenever** statement is as follows:

```
exec sql whenever condition action
```

condition is **dbevent**, **sqlwarning**, **sqlerror**, **sqlmessage**, or **not found**. *action* is **continue**, **stop**, **goto** a label or a line number, or **call** a BASIC subroutine. For a detailed description of this statement, see the *SQL Reference Guide*.

The subroutine names qualifying the call action must be legal BASIC identifiers beginning with an alphabetic character or an underscore. If the subroutine name is an Embedded SQL reserved word, specify it in quotes. Note that the label or line number targeted by the **goto** action must be in the scope of all subsequent Embedded SQL statements until another **whenever** statement is encountered for the same action. This is necessary because the preprocessor may generate the BASIC statement:

```
if (condition) then
    goto label
end if
```

after an Embedded SQL statement. If the label is outside the scope of the current Embedded SQL statement, the BASIC compiler will generate an error.

The same scope rules apply to subroutine names used with the **call** action. However, the reserved subroutine name **sqlprint**, which prints errors or database procedure messages and then continues, is always in the scope of the program.

When a **whenever** statement specifies a **call** as the action, the target subroutine is called and, after its execution, control returns to the statement following the statement that caused the subroutine to be called. Consequently, after handling the **whenever** condition in the called subroutine, you may want to take some action, instead of merely returning from the BASIC subroutine.

The following example demonstrates use of the **whenever** statements in the context of printing some values from the Employee table. The comments do not relate to the program but to the use of error handling.

```
10 ! Main error handling program
        exec sql include sqlca
        exec sql begin declare section
                        declare integer eno
                        declare string ename
                        declare byte eage
        exec sql end declare section
        exec sql declare empcsr cursor for                &
                 select idno, name, age                   &
                 from employee
! An error when opening the personnel database will
! cause the error to be printed and the program to abort
    exec sql whenever sqlerror stop
    exec sql connect personnel
! Errors from here on will cause the program to clean up
    exec sql whenever sqlerror call clean_up
    exec sql open empcsr
    print 'Some values from the "employee" table'
! When no more rows are fetched, close the csr
    exec sql whenever not found goto Close_Csr
! The last executable Embedded SQL statement was an
! OPEN, so we know that the value of "sqlcode" cannot
! be SQLERROR or NOT FOUND.

    while (sqlcode = 0)
         exec sql fetch empcsr &
                into :eno, :ename, :eage
! This "print" does not execute after the previous
! FETCH returns the NOT FOUND condition.

        print eno, ename, eage
    next
! From this point in the file onwards, ignore all
! errors. Also turn off the NOT FOUND condition,
! for consistency.

        Close_Csr: EXEC SQL CLOSE empcsr
        exec sql disconnect

end ! Db_Test
! Clean_Up: Error handling subroutine (print error and disconnect).

20 sub Clean_Up
    exec sql include sqlca
    exec sql begin declare section
        declare string errmsg
    exec sql end declare section
    exec sql inquire_sql(:errmsg = errortext)
    print 'aborting because of error'
    print errmsg
    exec sql disconnect
    ! Do not return to Db_Test
    stop

end sub ! Clean_Up
```

The Whenever Goto Action in Embedded SQL Blocks

An Embedded SQL block-structured statement is a statement delimited by the **begin** and **end** clauses. For example, the **select** loop and the **unloadtable** loop are both block-structured statements. These statements can only be terminated by the methods specified for the particular statement in the *SQL Reference Guide*. For example, the **select** loop is terminated either when all the rows in the database result table have been processed or by an **endselect** statement, and the **unloadtable** loop is terminated either when all the rows in the forms table field have been processed or by an **endloop** statement.

Therefore, if you use a **whenever** statement with the **goto** action in an SQL block, you must avoid going to a label outside the block. Such a **goto** would cause the block to be terminated without issuing the runtime calls necessary to clean up the information that controls the loop. (For the same reason, you must not issue a BASIC **exit** or **goto** statement that causes control to leave or enter an SQL block.) The target label of the **whenever goto** statement should be a label in the block. If, however, it is a label for a block of code that cleanly exits the program, the above precautions need not be taken.

The above information does not apply to error handling for database statements issued outside an SQL block nor to explicit hard-coded error handling. For an example of hard-coded error handling, see The Table Editor Table Field Application in this chapter.

Explicit Error Handling

The program can also handle errors by inspecting values of the SQLCA at various points. For further details, see the *SQL Reference Guide*.

The following example is functionally the same as the previous example, except that the error handling is hard-coded in BASIC statements.

```
10 ! Main error handling program
        exec sql include sqlca
        exec sql begin declare section
                declare integer eno
                declare string ename
                declare byte eage
        exec sql end declare section
        exec sql declare empcsr cursor for &
                select idno, name, age from employee
        ! Exit if database cannot be opened
        exec sql connect personnel
        if (sqlcode < 0) then
                print 'Cannot access database'
                stop
        end if
! Error if cannot open cursor
        exec sql open empcsr
        if (sqlcode < 0) then
                call Clean_Up('OPEN "empcsr"')
        end if
        print 'Some values from the "employee" table'
! The last executable Embedded SQL statement was an OPEN, so we know
! that the value of "sqlcode" cannot be SQLERROR or NOT FOUND
! The following loop is broken by NOT FOUND condition 100) or an error
        while (sqlcode = 0)
             exec sql fetch empcsr &
                into :eno, :ename, :eage
        if (sqlcode < 0) then
            call Clean_Up('FETCH "empcsr"')

! Do not print the last values twice
        else
            if (sqlcode <> 100) then
                print eno, ename, eage
            end if
        end if
    next
    exec sql close empcsr
    exec sql disconnect

end
! Clean_Up: Error handling subroutine (print error and disconnect).

20 sub Clean_Up(string reason)

        exec sql include sqlca
        exec sql begin declare section
            declare string errmsg
        exec sql end declare section
        print 'aborting because of error in', reason
        exec sql inquire_sql (:errmsg = errortext)
        print errmsg

        exec sql disconnect
! Do not return to main program
     stop

end sub ! clean_up
```

Determining the Number of Affected Rows

The SQLCA variable **sqlerrd(2)** indicates how many rows were affected by the last row-affecting statement. Note that this variable is referenced by **sqlerrd(2)** rather than **sqlerrd(3)** as in other languages, because BASIC allocates **sqlerrd** elements 0 through 5. The following program fragment, which deletes all employees whose employee numbers are greater than a given number, demonstrates how to use **sqlerrd**:

```
sub delete_rows(integer lower_bound_num)

    exec sql include sqlca
    exec sql begin declare section
        declare integer low_eno
    exec sql end declare section
! Use Embedded SQL variable in DELETE statement
    low_eno = lower_bound_num
    exec sql delete from employee &
        where eno :low_eno
! Print the number of employees deleted
    print sqlerrd(2), 'row(s) were deleted.'
 end sub ! Delete_Rows
```

Using the SQLSTATE Variable

You can use the **SQLSTATE** variable in an ESQL/BASIC program to return status information about the last SQL statement that was executed. **SQLSTATE** must be declared in a declaration section. Also, it is valid across all sessions, so you only need to declare one **SQLSTATE** per application.

To declare this variable, use:

```
character5 SQLSTATE
```

or :

```
character5 SQLSTA
```

Dynamic Programming for BASIC

Ingres provides Dynamic SQL and Dynamic FRS to allow you to write generic programs. Dynamic SQL allows a program to build and execute SQL statements at runtime. For example, an application can include an expert mode in which the runtime user can type in select queries and browse the results at the terminal. Dynamic FRS allows a program to interact with any form at runtime. For example, an application can load in any form, allowing the runtime user to retrieve new data from the form and insert it into the database.

The Dynamic SQL and Dynamic FRS statements are described in the *SQL Reference Guide* and the *Forms-based Application Development Tools User Guide,* respectively. This section discusses the BASIC-dependent issues of Dynamic programming. For a complete example of using Dynamic SQL to write an SQL Terminal Monitor application, see The SQL Terminal Monitor Application in this chapter. For an example of using both Dynamic SQL and Dynamic FRS to browse and update a database using any form, see A Dynamic SQL/Forms Database Browser in this chapter.

The SQLDA Record

The SQLDA (SQL Descriptor Area) passes type and size information about an SQL statement, an Ingres form, or a table field between Ingres and your program.

In order to use the SQLDA, you should issue the **include sqlda** statement at the proper scope of the source file, from where the SQLDA will be referenced. The **include sqlda** statement generates a BASIC **include** directive to a file that defines the SQLDA record type. The file does *not* declare an SQLDA record variable; your program must declare a variable of the specified type. You can also code this record declaration directly instead of using the **include sqlda** statement. When coding the declaration yourself, you can choose any name for the record type.

The definition of the SQLDA (as specified in the **include** file) is:

```
!
! IISQ_MAX_COLS - Maximum number of columns returned from Ingres
!
  declare word constant IISQ_MAX_COLS = 300
!
! IISQLDA - SQLDA with maximum number of entries for
! variables.
!
  record IISQLDA
        string sqldaid = 8
        long sqldabc
        word sqln
        word sqld
        group sqlvar(IISQ_MAX_COLS)
                word sqltype
                word sqllen
                long sqldata ! Address of any type
                long sqlind  ! Address of 2-byte integer
                group sqlname
                        word     sqlnamel
                        string   sqlnamec = 34
                end group sqlname
        end group sqlvar
  end record IISQLDA
!
! Type Codes
!
  declare integer constant IISQ_DTE_TYPE = 3
! Date - Out
  declare integer constant IISQ_MNY_TYPE = 5
! Money - Out
  declare integer constant IISQ_DEC_TYPE =10
! Decimal - Out
 declare integer constant IISQ_CHA_TYPE = 20
! Char - In/Out
  declare integer constant IISQ_VCH_TYPE = 21
! Varchar - In/Out
  declare integer constant IISQ_INT_TYPE = 30
! Integer - In/Out
  declare integer constant IISQ_FLT_TYPE = 31
! Float - In/Out
  declare integer constant IISQ_TBL_TYPE = 52
! Table field - Out
  declare integer constant IISQ_DTE_LEN = 25
! Date length
!
! Dynamic allocation sizes - When allocating an
! SQLDA for N results use:
! IISQDA_HEAD_SIZE + (N * IISQDA_VAR_SIZE)
!
  declare integer constant IISQDA_HEAD_SIZE = 16
  declare integer constant IISQDA_VAR_SIZE = 48
```

Record Definition and Usage Notes:

- The record type definition of the SQLDA is called IISQLDA. This is done so that an SQLDA variable can be called "SQLDA" without causing a compile-time BASIC conflict. You are not required to call your SQLDA record variable "SQLDA".

- The **sqlvar** array is an array of IISQ_MAX_COLS (300) elements. If you declare a record variable of type IISQLDA, then the program will have a variable with IISQ_MAX_COLS **sqlvar** elements.
- Note that the **sqlvar** array begins at subscript 0 because of the BASIC default of arrays being zero-based. Because this array begins at subscript zero, it implies that relevant result variables are described by the elements 0 through **sqld-1**, rather than 1 through **sqld**.
- The **sqldata** and **sqlind** group members are declared as **long** integers. These must be set to the addresses of other result variables before using the SQLDA to retrieve or set Ingres data in the database or form. You can use the BASIC **loc** function to assign addresses.
- If you declare your own SQLDA record type and variable, you must confirm that the record layout is identical to that of the IISQLDA record type, although you can allocate a different number of **sqlvar** array elements.
- The nested group **sqlname** is a varying length character string consisting of a length and data area. The **sqlnamec** member contains the name of a result field or column after a **describe** (or **prepare into**) statement. The length of the name is specified by **sqlnamel**. The characters in the **sqlnamec** field are blank padded. You can also set the **sqlname** group by a program using Dynamic FRS. The program is not required to pad **sqlnamec** with blanks. For more information, see Setting SQLNAME for Dynamic FRS in this chapter.
- The list of type codes represents the types that will be returned by the **describe** statement, and the types used by the program when retrieving or setting data with an SQLDA. The type code IISQ_TBL_TYPE indicates a table field and is set by the FRS when describing a form that contains a table field.

Declaring an SQLDA Variable

Once the SQLDA record definition has been included (or hard coded) the program can declare an SQLDA variable. This record variable must be declared outside of an Embedded SQL **declare section**, as the preprocessor does not understand the special meaning of the SQLDA record or the IISQLDA record type. When you use the variable in the context of a Dynamic SQL or Dynamic FRS statement, the preprocessor accepts any object name, and assumes that the variable refers to a legally declared SQLDA record variable. If a program requires an SQLDA record variable with the same number of **sqlvar** variables as in the IISQLDA record type, then it can accomplish this as in:

```
exec sql include sqlda ! Defines record type
    declare iisqlda sqlda ! Declares sqlda record variable
    sqlda::sqln = iisq_max_cols ! set the size
    ...

    exec sql describe s1 into :sqlda
```

Normally the same SQLDA can be used across various BASIC subroutines and external procedures. In these cases you can declare the SQLDA using any one of the BASIC storage classes, such as **common** or **external**. For example the above declaration could also have been:

```
exec sql include sqlda
 common (sqlda_area) iisqlda sqlda
    ! declares global sqlda
```

At other times you may want to dynamically allocate your SQLDA record variable out of another storage area. In that case you can use various BASIC **map** statements to define the position of the SQLDA in the storage area. However, you must confirm that the SQLDA record variable being used is a valid SQLDA, with storage allocated to it.

If a program requires an SQLDA variable with a different number of **sqlvar** variables (not IISQ_MAX_COLS), the program can then define its own record type and declare its own variable. For example:

```
record MY_SQLDA ! Record type with 50 elements
        string  myid = 8
        long    mybc
        word    myvars
        word    mycols
        group   vararray(50)
            word vartype
            word varlen
            long vardata
            long varind
            group varname
                word varnamel
                string varnamec = 34
            end group varname
        end group vararray
 end record MY_SQLDA

declare MY_SQLDA myda           ! SQLDA variable
...

myda::myvars = 50               ! Set the size
...

exec sql describe s1 into :myda
```

In the above record type definition, the names of the record members are not the same as those of the IISQLDA record, but their layout is identical.

Using the SQLVAR

The *SQL Reference Guide* discusses the legal values of the **sqlvar** array. The **describe** and **prepare into** statements set the type, length, and name information of the SQLDA. This information refers to the result columns of a prepared **select** statement, the fields of a form, or the columns of a table field. When the program uses the SQLDA to retrieve or set Ingres data, it must assign type and length information, which now refers to the variables being pointed at by the SQLDA.

BASIC Variable Type Codes

The type codes listed above are the types that describe Ingres result fields or columns. For example, the SQL types **date**, **decimal**, and **money** do not describe a program variable, but rather result data types that are compatible with BASIC character string and numeric types. IISQ_LVCH_TYPE is SQL only character compatible too. When these types are returned by the **describe** statement, the type code must be a change to a compatible BASIC or ESQL/BASIC type.

The following table describes the type codes to use with BASIC variables that will be pointed at by the **sqldata** pointers:

The SQLDA Type Codes

BASIC Type	SQLType Code (sqltype)	SQL Length (sqllen)
byte	IISQ_INT_TYPE	1
word	IISQ_INT_TYPE	2
long	IISQ_INT_TYPE	4
real	IISQ_FLT_TYPE	4
double	IISQ_FLT_TYPE	8
string = LEN	IISQ_CHA_TYPE	LEN
string	IISQ_DEC_TYPE	10

As described in the section BASIC Variables and Data Types, all other types are compatible with the above BASIC data types. For example, you can retrieve an SQL **date** into a **string** variable, while you can retrieve **money** into a **double** variable.

Nullable data types (those variables that are associated with a null indicator) are specified by assigning the negative of the type code to **sqltype**. If the type is negative, you must point at a null indicator by the **sqlind** variable. The type of the null indicator must be a 2-byte integer, a **word** variable. For information on how to declare and use a null indicator in BASIC, see BASIC Variables and Data Types in this chapter.

Character data and the SQLDA have the exact same rules as character data in regular Embedded SQL statements. Because string lengths must be assigned to **sqllen** before using the SQLDA, you cannot point at BASIC dynamic string variables (those declared without a length) if they have not yet been assigned any storage. For more details on character string processing in SQL, see BASIC Variables and Data Types in this chapter.

Pointing at BASIC Variables

In order to fill an element of the **sqlvar** array, you must set the type information and assign a valid address to **sqldata**. The address must be that of a legally declared variable. If the element is nullable then the corresponding **sqlind** member must point at a legally declared null indicator variable.

Because both the **sqldata** and **sqlind** members of the **sqlvar** group are declared as long integers, you must assign integer values to them. This requires the use of the BASIC **loc** function.

For example, the following program fragment sets the type information of and points at a 4-byte integer variable, an 8-byte nullable floating-point variable, and an **sqllen**-specified character sub-string. This example demonstrates how a program can maintain a pool of available variables, such as large arrays of a few different typed variables and a large string space. When a variable is chosen from the pool the next available spot is incremented:

```
 exec sql include sqlda
 declare iisqlda sqlda
...

! Numeric and string 'pool' declarations
 declare word            constant MAX_POOL = 50
 declare word           ind_store(MAX_POOL)      ! Indicators
 declare word           current_ind
 declare long           int4_store(MAX_POOL)     ! Integers
 declare word           current_int
 declare double flt8_store(MAX_POOL)             ! Floats
 declare word           current_flt
 declare string char_store(3000) = 1             ! String buffer
 declare word           current_chr
 declare word           need_len
...

!
! Note that if SQLD is set to 3 we use SQLVAR elements ! 0 through 2
!
 sqlda::sqlvar(0)::sqltype = IISQ_INT_TYPE       ! 4-byte integer
sqlda::sqlvar(0)::sqllen = 4
sqlda::sqlvar(0)::sqldata = loc(int4_store(current_int))
 sqlda::sqlvar(0)::sqlind = 0
current_int = current_int + 1 ! Update integer pool
sqlda::sqlvar(1)::sqltype = -IISQ_FLT_TYPE       ! 8-byte null float
sqlda::sqlvar(1)::sqllen = 8
sqlda::sqlvar(1)::sqldata = loc(float8_store(current_flt))
 sqlda::sqlvar(1)::sqlind = loc(ind_store(current_ind))
 current_flt = current_flt + 1 ! Update float and
current_ind = current_ind + 1 ! indicator pool
!
! SQLLEN has been assigned by DESCRIBE to be the length ! of a specific
! result column. This length is used to pick off a sub-string out of
! a large string space.
!

need_len = sqlda::sqlvar(2)::sqllen
sqlda::sqlvar(2)::sqltype = IISQ_CHA_TYPE
sqlda::sqlvar(2)::sqldata = loc(char_store(current_chr))
 sqlda::sqlvar(2)::sqlind = 0
current_chr = current_chr + need_len ! Update char pool
...
```

Of course, in the above example, verification of enough pool storage must be made before each cell of the different arrays is referenced in order to prevent **sqldata** and **sqlind** from pointing at undefined storage. For demonstrations of this method, see The SQL Terminal Monitor Application and A Dynamic SQL/Forms Database Browser in this chapter.

The IISQ_HDLR_TYPE is a host language type that is used for transmitting data to and from Ingres. Because it is not an Ingres data type, it will never be returned as a data type from the **describe** statement.

Setting SQLNAME for Dynamic FRS

Using the **sqlvar** with Dynamic FRS statements requires a few extra steps that relate to differences between Dynamic FRS and Dynamic SQL. These differences are described in the *SQL Reference Guide*.

When using the SQLDA in a forms input or output **using** clause, the value of **sqlname** must be set to a valid field or column name. If this name was set by a previous **describe** statement, it must be retained or reset by the program. If the name refers to a hidden column or table field, then your program must set it directly. If your program sets **sqlname** directly, it must also set **sqlnamel** and **sqlnamec**.

The name portion need not be padded with blanks. For example, a dynamically named table field has been described, and the application always initializes any table field with a hidden 6-byte character column called "rowid". The code used to retrieve a row from the table field including the hidden column and **_state** variable would have to construct the two named columns:

```
declare string rowid = 6
declare long rowstate
...
 exec frs describe table :formname :tablename INTO :sqlda
...

! BASIC is zero-based so save before incrementing
col_num = sqlda::sqld
sqlda::sqld = sqlda::sqld + 1
! Set up to retrieve rowid
sqlda::sqlvar(col_num)::sqltype = IISQ_CHA_TYPE
sqlda::sqlvar(col_num)::sqllen = 6
sqlda::sqlvar(col_num)::sqldata = loc(rowid)
 sqlda::sqlvar(col_num)::sqlind = 0
sqlda::sqlvar(col_num)::sqlname::sqlnamel = 5
sqlda::sqlvar(col_num)::sqlname::sqlnamec = 'rowid'

col_num = sqlda::sqld
sqlda::sqld = sqlda::sqld + 1
! Set up to retrieve _STATE
sqlda::sqlvar(col_num)::sqltype = IISQ_INT_TYPE
sqlda::sqlvar(col_num)::sqllen = 4
sqlda::sqlvar(col_num)::sqldata = loc(rowstate)
 sqlda::sqlvar(col_num)::sqlind = 0
sqlda::sqlvar(col_num)::sqlname::sqlnamel = 6
sqlda::sqlvar(col_num)::sqlname::sqlnamec = '_state'
...
```

```
exec frs getrow :formname :tablename using descriptor :sqlda
```

Advanced Processing

This section describes user-defined handlers. It includes information about user-defined error, dbevent, and message handlers as well as data handlers for large objects.

User-Defined Error, DBevent, and Message Handlers

You can use user-defined handlers to capture errors, messages, or events during the processing of a database statement. Use these handlers instead of the **sql whenever** statements with the SQLCA when you want to do the following:

- Capture more than one error message on a single database statement.
- Capture more than one message from database procedures fired by rules.
- Trap errors, events, and messages as the DBMS raises them. If an event is raised when an error occurs during query execution, the WHENEVER mechanism detects only the error and defers acting on the event until the next database statement is executed.

User-defined handlers offer you flexibility. If, for example, you want to trap an error, you can code a user-defined handler to issue an **inquire_sql** to get the error number and error text of the current error. You can then switch sessions and log the error to a table in another session; however, you must switch back to the session from which the handler was called before returning from the handler. When the user handler returns, the original statement continues executing. User code in the handler cannot issue database statements for the session from which the handler was called.

The handler must be declared to return an integer. However, the preprocessor ignores the return value.

Syntax Notes:

The following syntax describes the three types of handlers:

```
exec sql set_sql (errorhandler = error_routine|0)
 exec sql set_sql (dbeventhandler = event_routine|0)
 exec sql set_sql (messagehandler = message_routine|0)
```

- Errorhandler, dbeventhandler, and messagehandler denote a user-defined handler to capture errors, events, and database messages respectively, as follows:
  - error_routine is the name of the function the Ingres runtime system calls when an error occurs.

- event_routine is the name of the function the Ingres runtime system calls when an event is raised. message_routine is the name of the function the Ingres runtime system calls whenever a database procedure generates a message.

  Errors that occur in the error handler itself do not cause the error handler to be re-invoked. You must use **inquire_sql** to handle or trap any errors that may occur in the handler.

- Unlike regular variables, the handler must not be declared in an ESQL declare section; therefore, do not use a colon before the handler argument. (However, you must declare the handler to the compiler.)
- If you specify a zero (0) instead of a name, the zero will unset the handler.

User-defined handlers are also described in the *SQL Reference Guide*.

Declaring and Defining User-Defined Handlers

The following example shows how to declare a handler for use in the set_sql errorhandler statement for ESQL/BASIC:

```
! Main program

program error_trap
    exec sql include sqlca
    external integer error_func          ! declare error handler
    exec sql connect dbname
    exec sql set_sql (errorhandler = error_func)
!
!   esql will generate
!   call iilqshsethandler (1, error_func)
!
    . . .
 end program

function integer error_func()
exec sql include sqlca

exec sql begin declare section
        declare integer errnum
exec sql end declare section
        exec sql inquire_sql (:errnum = errorno)
        print 'error number is ' + str$(errnum)
 end function
```

Sample Programs

The programs in this section are examples of how to declare and use user-defined data handlers in an ESQL/BASIC program. There are examples of a handler program, a Put Handler program, a Get Handler program, and a dynamic SQL handler program.

Handler Program

This program inserts a row into the book table using the data handler Put_Handler to transmit the value of column chapter_text from a text file to the database. Then it selects the column chapter_text from the table book using the data handler Get_Handler to process each row returned.

```
!main program
!*****************

    program handler
        exec sql include sqlca
! Do not declare the data handlers nor the data handler
! argument to the ESQL preprocessor
        external integer Put_Handler
        external integer Get_Handler
        record hdlr_arg
                string  argstr
                integer argint
        end record hdlr_arg
        declare hdlr_arg hdlarg
! Null indicator for data handler must be declared to ESQL
        exec sql begin declare section
                word indvar
        exec sql end declare section
! INSERT a long varchar value chapter_text into the
! table book using the data handler put_handler. The
! argument passed to the data handler the record hdlarg.

    . . .

        exec sql insert into book (chapter_name, chapter_text) &
                values (5, 'One Dark and Stormy Night',
               data handler(Put_Handler(hdlarg)))

! SELECT the long varchar column chapter_text from
! The data handler (get_handler) will be invoked for
! each non-null value of column chapter_text retrieved.
! For null values the indicator variable will be set
! to "-1" and the data handler will not be called.

    ...
    exec sql select chapter)text into           &
                data handler(get_handler(hdlarg)):indvar from book
    exec sql begin
            process row...
    exec sql end
    ...

    end program
```

Put Handler

This user-defined handler shows how an application can use the put data handler to enter a chapter of a book from a text file into a database.

```
! Put_handler
! ************

100 function integer Put_handler(hdlr_arg info)

    record  hdlr_arg
        string argstr
        integer argint
    end record hdlr_arg
    exec sql begin declare section;
        declare sting    segbuf
        declare integer  seglen
        declare integer  datend
    exec sql end declare sections
    process information passed in via the info record
    open file.....

    datend = 0
        while not end-of-file
            read segment from file into segbuf...

            if (end-of-file) then
                datend = 1
            end if
        exec sql put data (segment = :segbuf,           &
    segmentlength = :seglen, dataend = :datend)

    next
    ...
    close file...
    set info record to return appropriate values...
    ..
    Put_handler = 0
end function
```

Get Handler

This user-defined data handler shows how an application can use the get data handler to enter a chapter of a book from a text file into a database.

```
! Get_Handler
!   *************
200 integer function Get_Handler(hdlr_arg info)
        record hdlr_arg
            string     argstr
            integer    argint
        end record hdlr_arg
        exec sql begin declare section
            declare string              segbuf
            declare integer seglen
            declare integer datend
            declare integer      maxlen
        exec sql end declare section

        ...
        process information passed in via the
             info record...
        open file....

        datend = 0

        while (datend = 0)
            exec sql get data (:segbuf = segment,&
             :seglen = segmentlenght, & :datend = dataend) &
            with maxlength = :maxlen
        write segment to file

        next
        ...
        set info record to return appropriate values...
        ...

        Get_Handler = 0

end function
```

User-Defined Data Handlers for Large Objects

Use the following definitions when you code user-defined data handlers for large objects in Dynamic SQL programs that use the **exec sql include sqlda** statement:

```
    declare integer constant IISQ_LVCH_TYPE = 22
    declare integer constant IISQ_HDLR_TYPE = 22
record IISQLHDLR
        long    sqlarg
        long    sqlhdlr
end record IISQLHDLR
```

Dynamic SQL Handler Program

The following is an example of a dynamic SQL handler program:

```
! main program using SQLDA
! **************************

  program dynamic_hdlr

    exec sql include sqlca
    exec sql include sqlda
! Do not declare the data handlers nor the data handler
! argument to the ESQL preprocessor

  external   integerPut_Handler
  external   integerGet_Handler
! Declare argument to be passed to data handler

  record     hdlr_arg
      string     argstr
      integer    argint
  end record  hdlr_arg
! Declare SQLDA and IISQLHDLR

  common (sqlda_area) IISQLDA sqlda
  common (result_area) num_store        nums(IISQ_MAX_COLS), &
                                        char_store chars

  declare IISQLHDLR    data_handler
  declare hdlr_arg        hdlarg
  declare       integer   base_type
! Declare null indicator to ESQL

  exec sql begin declare section
      word              indvar
      string(100)       stmt_buf
      integer           i
  exec sql end declare section
    ...
! Set the IISQLHDLR structure with the appropriate
! data handler and data handler argument.

  data_handler::sqlhdlr = loc(Get_Handler)
  data_handler::sqlarg  = loc(hdlarg)

! Describe the statment into the SQLDA
  stmt_buf = 'select * from book'.
  exec sql prepare stmt from :stmt_buf
  exec sql describe stmt into sqlda
  ...

! Determine the base_type of the SQLDATA variables
    while ( i < sqlda::sqld)
        i = i + 1
        if (sqlda::sqlvar(i)::sqltype > 0) then
            base_type = sqlda::sqlvar(I)::sqltype
        else
             base_type = -sqlda::sqlvar(i)::sqltype
        end if
! Set the sqltype, sqldata and sqlind for each column
! The long varchar column chapter_text will be set to
! use a data handler
      if (base_type = IISQ_LVCH_TYPE) then
            sqlda::sqlvar(i)::sqltype = IISQ_HDLR_TYPE
            sqlda::sqlvar(i)::sqldata = loc(data_handler)
            sqlda::sqlvar(i)::sqlind = loc(indvar)
```

```
      else
    .   .   .
      end if
    next
! The Data handler (Get_Handler) will be invoked for
! each non-null value of column chapter_text retrieved.
! For null values the indicator variable will be
! set to "-1" and the data handler will not be called

  ...

    exec sql execute immediate :stmt_buf using :SQLDA
    exec sql begin
         process row...
    exec sql end
    ...

end program
```

Preprocessor Operation

This section describes the operation of the Embedded SQL preprocessor for BASIC and the steps required to create, compile, and link an Embedded SQL program.

Command Line Operations

The following sections describe how to turn an embedded ESQL/BASIC source program into an executable program. These sections include commands that preprocess, compile, and link a program.

The Embedded SQL Preprocessor Command

The BASIC preprocessor is invoked by the following command line:

esqlb {*flags*} {*filename*}

where *flags* are

Flag	Description
-d	Adds debugging information to the runtime database error messages generated by Embedded SQL. The source file name, line number, and statement in error will be displayed with the error message.
-f[*filename*]	Writes preprocessor output to the named file. If no filename is specified, the output is sent to standard output, one screen at a time.

Flag	Description
-i*N*	Sets the default size of integers to *N* bytes. *N* must be 1, 2, or 4. The default setting is 4.
-l	Writes preprocessor error messages to the preprocessor's listing file as well as to the terminal. The listing file includes preprocessor error messages and your source text in a file named *filename.lis*, where filename is the name of the input file.
-lo	Like -l, but the generated BASIC code also appears in the listing file.
-o	Directs the preprocessor not to generate output files for include files. This flag does not affect the translated include statements in the main program. The preprocessor will generate a default extension for the translated include file statements unless you use the -o.*ext* flag.
-o.*ext*	Specifies the extension given by the preprocessor to both the translated include statements in the main program and the generated output files. If this flag is not provided, the default extension is ".bas". If you use this flag in combination with the -o flag, then the preprocessor generates the specified extension for the translated include statements, but does not generate new output files for the include statements.
-?	Shows which command line options are available for esqlb.
-r*N*	Sets the default size of reals to n bytes. *N* must be 4 or 8. The default setting is 4.
-s	Reads input from standard input and generates BASIC code to standard output. This is useful for testing unfamiliar statements. If you specify the -l option with this flag, the listing file is called "stdin.lis". To terminate the interactive session, type Ctrl Z.
-sqlcode	Indicates the file declares ANSI SQL code. The ANSI-92 specification describes SQLCODE as a "deprecated feature" and recommends using the SQLSTATE variable.
-[no]sqlcode	Tells the preprocessor not to assume a declared SQLCODE is for ANSI status information.
-w	Prints warning messages.
-wopen	This flag is identical to -wsql=open. However, -wopen is

Flag	Description
	supported only for backwards capability. See -wsql=open for more information.
-wsql=entry_ SQL92|open	Prints warning messages that indicate all non-entry SQL92 compliant syntax. Use *open* only with OpenSQL syntax. -wsql = open generates a warning if the preprocessor encounters an Embedded SQL statement that does not conform to OpenSQL syntax. (OpenSQL syntax is described in the *OpenSQL Reference Guide*.) This flag is useful if you intend to port an application across different Ingres Gateways. The warnings do not affect the generated code and the output file may be compiled. This flag does not validate the statement syntax for any SQL Gateway whose syntax is more restrictive than that of OpenSQL.

The Embedded SQL BASIC preprocessor assumes that input files are named with the extension ".sb". This default can be overridden by specifying the file extension of the input file(s) on the command line. The output of the preprocessor is a file of generated BASIC statements in tab format with the same name and the extension ".bas".

If you enter the command without specifying any flags or a filename, Ingres displays a list of flags available for the command.

The following examples present a range of the options available with **esqlb**:

Esqlb Command Examples

Command	Comment
esqlb file1	Preprocesses "file1.sb" to "file1.bas"
esqlb file2.xb	Preprocesses "file2.xb" to "file2.bas"
esqlb -l file3	Preprocesses "file3.sb" to "file3.bas" and creates listing "file3.lis"
esqlb -s	Accepts input from standard input
esqlb -ffile4.out file4	Preprocesses "file4.sb" to "file4.out"
esqlb	Displays a list of flags available for this command

The BASIC Compiler

As mentioned above, the preprocessor generates BASIC code. You should use the VMS **basic** command to compile this code. Most of the basic command line options can be used. You should not use the **g_float** or **h_float** qualifiers if floating-point values in the program are interacting with Ingres floating-point objects. If you use the **byte** or **word** compiler qualifiers, you must run the Embedded SQL preprocessor with the **-i1** or **-i2** flag. Similarly, use of the BASIC **double** qualifier requires that you have preprocessed your Embedded SQL file using the **-r8** flag. Note, too, that many of the statements that the Embedded SQL preprocessor generates are BASIC language extensions provided by VAX/VMS. Consequently, you should not attempt to compile with the **ansi_standard** qualifier.

The following example preprocesses and compiles the file “test1”. Note that both the Embedded SQL preprocessor and the BASIC compiler assume the default extensions.

```
$ esqlb test1
$ basic/list test1
```

VMS

As of Ingres II 2.0/0011 (axm.vms/00) Ingres uses member alignment and IEEE floating-point formats. Embedded programs must be compiled with member alignment turned on. In addition, embedded programs accessing floating-point data (including the MONEY data type) must be compiled to recognize IEEE floating-point formats.

Note: Check your Release Notes for any operating system specific information on compiling and linking ESQL/BASIC programs.

Linking an Embedded SQL Program

Embedded SQL programs require procedures from several VMS shared libraries in order to run properly. Once you have preprocessed and compiled an Embedded SQL program, you can link it. Assuming the object file for your program is called “dbentry,” use the following link command:

```
$ link dbentry.obj,-
  ii_system:[ingres.files]esql.opt/opt
```

Assembling and Linking Pre-Compiled Forms

The technique of declaring a pre-compiled form to the FRS is discussed in the *SQL Reference Guide* and in the BASIC Variables and Data Types in this chapter. To use such a form in your program, you must also follow the steps described here.

In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED lets you select the name for the file. Once you have created the MACRO file this way, you can assemble it into linkable object code with the following VMS command:

macro *filename*

The output of this command is a file with the extension ".obj". You then link this object file with your program by listing it in the link command, as in the following example:

```
$ link formentry,-
  empform.obj,-
  ii_system:[ingres.files]esql.opt/opt
```

Linking an Embedded SQL Program without Shared Libraries

While the use of shared libraries in linking Embedded SQL programs is recommended for optimal performance and ease of maintenance, non-shared versions of the libraries have been included in case you require them. Non-shared libraries required by Embedded SQL are listed in the esql.noshare options file. The options file must be included in your link command *after* all user modules. Libraries must be specified in the order given in the options file.

The following example demonstrates the link command for an Embedded SQL program called "dbentry" that has been preprocessed and compiled:

```
$ link dbentry,-
  ii_system:[ingres.files]esql.noshare/opt
```

Placing User-written Embedded SQL Routines in Shareable Images

When you plan to place your code in a shareable image, note the following about the **psect** attributes of your global or external variables.

- As a default, some compilers mark global variables as shared (SHR: every user who runs a program linked to the shareable image sees the same variable) and others mark them as not shared (NOSHR: every user who runs a program linked to the shareable image gets their own private copy of the variable).
- Some compilers support modifiers you can place in your source code variable declaration statements to explicitly state which attributes to assign a variable.
- The attributes that a compiler assigns to a variable can be overridden at link time with the **psect_attr** link option. This option overrides attributes of all variables in the **psect**.

  For further details, consult your compiler reference manual.

Include File Processing

The Embedded SQL **include** statement provides a means to include external files in your program's source code. The syntax of the statement is:

exec sql include *filename*

where *filename* is a quoted string constant specifying a file name or a logical name that points to the file name. If the file is in the local directory, you can also specify the filename without the surrounding quotes. If no extension is given to the file name (or to the file name pointed at by the logical name), the program assumes the default BASIC input file extension ".sb".

This statement is normally used to include variable declarations, although it is not restricted to such use. For more details on the **include** statement, see the *SQL Reference Guide*.

The included file is preprocessed and an output file with the same name but with the default output extension ".bas" is generated. You can override this default output extension with the **-o**.*ext* flag on the command line. The preprocessed output of the **include** statement is the BASIC **%include** directive. If the **-o** flag is used without an extension, then the output file is not generated for the **include** statement. This is useful for program libraries that use VMS MMS dependencies.

If you use both the **-o.*ext*** and the **-o** flags, then the preprocessor will generate the specified extension for the **include** statements in the program but will not generate new output files for the statements.

In the following example, assume that no overriding output extension was explicitly given on the command line. The Embedded SQL statement:

```
exec sql include 'employee.sb'
```

is preprocessed to the BASIC statement:

```
%include "employee.bas"
```

and the employee.sb file is translated into the BASIC employee.bas file.

In the next example, the system logical name "mydecls" points at the file "dra1:[headers]myvars.sb". If the following commands are invoked on the system level:

```
$ define mydecls dra1:[headers]myvars.sb
$ esqlb -o.hdr inputfile
```

the Embedded SQL statement:

```
exec sql include 'mydecls'
```

is preprocessed to the BASIC statement:

```
%include "dra1:[headers]myvars.hdr"
```

and the BASIC file 'dra1:[headers]myvars.hdr' is generated.

You can also specify include files with a relative path. For example, if you preprocess the file "dra1:[mysource]myfile.sb," the Embedded SQL statement:

```
exec sql include '[-.headers]myvars.sb'
```

is preprocessed to the BASIC statement:

```
%include "[-.headers]myvars.bas"
```

and the BASIC file "dra1:[headers]myvars.bas," is generated as output for the original include file, "dra1:[headers]myvars.sb."

Including Source Code with Labels

Some Embedded SQL statements generate labels. If you include files containing such statements, you must be careful to include the file only once in a given BASIC scope. Otherwise, you may find that the compiler later complains that the generated labels are defined more than once in that scope.

The statements that generate labels are the Embedded SQL block-type statements, such as **display**, **unloadtable**, and the **select**-loop.

Coding Requirements for Writing Embedded SQL Programs

The following sections describe coding requirements for Embedded SQL programs.

Comments Embedded in BASIC Output

Each Embedded SQL statement generates one comment and a few lines of BASIC code. You may find that the preprocessor translates 50 lines of Embedded SQL into 200 lines of BASIC. This can confuse the program developer who is trying to debug the original source code. To facilitate debugging, each group of BASIC statements associated with a particular statement is delimited by a comment corresponding to the original Embedded SQL source. Each comment is one line long and informs the reader of the file name, line number, and type of statement in the original source file.

Embedding Statements Inside BASIC If Blocks

As mentioned above, the preprocessor never generates line numbers on its own. Therefore, you can enclose Embedded SQL statements in the **then** or **else** clause of a BASIC **if** statement without changing program control. For example:

```
if (error = 1) then
        exec sql message 'Error on update'
        exec sql sleep 2
end if
```

Embedded SQL Statements that Do Not Generate Code

The following Embedded SQL declarative statements do not generate any BASIC code:

declare cursor
declare statement
declare table
whenever

These statements must not contain labels. Also, they must not be coded as the only statements in BASIC constructs that do not allow *empty* statements.

Embedded SQL/BASIC Preprocessor Errors

To correct most errors, you may wish to run the Embedded SQL preprocessor with the listing (**-l**) option on. The listing will be sufficient for locating the source and reason for the error.

For preprocessor error messages specific to BASIC, see Preprocessor Error Messages in this chapter.

Preprocessor Error Messages

The following is a list of error messages specific to BASIC.

E_DC000A "Table 'employee' contains column(s) of unlimited length."

Explanation: Character strings(s) of zero length have been generated. This causes a compile-time error. You must modify the output file to specify an appropriate length.

E_E30001 "BASIC array '%0c' should be subscripted."

Explanation: A variable declared as an array must be subscripted when used."

E_E30002 "Value assigned does not match BASIC constant type."

Explanation: The type of the literal assigned to the constant name does not match the type of the CONSTANT declaration. Numerics and strings cannot be mixed.

E_E30005 "BASIC identifier '%0c' expected on END RECORD/END GROUP statement."

Explanation: If you name the RECORD or GROUP declaration on the END RECORD or END GROUP statement, then the name must be the same with which the RECORD or GROUP was declared.

E_E30006 "RECORD or GROUP subscripts are required in '%0c'."

Explanation: In the specified variable reference, the record component lacks subscripts at the group or record level.

E_E30007 "RECORD or GROUP subscripts should not be used in '%0c'."

Explanation: In the specified variable reference, the record component has extra subscripts at the group or record level.

E_E3000A "Incorrect type used on EXTERNAL variable or constant."

Explanation: EXTERNAL variables can be declared with a limited subset of data types. The declaration refers to an unknown or non-EXTERNAL data type.

E_E3000B "EXTERNAL identifiers may not have subscripts or an assignment clause."

Explanation: The preprocessor does not support EXTERNAL arrays, or size-initialized variables. Use DIMENSION or COMMON for global non-scalar declarations.

E_E3000C "CONSTANT declaration may not refer to program-defined RECORD type."

Explanation: CONSTANT declarations may not refer to RECORD data types, even if they have been previously defined.

E_E3000D "CONSTANT declaration may not be subscripted."

Explanation: CONSTANT declarations may not refer to arrays.

E_E3000E "Assignment clause missing from BASIC CONSTANT declaration."

Explanation: A CONSTANT declaration must include an assignment to a numeric or string literal.

E_E3000F "Array subscripts missing from BASIC DIMENSION declaration."

Explanation: DIMENSION declarations must include array subscripts.

E_E30010 "String length is not allowed on BASIC DIMENSION declaration."

Explanation: DIMENSION declarations may not include string lengths nor an assignment clause.

E_E30011 "String length may only qualify a variable of STRING type."

Explanation: An assignment clause (string length) is only allowed with STRING declarations.

E_E30012 "String length is not allowed on dynamic string variable."

Explanation: A dynamic STRING type may not specify a length. A length may only be specified with static STRING declarations.

E_E30013 "BASIC variables must have an explicit type."

Explanation: All variable declarations must have an explicit type. Default types are not accepted by the preprocessor.

E_E30014 "Found identifier '%0c' where literal expected."

Explanation: You must use numeric or string literals to initialize constants. You must use a numeric literal when declaring the length of a static string variable.

E_E30017 "Quotes may not be embedded in string literals."

Explanation: In order to embed a quote in a string literal, you must use the BASIC rules to assign the string literal to a string variable, and use the variable in the embedded statement.

E_E3001A "Field '%0c' in record '%1c' is not elementary."

Explanation: The specified field was used as a variable. However, the field is not a scalar-valued variable (numeric or string). You cannot use arrays or records to set or retrieve data in this context.

Sample Applications

This section contains sample applications.

The Department-Employee Master/Detail Application

This application uses two database tables joined on a specific column. This typical example of a department and its employees demonstrates how to process two tables as a master and a detail.

The program scans through all the departments in a database table, in order to reduce expenses. Based on certain criteria, the program updates department and employee records. The conditions for updating the data are the following:

Departments:

- If a department has made less than $50,000 in sales, the department is dissolved.

Employees:

- If an employee was hired since the start of 1985, the employee is terminated.
- If the employee's yearly salary is more than the minimum company wage of $14,000 and the employee is not nearing retirement (over 58 years of age), the employee takes a 5% pay cut.
- If the employee's department is dissolved and the employee is not terminated, the employee is moved into a state of limbo to be resolved by a supervisor.

This program uses two cursors in a master/detail fashion. The first cursor is for the Department table, and the second cursor is for the Employee table. Both tables are described in **declare table** statements at the start of the program. The cursors retrieve all the information in the tables, some of which are updated. The cursor for the Employee table also retrieves an integer date interval whose value is positive if the employee was hired after January 1, 1985.

Each row that is scanned from both the Department table and the Employee table is recorded in an output file. This file serves both as a log of the session and as a simplified report of the updates that were made.

Each section of code is commented for the purpose of the application and also to clarify some of the uses of the Embedded SQL statements. The program illustrates table creation, multi-statement transactions, all cursor statements, direct updates, and error handling.

For readability, the BASIC exclamation point (!) is used as an end-of-line comment indicator.

```
10  !
    ! Program: Process_Expenses
    ! Purpose: Main entry point to process department and employee expenses.
    !

        exec sql include sqlca
        ! The department table
        exec sql declare dept table     &
            (name          char(12) not null,                   &
             totsales      money not null,                      &
             employees     smallint not null)

        ! The employee table
        exec sql declare employee table &
            (name          char(20) not null,                   &
             age           integer1 not null,                   &
             idno          integer not null,                    &
             hired         date not null,                       &
             dept          char(12) not null,                   &
             salary        money not null)

        ! "State-of-Limbo" for employees who lose their department
        exec sql declare toberesolved table &
            (name          char(20) not null,                   &
             age           integer1 not null,                   &
             idno          integer not null,                    &
             hired         date not null,                       &
             dept          char(12) not null,                   &
             salary        money not null)

       print 'Entering application to process expenses.'
       open "expenses.log" for output as file #1
       call Init_Db
       call Process_Depts
       call End_Db
       close #1
       print 'Successful completion of application.'

end                                      ! of Process_Expenses

    !
    ! Subroutine: Init_Db
    ! Purpose: Initialize the database. Connect to the database,
    ! and abort if an error. Before processing employees create the table for
    ! employees who lose their department,"toberesolved".
    ! Parameters: None.
    !

100 sub Init_Db
    exec sql include sqlca
    exec sql whenever sqlerror stop
    exec sql connect personnel
        print #1, 'Creating "To_Be_Resolved" table.'
        exec sql create table toberesolved                      &
            (name       char(20) not null,                      &
             age        integer1 not null,                      &
             idno       integer not null,                       &
             hired      date not null,                          &
             dept       char(12) not null,                      &
             salary     money not null)

    end sub ! of Init_Db
```

```
    !
    ! Subroutine: End_Db
    ! Purpose: Commit the multi-statement transaction and disconnect
    ! from the database.
    ! Parameters: None.
    !
 200 sub End_Db
        exec sql include sqlca
        exec sql commit
        exec sql disconnect
    end sub                                         ! of End_Db

    !
    ! Subroutine: Process_Depts
    ! Purpose: Scan through all the departments, processing each one.
    ! If the department has made less than $50,000 in sales
    ! then the department is dissolved. For each department,
    ! process all the employees (they may even be moved to another table).
    1 If an employee was terminated, then update the department's employee
    ! counter.
    ! Parameters: None
    !

300 sub Process_Depts
        exec sql include sqlca
        exec sql begin declare section
        record department
            string      dname = 12
            real        totsales
            word        employees
        end record
        declare department      dept
        declare word            emps_term  ! Employees terminated
        declare string          loc_dname  ! For parameter passing
    exec sql end declare section

    ! Minimum sales of department
    declare real constant MIN_DEPT_SALES = 50000.00
    ! Was the dept deleted?
    declare byte deleted_dept                    ! Was the dept declared?
    declare string dept_format                   ! Formatting value
    exec sql declare deptcsr cursor for                          &
        select name, totsales, employees                         &
        from dept                                                &
        for direct update of name, employees
    ! All errors from this point on close down the application
    exec sql whenever sqlerror call Close_Down
    exec sql whenever not found goto CloseDCsr
    ! Close deptcsr
    exec sql open deptcsr
    while (sqlcode = 0)

        exec sql fetch deptcsr into :dept
        ! Did the department reach minimum sales?
        if (dept::totsales \ MIN_DEPT_SALES) then
            exec sql delete from dept                            &
                where current of deptcsr
                        deleted_dept = 1
                        dept_format = ' -- DISSOLVED --'
            else
                        deleted_dept = 0
                        dept_format = ' '
            end if
        ! Log what we have just done
        print #1, 'Department: ' + (dept::dname)                 &
```

```
            + ', Total Sales: ';
        print #1 using '$$####.##', dept::totsales;
        print #1, dept_format
        ! Now process each employee in the department
        loc_dname = dept::dname
        call Process_Employees(loc_dname, deleted_dept, emps_term)

        ! If some employees were terminated, record this fact
        if (emps_term > 0 and deleted_dept = 0) then
                exec sql update dept &
                        set employees = :dept::employees - :emps_term &
                        where current of deptcsr
        end if
    next
        exec sql whenever not found continue
        CloseDCsr: EXEC SQL CLOSE deptcsr
        end sub ! of Process_Depts
        !
        ! Subroutine: Process_Employees
        ! Purpose: Scan through all the employees for a particular
        ! department. Based on given conditions the employee
        ! may be terminated, or given a salary reduction.
        ! 1. If an employee was hired since 1985 then the
        ! employee is terminated.
        ! 2. If the employee's yearly salary is more than the
        ! minimum company wage of $14,000 and the employee
        ! is not close to retirement (over 58 years of
        !age), then the employee takes a 5% salary reduction.
        ! 3. If the employee's department is dissolved and the
        ! employee is not terminated, then the employee is
        ! moved into the "toberesolved" table.
        !
        ! Parameters:   loc_dname  -     Name of current department
        !               deleted_dept - Is current department being dissolved?
        !               emps_term  - Set locally to record how many employees
        !                        were terminated for the current department.
        !

400 sub Process_Employees(string loc_dname, byte deleted_dept, &
                          integer emps_term)

        exec sql include sqlca
        exec sql begin declare section
            record employee ! Corresponds to "employee" table
                string          ename = 20
                word            age
                long            idno
                string          hired = 25
                real            salary
                long            hired_since_85
            end record
            declare employee emp
            declare real constant SALARY_REDUC = 0.95
        exec sql end declare section
        ! Minimum employee salary
        declare real constant           MIN_EMP_SALARY = 14000.00
        declare integer constant        NEARLY_RETIRED = 58
        declare string title                            ! Formatting values
        declare string description
        ! Note the use of the INGRES function to find out who was hired
        ! since 1985.

        exec sql declare empcsr cursor for                        &
            select name, age, idno, hired, salary,                &
                int4(interval('days', hired-date('01-jan-1985'))) &
            from employee &
```

```
        where dept = :loc_dname &
        for direct update of name, salary
    ! All errors from this point on close down the application
    exec sql whenever sqlerror call Close_Down
    exec sql whenever not found goto CloseECsr ! Close empcsr
    exec sql open empcsr
    emps_term = 0
        while (sqlcode = 0)

            exec sql fetch empcsr into :emp
            if (emp::hired_since_85 > 0) then
                    exec sql delete from employee                   &
                       where current of empcsr
                    title = 'Terminated: '
                    description = 'Reason: Hired since 85.'
                    emps_term = emps_term +1
        else
            if (emp::salary > MIN_EMP_SALARY) then
                    ! Reduce salary if not nearly retired
                    if (emp::age < NEARLY_RETIRED) then
                            exec sql update employee                &
                            set salary = salary * :SALARY_REDUC     &
                            where current of empcsr
                            title = 'Reduction: '
                            description = 'Reason: Salary.'
                    else
                    ! Do not reduce salary
                    title = 'No Changes: '
                    description = 'Reason: Retiring.'
            end if

        else
            ! Leave employee alone
            title = 'No Changes: '
            description = 'Reason: Salary.'
        end if
    end if
    ! Was employee's department dissolved?
    if (deleted_dept = 1) then
        exec sql insert into toberesolved                           &
            select *                                                &
            from employee                                           &
            where idno = :emp::idno

        exec sql delete from employee                               &
            where current of empcsr
    end if
    ! Log the employee's information
    print #1, ' ' + title;
    print #1, str$(emp::idno);
    print #1, ', ' + (emp::ename) + ', ';
    print #1, str$(emp::age) + ', ';
    print #1 using '$$####.##', emp::salary;
    print #1, '; ' + description
next
  exec sql whenever not found continue
  CloseEcsr:        exec sql close empcsr
end sub ! of Process_Employees
!
! Subroutine: Close_Down
! Purpose: Error handler called any time after Init_Db was successfully
! completed. In all cases print the cause of the error, and abort the
! transaction, backing out changes. Note that disconnecting from the database
! will implicitly close any open cursors too.
! Parameters: None
!
```

```
500 sub Close_Down
        exec sql include sqlca
        exec sql begin declare section
            declare string errbuf
        exec sql end declare section
        exec sql whenever sqlerror continue ! Turn off error handling
        exec sql inquire_sql(:errbuf = errortext)
        print 'Closed down because of database error:'
        print errbuf
        close #1
        exec sql rollback
        exec sql disconnect
        stop
    end sub                              ! of Close_Down
```

The Table Editor Table Field Application

This application edits the Person table in the Personnel database. It is a forms application that allows the user to update a person's values, remove the person, or add new persons. Various table field utilities are provided with the application to demonstrate how they work.

The objects used in this application are:

Object	Description
personnel	The program's database environment.
person	A table in the database, with three columns: name (**char**(20)) age (**smallint**) number (**integer**) Number is unique.
personfrm	The VIFRED form with a single table field.
persontbl	A table field in the form, with two columns: name (**char(20)**) age (**integer**) When initialized, the table field includes the hidden column, number (**integer**).

At the start of the application, a database cursor is opened to load the table field with data from the Person table. After loading the table field, you can browse and edit the displayed values. You can add, update, or delete entries. When finished, the values are unloaded from the table field and, in a multi-statement transaction, your updates are transferred back into the Person table.

Also for readability, the BASIC exclamation point (!) is used as an end-of-line comment indicator.

```
10  !

    ! Program:  Table_Edit
    ! Purpose:  Main entry point to edit the "person" table in the
    !           database, using a table field.
    !

    exec sql include sqlca
    exec sql declare person table                        &
        (name   char(20),                                &
         age    smallint,                                &
         number integer)

    exec sql begin declare section
    ! Person information
    declare string p_name                        ! Full name
    declare integer p_age                        ! Age of person
    declare integer p_number                     ! Unique person number
    declare integer maxid                        ! Max person id number
    ! Table field entry information
    declare integer state                        ! State of data set entry
    declare integer recnum                       ! Record number
    declare integer lastrow                      ! Last row in table field
    ! Utility buffers
    declare string msgbuf                        ! Message buffer
    declare string respbuf                       ! Response buffer for prompts
    exec sql end declare section
    declare byte update_error                    ! Update error from database
    declare byte xact_aborted                    ! Transaction aborted
    external integer function Load_Table         ! Function to fill table field
    ! Table field row states
    declare byte constant ROWUNDEF       = 0 ! Empty or undefined row
    declare byte constant ROWNEW         = 1 ! Appended by user
    declare byte constant ROWUNCHANGD    = 2 ! Loaded by program, same
    declare byte constant ROWCHANGD      = 3 ! Loaded by program, changed
    declare byte constant ROWDELETE      = 4 ! Deleted by program
    declare byte constant NOTFOUND       = 100 ! SQL value for no rows
    ! Set up error handling for main program
    exec sql whenever sqlwarning continue
    exec sql whenever not found continue
    exec sql whenever sqlerror stop
    ! Start up Ingres and the Ingres/Forms system
    exec sql connect 'personnel'

    exec frs forms
    ! Verify that the user can edit the "person" table
    exec frs prompt noecho ('Password for table editor: ', :respbuf)

    if (respbuf <> 'MASTER_OF_ALL') then
        exec frs endforms
        exec sql disconnect
        print 'No permission for task. Exiting . . .'
        stop
```

```
end if
! We assume no SQL errors can happen during screen updating
exec sql whenever sqlerror continue
exec frs message 'Initializing Person Form . . .'
exec frs forminit personfrm
!
! Initialize "persontbl" table field with a data set in FILL mode
! so that the runtime user can append rows. To keep track of
! events occurring to original rows that will be loaded into
! the table field, hide the unique person number.
!
exec frs inittable personfrm persontbl fill (number = integer)

maxid = Load_Table
exec frs display personfrm update
    exec frs initialize
exec frs activate menuitem 'Top'
    exec frs begin
    !
    ! Provide menu, as well as system FRS keys to scroll
    ! to both extremes of the table field.
    !
    exec frs scroll personfrm persontbl to 1
exec frs end ! 'Top'

exec frs activate menuitem 'Bottom'
exec frs begin
    exec frs scroll personfrm persontbl to end ! Forward
exec frs end ! 'Bottom'

exec frs activate menuitem 'Remove'
    exec frs begin
    !
    ! Remove the person in the row the user's cursor is on.
    ! If there are no persons, exit operation with message.
    ! Note that this check cannot really happen as there is
    ! always an undefined row in fill mode.
    !
    exec frs inquire_frs table personfrm &
        (:lastrow = lastrow(persontbl))
    if (lastrow = 0) then
        exec frs message 'Nobody to Remove'
        exec frs sleep 2
        exec frs resume field persontbl
    end if
exec frs deleterow personfrm persontbl ! Record it later
exec frs end ! 'Remove'

exec frs activate menuitem 'Find'
exec frs begin
    !
    ! Scroll user to the requested table field entry.
    ! Prompt the user for a name, and if one is typed in
    ! loop through the data set searching for it.
    !
    exec frs prompt ('Name of person: ', :respbuf)
    if (respbuf = '') then
        exec frs resume field persontbl
    end if
    exec frs unloadtable personfrm persontbl                  &
        (:p_name = name,                                      &
         :recnum = _record,                                   &
         :state = _state)
    exec frs begin
    ! Do not compare with deleted rows
    if ((p_name = respbuf) and (state <> ROWDELETE)) then
```

```
            exec frs scroll personfrm persontbl to :recnum
            exec frs resume field persontbl
        end if
        exec frs end
        ! Fell out of loop without finding name
        msgbuf =
         'Person "'+respbuf+'" not found in table [HIT RETURN] '
        exec frs prompt noecho (:msgbuf, :respbuf)

    exec frs end ! 'Find'

    exec frs activate menuitem 'Exit'
    exec frs begin
        exec frs validate field persontbl
        exec frs breakdisplay
    exec frs end ! 'Exit'

    exec frs finalize

!
! Exit person table editor and unload the table field. If any
! updates, deletions or additions were made, duplicate these
! changes in the source table. If the user added new people we
! must assign a unique person id before returning it to the table.
! To do this, increment the previously saved maximum id number
! with each insert.
!

! Do all the updates in a transaction
exec sql savepoint savept

!
! Hard code the error handling in the UNLOADTABLE loop, as
! we want to cleanly exit the loop.
!
exec sql whenever sqlerror continue

update_error = 0
xact_aborted = 0

exec frs message 'Exiting Person Application . . .'
exec frs unloadtable personfrm persontbl                        &
   (:p_name = name, :p_age = age,                               &
    :p_number = number, :state = _state)
exec frs begin
   ! Appended by user. Insert with new unique id
   if (state = ROWNEW) then
       maxid = maxid + 1
       exec sql insert into person (name, age, number)          &
           values (:p_name, :p_age, :maxid)

   ! Updated by user. Reflect in table
   else
   if (state = ROWCHANGD) then
       exec sql update person set                               &
           name = :p_name, age = :p_age                         &
           where number = :p_number
!
! Deleted by user, so delete from table. Note that only
! original rows are saved by the program, and not rows
! appended at runtime by the user.
!
       else
           if (state = rowdelete) then
               exec sql delete from person                      &
                       where number = :p_number
```

```
            end if
        end if
        end if                  ! ignore undefined or unchanged - No updates
        !
        ! Handle error conditions -
        ! If an error occurred, then abort the transaction.
        ! If a no rows were updated then inform user, and
        ! prompt for continuation.
        !
        if (sqlcode < 0) then
 ! SQL error
            exec sql inquire_sql (:msgbuf = errortext)
            exec sql rollback to savept
            update_error = 1
            xact_aborted = 1
            exec frs endloop
        else
            if (sqlcode = NOTFOUND) then
                msgbuf = 'Person "' + p_name + &
                        '" not updated. Abort all updates?'
                exec frs prompt (:msgbuf, :respbuf)
                if (respbuf = 'Y' or respbuf = 'y') then
                        exec sql rollback to savept
                        xact_aborted = 1
                        exec frs endloop
                end if
            end if
        end if
    exec frs end                                ! 'Unloadtable'
    if (xact_aborted = 0) then
        exec sql commit                         ! Commit the updates
    end if
    exec frs endforms                   ! Terminate the Forms and Ingres
    exec sql disconnect
    if (update_error = 1) then
        print 'Your updates were aborted because of error:';
        print msgbuf
    end if

    end ! of Table_Edit - Main Program
 !
 ! Function:     Load_Table
 ! Purpose:      Load the table field from the 'person' table.
!                The columns 'name' and 'age' will be displayed, and 'number'
 !               will be hidden.
 ! Parameters:   None
 ! Returns: Maximum employee number
 !

20 function integer Load_Table
    exec sql include sqlca
    !
    ! Declare person information:
    ! The preprocessor already knows that these variables have been
    ! declared from their declarations in the main program.
    !
    declare string p_name                       ! Full name
    declare integer p_age                       ! Age of person
    declare integer p_number                    ! Unique person number
    declare integer maxid                       ! Max person id number to return
    exec sql declare loadtab cursor for                         &
        select name, age, number                                &
        from person
    ! Set up error handling for loading procedure
    exec sql whenever sqlerror goto Closeld ! Close loadtab
    exec sql whenever not found goto Closeld ! Close loadtab
```

```
    exec frs message 'Loading Person Information . . .'

    maxid = 0
    ! Fetch the maximum person id number for later use
    exec sql select max(number) &
        into :maxid &
        from person
    exec sql open loadtab
        while (sqlcode = 0)
            ! Fetch data into record, and load table field
            exec sql fetch loadtab into :p_name, :p_age, :p_number
            exec frs loadtable personfrm persontbl &
                (name = :p_name, age = :p_age, number = :p_number)
        next
        exec sql whenever sqlerror continue
        Closeld: exec sql close loadtab
        Load_Table = maxid

end function                                     ! of Load_Table
```

The Professor-Student Mixed Form Application

This application lets the user browse and update information about graduate students who report to a specific professor. The program is structured in a master/detail fashion, with the professor being the master entry, and the students the detail entries. The application uses two forms—one to contain general professor information and another for detailed student information.

The objects used in this application are:

Obejct	Description
personnel	The program's database environment.
professor	A database table with two columns: pname (**char(25)**) pdept (**char(10)**) See its **declare table** statement in the program for a full description.
student	A database table with seven columns: sname (**char(25)**) sage **(integer1**) sbdate (**char(25)**) sgpa (**float4**) sidno (**integer**) scomment (**varchar(200)**) sadvisor (**char(25)**) See its **declare table** statement for a full description. The sadvisor column is the join field with the pname column in the Professor table.

Obejct	Description
masterfrm	The main form has the pname and pdept fields, which correspond to the information in the Professor table, and the studenttbl table field. The pdept field is display only.
studenttbl	A table field in “masterfrm” with two columns, “sname” and “sage”. When initialized, it also has five hidden columns corresponding to information in the student table.
studentfrm	The detail form, with seven fields, which correspond to information in the Student table. Only the sgpa, scomment, and sadvisor fields are updatable. All other fields are display-only.
grad	A global BASIC record, whose fields correspond in name and type to the columns of the student database table, the studentfrm form, and the studenttbl table field.

The program uses “masterfrm” as the general-level master entry, in which data can only be retrieved and browsed. It uses “studentfrm” as the detailed screen, in which specific student information can be updated.

The user can enter a name in the pname field and then select the **Students** menu operation. The operation fills the studenttbl table field with detailed information of the students reporting to the named professor. This is done by the studentcsr database cursor in the Load_Students procedure. The program assumes that each professor is associated with exactly one department.

The user can then browse the table field (in **read** mode), which displays only the names and ages of the students. More information about a specific student can be requested by selecting the **Zoom** menu operation. This operation displays the form “studentfrm” (in **update** mode). The fields of studentfrm are filled with values stored in the hidden columns of studenttbl. The user can make changes to three fields (sgpa, scomment, and sadvisor). If validated, these changes will be written back to the database table (based on the unique student id), and to the table field’s data set. This process can be repeated for different professor names.

Also for readability, the BASIC exclamation point (!) is used as an end-of-line comment indicator.

```
10  !
    ! Program: Professor_Student
    ! Purpose: Main entry point into "Professor-Student"
    !          mixed-form master detail application.
    !

        exec sql include sqlca
        exec sql declare student table                          &
            (sname       char(25),                              &
             sage        integer1,                              &
             sbdate      char(25),                              &
             sgpa        float4,                                &
             sidno       integer,                               &
             scomment    varchar(200),                          &
             sadvisor    char(25))

        exec sql declare professor table                        &
            (pname       char(25),                              &
             pdept       char(10))
        exec sql begin declare section
            ! Externally compiled master and student form
            external integer masterfrm, studentfrm
        exec sql end declare section
        ! Start up Ingres and the Forms system
        exec frs forms
        exec sql whenever sqlerror stop
        exec frs message 'Initializing Student Administrator . . .'
        exec sql connect personnel
        exec frs addform :masterfrm
        exec frs addform :studentfrm
        call Master
        exec frs clear screen
        exec frs endforms
        exec sql disconnect
    end ! of Professor_Student

 !
 ! Subroutine:  Master
 ! Purpose:     Drive the application, by running 'masterfrm', and
 !              allowing the user to 'zoom' into a selected student.
 ! Parameters:  None - Uses the global student 'grad' record.
 !
 100 sub Master
        exec sql include sqlca
        exec sql begin declare section
            ! Global grad student record maps to database table
            record grad_student
                string  sname = 25
                word    sage
                string  sbdate = 25
                real    sgpa
                integer sidno
                string  scomment = 200
                string  sadvisor = 25
            end record
            common (grad_area) grad_student grad
            ! Professor info maps to database table
            record professor
                string pname = 25
                string pdept = 10
            end record
            declare professor prof
            ! Useful forms system information
        declare integer lastrow          ! Lastrow in table field
            declare integer istable              ! Is a table field?
```

```
! Local utility buffers
declare string msgbuf                ! Message buffer
declare string respbuf               ! Response buffer
declare string old_advisor           ! Old advisor before Zoom
exec sql end declare section
external byte function Student_Info_changed
                                    ! Function defined below
declare string tmp_pname            ! Temporary string param
!
! Initialize "studenttbl" with a data set in READ mode.
! Declare hidden columns for all the extra fields that
! the program will display when more information is
! requested about a student. Columns "sname" and "sage"
! are displayed, all other columns are hidden, to be
! used in the student information form.
!
exec frs inittable masterfrm studenttbl read                 &
    (sbdate  = char(25),                                     &
     sgpa = float4,                                          &
     sidno = integer,                                        &
     scomment = char(200),                                   &
     sadvisor = char(20))

exec frs display masterfrm update
exec frs initialize
exec frs begin
    exec frs message 'Enter an Advisor name . . .'
    exec frs sleep 2
exec frs end
exec frs activate menuitem 'Students', FIELD 'pname'
exec frs begin
    ! Load the students of the specified professor
    exec frs getform (:prof::pname = pname)

    ! If no professor name is given then resume
    if (prof::pname = '') then
            exec frs resume field pname
    end if
    !
    ! Verify that the professor exists. Local error
    ! handling just prints the message, and continues.
    ! We assume that each professor has exactly one
    ! department.
    !
    exec sql whenever sqlerror call sqlprint
    exec sql whenever not found continue
    prof::pdept = ' '
    exec sql select pdept                                    &
            into :prof::pdept                                &
            from professor                                   &
            where pname = :prof::pname
    if (prof::pdept = '') then
            msgbuf = 'No professor with name "' +            &
            prof::pname + '" [RETURN]'
            exec frs prompt noecho (:msgbuf, :respbuf)
            exec frs clear field all
            exec frs resume field pname
    end if
    ! Fill the department field and load students
    exec frs putform (pdept = :prof::pdept)
    exec frs redisplay  ! Refresh for query
    tmp_pname = prof::pname
    call Load_Students(tmp_pname)

    exec frs resume field studenttbl
exec frs end ! 'Students'
```

```
    exec frs activate menuitem 'Zoom'
    exec frs begin
        !
        ! Confirm that user is on "studenttbl", and that
        ! the table field is not empty. Collect data from
        ! the row and zoom for browsing and updating.
        !
        exec frs inquire_frs field masterfrm                     &
                (:istable = table)
        if (istable = 0) then
                exec frs prompt noecho                           &
                ('Select from the student table [RETURN]',       &
                :respbuf)
                exec frs resume field studenttbl
        end if
        exec frs inquire_frs table masterfrm                     &
                (:lastrow = lastrow)

        if (lastrow = 0) then
                exec frs prompt noecho                           &
                ('There are no students [RETURN]', :respbuf)
                exec frs resume field pname
        end if
        ! Collect all data on student into global record
        exec frs getrow masterfrm studenttbl                     &
                (:grad::sname = sname,                           &
                :grad::sage = sage,                              &
                :grad::sbdate = sbdate,                          &
                :grad::sgpa = sgpa,                              &
                :grad::sidno = sidno,                            &
                :grad::scomment = scomment,                      &
                :grad::sadvisor = sadvisor)

        !
        ! Display "studentfrm", and if any changes were made
        ! make the updates to the local table field row.
        ! Only make updates to the columns corresponding to
        ! writable fields in "studentfrm". If the student
        ! changed advisors, then delete this row from the
        ! display.
        !
        old_advisor = grad::sadvisor
        if (Student_Info_Changed = 1) then
        if (old_advisor <> grad::sadvisor) then
                exec frs deleterow masterfrm studenttbl
        else
                exec frs putrow masterfrm studenttbl             &
                        (sgpa = :grad::sgpa,                     &
                        scomment = :grad::scomment,              &
                        sadvisor = :grad::sadvisor)
                end if
        end if
        exec frs end                            ! 'Zoom'

        exec frs activate menuitem 'Exit'
        exec frs begin
                exec frs breakdisplay
        exec frs end                            ! 'Exit'

        exec frs finalize
end sub                                         ! Master
!
! Subroutine: Load_Students
! Purpose:    Given an advisor name, load into the 'studenttbl'
!             table field all the students who report to the
```

```
        !             professor with that name.
        ! Parameters:
        !             advisor - User specified professor name.
        !             Uses the global student record.
        !

200 sub Load_Students(string tmp_advisor)

            exec sql include sqlca
            exec sql begin declare section
                declare string advisor
            exec sql end declare section
        !
        ! Global grad student - do not redeclare the structure as it
        ! was declared in subroutine "Master"
        !
        record grad_student
                string          sname = 25
                word            sage
                string          sbdate = 25
                real            sgpa
                integer         sidno
                string          scomment = 200
                string          sadvisor = 25
        end record
        common (grad_area) grad_student grad
        exec sql declare studentcsr cursor for                          &
            select sname, sage, sbdate, sgpa,                           &
                sidno, scomment, sadvisor                               &
            from student                                                &
            where sadvisor = :advisor
        ! Move string parameter into variable known by preprocessor
        advisor = tmp_advisor
        !
        ! Clear previous contents of table field. Load the table
        ! field from the database table based on the advisor name.
        ! Columns "sname" and "sage" will be displayed, and all
        ! others will be hidden.
        !
        exec frs message 'Retrieving Student Information . . .'

        exec frs clear field studenttbl
        exec sql whenever sqlerror goto EndLoad ! End loading
        exec sql whenever not found goto EndLoad
        exec sql open studentcsr
        !
        ! Before we start the loop we know that the OPEN was
        ! successful and that NOT FOUND was not set.
        !
        while (sqlcode = 0)

            exec sql fetch studentcsr into :grad
            exec frs loadtable masterfrm studenttbl                     &
                (sname = :grad::sname,                                  &
                 sage = :grad::sage,                                    &
                 sbdate = :grad::sbdate,                                &
                 sgpa = :grad::sgpa,                                    &
                 sidno = :grad::sidno,                                  &
                 scomment = :grad::scomment,                            &
                sadvisor = :grad::sadvisor)

        next
        ! Clean up on an error, and close cursors
        exec sql whenever not found continue
        exec sql whenever sqlerror continue
        EndLoad: exec sql close studentcsr
```

```
 end sub                                                ! Load_Students
        !
        ! Function:     Student_Info_Changed
        ! Purpose:      Allow the user to zoom into the details of
        !               a selected student. Some of the data can be
        !               updated by the user.If any updates were made,
        !               then reflect these back into the database table.
        !               The procedure returns TRUE if any changes were made.
        ! Parameters:   None - Uses with data in the global "grad" record.
        ! Returns:      TRUE/FALSE - Changes were made to the database.
        !               Sets the global "grad" record with the new data.
        !

300 function byte Student_Info_Changed
        exec sql include sqlca
        exec sql begin declare section
            declare integer changed             ! Changes made to data in form
            declare integer valid_advisor       ! Valid advisor name ?
        exec sql end declare section
        !
        ! Global grad student - do not redeclare the structure as it
        ! was declared in subroutine "Master"
        !
        record grad_student
            string      sname = 25
            word        sage
            string      sbdate = 25
            real        sgpa
            integer     sidno
            string      scomment = 200
            string      sadvisor = 25
        end record
        common (grad_area) grad_student grad
        ! Local error handle just prints error, and continues
        exec sql whenever sqlerror call sqlprint
        exec sql whenever not found continue
        exec frs display studentfrm fill
        exec frs initialize &
               (sname = :grad::sname,                          &
                sage = :grad::sage,                            &
                sbdate = :grad::sbdate,                        &
                sgpa = :grad::sgpa,                            &
                sidno = :grad::sidno,                          &
                scomment = :grad::scomment,                    &
                sadvisor = :grad::sadvisor)
        exec frs activate menuitem 'Write'
        exec frs begin
            !
            ! If changes were made then update the database table.
            ! Only bother with the fields that are not read-only.
            !
            exec frs inquire_frs form (:changed = change)
            if (changed = 1) then
                exec frs validate
                exec frs getform                               &
                       (:grad::sgpa = sgpa,                    &
                         grad::scomment = scomment,            &
                        :grad::sadvisor = sadvisor)

                ! Enforce integrity of professor name
                valid_advisor = 0
                exec sql select 1 into :valid_advisor          &
                        from professor                         &
                        where pname = :grad::sadvisor
                if (valid_advisor = 0) then
```

```
                exec frs message 'Not a valid advisor name'
                exec frs sleep 2
                exec frs resume field sadvisor
        end if
        exec frs message 'Writing changes to database. . .'
        exec sql update student set                                &
                sgpa = :grad::sgpa,                                &
                scomment = :grad::scomment,                        &
                sadvisor = :grad::sadvisor                         &
                where sidno = :grad::sidno
        end if
        exec frs breakdisplay
    exec frs end                                ! 'Write'

    exec frs activate menuitem 'Quit'
    exec frs begin
        ! Quit without submitting changes
        changed = 0
        exec frs breakdisplay
    exec frs end                                ! 'Quit'

    exec frs finalize
    Student_Info_Changed = changed

end function                                    ! Student_Info_Changed
```

The SQL Terminal Monitor Application

This application executes SQL statements that are read in from the terminal. The application reads statements from input and writes results to output. Dynamic SQL is used to process and execute the statements.

When the program starts, it prompts the user for the database name. The program then prompts for an SQL statement. SQL comments and statement delimiters are not accepted. The SQL statement is processed using dynamic SQL, and results and SQL errors are written to output. At the end of the results, the program displays an indicator of the number of rows affected. The loop is then continued and the program prompts you for another SQL statement. When end-of-file is typed in, the application rolls back any pending updates and disconnects from the database.

The user's SQL statement is prepared using **prepare** and **describe**. If the SQL statement is not a **select** statement, then it is run using **execute** and the number of rows affected is printed. If the SQL statement is a **select** statement, a dynamic SQL cursor is opened, and all the rows are fetched and printed. The routines that print the results do not try to tabulate the results. A row of column names is printed, followed by each row of the results.

Keyboard interrupts are not handled. Fatal errors, such as allocation errors, and boundary condition violations are handled by rolling back pending updates and disconnecting from the database session.

```
100     !
        ! Program: SQL_Monitor
        ! Purpose: Main entry of SQL Monitor application. Prompt
        !          for database name and connect to the database.
        !          Run the monitor and disconnect from the database.
        !          Before disconnecting roll back any pending updates.
        !

        program SQL_Monitor
            exec sql include sqlca
            exec sql begin declare section
                declare string dbname            ! Database name
                exec sql end declare section
            linput 'SQL Database'; dbname        ! Prompt for database name
            if (dbname = '') then
                exit program
            end if
            print '-- SQL Terminal Monitor --'

            exec sql whenever sqlerror stop      ! Connection errors are fatal
            exec sql connect :dbname
            call Run_Monitor
            exec sql whenever sqlerror continue
            print 'SQL: Exiting monitor program.'

            exec sql rollback
            exec sql disconnect
        end program ! SQL_Monitor
        !
        ! Subroutine:   Run_Monitor
        ! Purpose:      Run the SQL monitor. Initialize the global
        !               SQLDA with the number of SQLVAR elements. Loop
        !               while prompting the user for input and processing
        !               the SQL statement;if end-of-file is typed then
        !               return to the main program.
        !
        !               If the statement is not a SELECT statement
        !               then EXECUTE it, otherwise open a cursor a process
        !               a dynamic SELECT statement (using Execute_Select).
        !

200     sub Run_Monitor
        ! Declare the global SQLCA and the SQLDA records
        exec sql include sqlca
        exec sql include sqlda
        common (sqlda_area) IISQLDA sqlda
        exec sql begin declare section
        declare string stmt_buf          ! SQL statement input buffer
        exec sql end declare section
        declare integer stmt_num                 ! SQL statement number
        declare integer rows                     ! Rows affected
        external byte function Read_Stmt         ! Function to read input
        external integer function Execute_Select  ! and to execute SELECTs
        exec sql declare stmt statement          ! Dynamic SQL statement
        sqlda::sqln = IISQ_MAX_COLS ! Initialize the SQLDA
        stmt_num = 1
        !
        ! Now we are set for input. Call Read_Stmt each time through
        ! the loop. Read_Stmt prompts the user for input (into
        ! stmt_buf) and returns 0 if end-of-file was typed.
        !
        while (Read_Stmt(stmt_num, stmt_buf))

            stmt_num = stmt_num + 1
            ! SQL errors cause current statement to be aborted.
            exec sql whenever sqlerror goto Stmt_Err
```

```
            !
            ! PREPARE and DESCRIBE the statement. If the statement
            ! is not a SELECT then EXECUTE it, otherwise inspect the
            ! contents of the SQLDA and call Execute_Select.
            !
            exec sql prepare stmt from :stmt_buf
            exec sql describe stmt into :sqlda
            !
            ! If SQLD = 0 then this is not a SELECT statement. Otherwise
            ! call Execute_Select to process a dynamic cursor.
            !
            if (sqlda::sqld = 0) then
                exec sql execute stmt
                rows = sqlerrd(2)

            else
                rows = Execute_Select
            end if                               ! If SELECT or not
            exec sql whenever sqlerror continue
        Stmt_Err:
            !
            ! Only display error message if we arrived here because
            ! of the SQLERROR condition. Otherwise print the rows
            ! processed and continue with the loop.
            !
            if (sqlcode < 0) then
                call Print_Error
            else
                print '[' + str$(rows) + ' row(s)]'
            end if
        next                                     ! While reading statements
    end sub ! Run_Monitor
    !
    ! Function:  Execute_Select
    ! Purpose:   Run a dynamic SELECT statement. The SQLDA has
    !            already been described. This routine calls Print_Header
    !            to print column headers and set up result storage
    !            information. A Dynamic SQL cursor is then opened, and
    !            each row is fetched and printed by Print_Row.
    !            Any error causes the cursor to be closed.
    ! Returns:   Number of rows fetched from cursor.
    !

300 function integer execute_select
    ! Declare the global SQLCA and the SQLDA records
    exec sql include sqlca
    exec sql include sqlda
    common (sqlda_area) IISQLDA sqlda
    declare integer rows                         ! Counter for rows fetched
    external byte function Print_Header          ! Function to set up header
    exec sql declare csr cursor for stmt         ! Cursor for dynamic statement
    !
    ! Print the result column names and set up the result data
    ! types and variables. Print_Header returns 0 if it fails.
    !
    if (not Print_Header) then
        Execute_Select = 0
        exit function
    end if
    exec sql whenever sqlerror goto Close_Csr
    rows = 0
    ! Open the dynamic cursor.
    exec sql open csr
    ! Fetch and print each row.
    while (sqlcode = 0)
```

```
        exec sql fetch csr using descriptor :sqlda
        if (sqlcode = 0) then
            rows = rows + 1                             ! Count the rows
            call Print_Row
        end if
    next                                            ! While there are more rows
    Close_Csr:
        ! Display error message if the SQLERROR condition was set.
        if (sqlcode < 0) then
            call Print_Error
        end if
        exec sql whenever sqlerror continue
        exec sql close csr for readonly
        Execute_Select = rows
    end function ! Execute_Select
        !
        ! Function:     Print_Header
        ! Purpose:      A statement has just been described so set up
        !               the SQLDA for result processing. Print all the
        !               column names and allocate result variables for
        !               retrieving data. The result variables are chosen out
        !               of a global pool of numeric variables (integers,
        !               floats and 2-byte indicators) and a large character
        !               buffer. The SQLDATA and SQLIND fields are pointed
        !               at the addresses of the result variables.
        !   Returns:    TRUE (-1) if successfully set up the SQLDA for
        !               result variables,
        !               FALSE (0) if an error occurred.
        !

400     function byte Print_Header
            ! Declare global SQLDA record
            exec sql include sqlda
            common (sqlda_area) IISQLDA sqlda
            !
            ! Global result data storage. This area includes an array
            ! of numerics (integers, floats and indicator variables), as
            ! well as a large character buffer from which sub-strings are
            ! chosen for string retrieval.
            !
            declare word constant CHAR_MAX = 2500
            record num_store                    ! Pool of numeric variables
                long    int4
                double  flt8
                word    indicator
            end record num_store
            record char_store                   ! Pool of string data
                word buf_used
                string charbuf(CHAR_MAX) = 1
                end record char_store
            common (result_area) num_store nums(IISQ_MAX_COLS), &
                        char_store chars
            declare integer i                   ! Index into SQLVAR
            declare integer base_type           ! Base type w/o nullability
            declare byte nullable               ! Is column nullable
            declare integer ch_len              ! Required character length
            !
            ! Verify that there are enough result variables.
            ! If not print error and return.
            !
            if (sqlda::sqld > sqlda::sqln) then
                print 'SQL Error: SQLDA requires ' +                &
                        str$(sqlda::sqld) +                         &
                        ' variables, but has only ' +               &
                        str$(sqlda::sqln) + '.'
                        Print_Header = 0                    ! FALSE
```

```
            exit function
end if
!
! For each column print the number and title. For example:
!           [1] name [2] age [3] salary
! While processing each column determine the type of the
! column and to where SQLDATA and SQLIND must point in
!   order to retrieve type-compatible results. Note that the
! index into SQLVAR begins at 0 and not 1 because the
!   array is zero-based.
!
chars::buf_used = 1                          ! Nothing used yet
for i = 0 to sqlda::sqld - 1                 ! For each column
    ! Print column name and number
    print '[' + str$(i+1) + '] ' + &
            left$(sqlda::sqlvar(i)::sqlnamec, &
                    sqlda::sqlvar(i)::sqlnamel) + ' ';

!
! Process the column for type and length information. Use
! global result area from which variables can be allocated.
!

! Find the base-type of the result (non-nullable).
if (sqlda::sqlvar(i)::sqltype > 0) then
    base_type = sqlda::sqlvar(i)::sqltype
    nullable = 0                                     ! FALSE
else
    base_type = -sqlda::sqlvar(i)::sqltype
    nullable = -1 ! TRUE
end if
!
!    Collapse all different types into one of 4-byte integer,
! 8-byte floating-point, or fixed length character. Figure
! out where to point SQLDATA and SQLIND - which member of
! the global result storage area will retrieve the data.
!
select base_type
    case IISQ_INT_TYPE                       ! Use 4-byte integer
            sqlda::sqlvar(i)::sqltype = IISQ_INT_TYPE
            sqlda::sqlvar(i)::sqllen = 4
            sqlda::sqlvar(i)::sqldata = loc(nums(i)::int4)

    case IISQ_FLT_TYPE, IISQ_MNY_TYPE ! Use 8-byte float
            sqlda::sqlvar(i)::sqltype = IISQ_FLT_TYPE
            sqlda::sqlvar(i)::sqllen = 8
            sqlda::sqlvar(i)::sqldata = loc(nums(i)::flt8)

    case IISQ_CHA_TYPE, IISQ_VCH_TYPE, IISQ_DTE_TYPE
    !
    ! Determine the length of the sub-string required
    ! from the large character buffer. If we have enough
    ! space left then point at the start of the
    ! corresponding sub-string, otherwise print an
    !       error and return.
    !
    ! Note that for DATE types we must set the length.
    !
    if (base_type = IISQ_DTE_TYPE) then
            ch_len = IISQ_DTE_LEN
    else
            ch_len = sqlda::sqlvar(i)::sqllen
    end if
    if ((chars::buf_used + ch_len) > CHAR_MAX) then
            print 'SQL Error: Character data overflow.' +    &
                    ' Need more than '                +    &
```

```
                        str$(CHAR_MAX) + ' bytes.'
                Print_Header = 0                        ! FALSE
                exit function
            end if                          ! If too many characters
            !
            ! Grab space out of the large character buffer and
            ! keep track of the amount of space used so far.
            !
            sqlda::sqlvar(i)::sqltype = IISQ_CHA_TYPE
            sqlda::sqlvar(i)::sqllen = ch_len
            sqlda::sqlvar(i)::sqldata =                              &
                    loc(chars::charbuf(chars::buf_used))
            chars::buf_used = chars::buf_used + ch_len
        case else                                       ! Bad data type
            print 'SQL Error: Unknown data type returned: ' + &
                    str$(sqlda::sqlvar(i)::sqltype)
            Print_Header = 0                            ! FALSE
            exit function
        end select                              ! Of checking types
        ! If nullable then point at a null indicator and negate type id
        if (nullable) then
            sqlda::sqlvar(i)::sqltype = -sqlda::sqlvar(i)::sqltype
            sqlda::sqlvar(i)::sqlind = loc(nums(i)::indicator)
        else
            sqlda::sqlvar(i)::sqlind = 0
        end if
    next i                                  ! End of processing each column
    print ''                                ! Add separator line
    print '--------------------------------------'

    Print_header = -1                       ! TRUE
end function ! Print_Header
!
! Subroutine: Print_Row
! Purpose:  For each element inside the SQLDA, print the value.
!           Print its column number too in order to identify it with
!           the column name printed earlier. If the value is NULL
!           print 'N/A'. This routine prints the values using very
!           basic formats and does not try to tabulate the results.
!

500 sub Print_Row
        ! Declare global SQLDA record
        exec sql include sqlda
        common (sqlda_area) IISQLDA sqlda
        !
        ! Global result data storage. Variables from these
        !   pools were pointed at by the Print_Header routine.
        !
        declare word constant char_max = 2500
        record num_store                    ! Pool of numeric variables
            long    int4
            double  flt8
            word    indicator
        end record num_store
        record char_store                   ! Pool of string data
            word buf_used
            string charbuf(char_max) = 1
        end record char_store
        common (result_area) num_store nums(IISQ_MAX_COLS), &
                char_store chars
        declare integer i                   ! Index into SQLVAR
        declare integer ch                  ! Index for print characters
        declare integer ch_len              ! Required character length
        !
        ! For each column, print the column number and the data. The
```

```
            ! number identifies the column with the column name
            !   printed in Print_Header. NULL columns are
            !   printed as 'N/A'.
            !
            chars::buf_used = 1                  ! No characters printed yet
            for i = 0 to sqlda::sqld - 1         ! For each column
                print '[' + str$(i+1) + '] '; ! Print column number
                ! If nullable and is NULL then print 'N/A'
                if (sqlda::sqlvar(i)::sqltype > 0) and          &
                         (nums(i)::indicator = -1) then
                        print 'N/A';

                else
                        !
                        ! The type is either not nullable, or nullable
                        ! but not NULL. Print the result using very basic
                        ! output formats.
                        !
                        select abs(sqlda::sqlvar(i)::sqltype)

                        case IISQ_INT_TYPE
                                print str$(nums(i)::int4);

                        case IISQ_FLT_TYPE
                                ! This format may lose precision
                                print str$(nums(i)::flt8);

                        case IISQ_CHA_TYPE
                                !
                                ! Use a current-length sub-string from the large
                                ! character buffer, as allocated in Print_Header.
                                !
                                ch_len = sqlda::sqlvar(i)::sqllen
                                for ch = 0 to ch_len - 1
                                print chars::charbuf(chars::buf_used + ch);
                        next ch
                        chars::buf_used = chars::buf_used + ch_len
                end select                       ! Of different types
        end if                                   ! If null or not
            if (i < sqlda::sqld - 1) then        ! Add trailing space
                print ' ';
            end if
            next i                               ! End of each column
            print ''                             ! Print new line
        end sub                                  ! Print_Row
        !
        ! Subroutine: Print_Error
        ! Purpose:    SQLCA error detected. Retrieve the error message
        !             and print it.
        !

600 sub Print_Error
            exec sql include sqlca
            exec sql begin declare section
                declare string error_buf         ! For error text retrieval
            exec sql end declare section
            exec sql inquire_sql (:error_buf = errortext)
            print 'SQL Error:'
            print error_buf
        end sub                                  ! Print_Error
        !
        ! Function:     Read_Stmt
        ! Purpose:      Read a statement from standard input. This
        !               routine issues a prompt with the current statement
        !               number, and reads the statement from the screen into
        !               the parameter 'stmt_buf'. No special scanning is done
```

```
!               to look for terminators, string delimiters or line
!               continuations.
!
!               This routine can be extended to allow line
!               continuation, SQL-style comments, and a semicolon
!               terminator.
! Parameters:
!               stmt_num - Statement number for prompt.
!               stmt_buf - Input statement buffer.
! Returns:
!               TRUE (-1) - If a statement is typed in.
!               FALSE (0) - If end-of-file is typed in,
!                or an error occurred.
!

700 function byte Read_Stmt(integer stmt_num, string stmt_buf)

        declare byte was_input                          ! Return value
        stmt_buf = ''
        was_input = -1                                  ! TRUE
        ! Ignore empty lines and stop on error
        while (stmt_buf = '') and (was_input = -1)
            when error in
            print ' ' + str$(stmt_num);
            linput ' '; stmt_buf
            use
            was_input = 0                               ! FALSE
        end when
        next
    Read_Stmt = was_input

  end function                                  ! Read_Stmt
```

A Dynamic SQL/Forms Database Browser

This program lets the user browse data from and insert data into any table in any database, using a dynamically defined form. The program uses Dynamic SQL and Dynamic FRS statements to process the interactive data. You should already have used VIFRED to create a Default Form based on the database table that you want to browse. VIFRED will build a form with fields that have the same names and data types as the columns of the specified database table.

When run, the program prompts the user for the name of the database, the table and the form. The form is profiled using the **describe form** statement, and the field name, data type and length information is processed. From this information the program fills in the SQLDA data and null indicator areas, and builds two Dynamic SQL statement strings to **select** data from and **insert** data into the database.

The **Browse** menu item retrieves the data from the database using an SQL cursor associated with the dynamic **select** statement, and displays that data using the dynamic **putform** statement. A **submenu** allows you to continue with the next row or return to the main menu. The **Insert** menu item retrieves the data from the form using the dynamic **getform** statement, and adds the data to the database table using a prepared **insert** statement. The **Save** menu item commits your changes and, because prepared statements are discarded, re-prepares the **select** and **insert** statements. When the **Quit** menu item is selected, all pending changes are rolled back and the program is terminated.

```
100     !
        ! Program: Dynamic_FRS
        ! Purpose: Main body of Dynamic SQL forms application. Prompt for
        !          database, form and table name. Call Describe_Form
        !          to obtain a profile of the form and set up the SQL
        !          statements. Then allow the user to interactively browse
        !          the database table and append new data.
        !

        program Dynamic_FRS
            ! Declare the global SQLCA and SQLDA records
            exec sql include sqlca
            exec sql include sqlda
            common (sqlda_area) IISQLDA sqlda
            exec sql declare sel_stmt statement  ! Dynamic SQL SELECT and
            exec sql declare ins_stmt statement  ! INSERT statements
            exec sql declare csr cursor
                for sel_stmt                     ! Cursor for dynamic SELECT
            external byte function
                Describe_Form                    ! DESCRIBE form/SQL statements
            exec sql begin declare section
                declare string  dbname           ! Database name
                declare string  formname         ! Form name
                declare string  tabname          ! Database table name
                declare string  sel_buf  ! Prepared SELECT statement
                declare string  ins_buf  ! Prepared INSERT statement
                declare integer er               ! Error status
                declare string  ret              ! Prompt error buffer
            exec sql end declare section
            exec frs forms
            ! Prompt for database name - will abort on errors
            exec sql whenever sqlerror stop
            exec frs prompt ('Database name: ', :dbname)
            exec sql connect :dbname
            exec sql whenever sqlerror call sqlprint
            !
            ! Prompt for table name - later a Dynamic SQL SELECT statement
            ! will be built from it.
            !
            exec frs prompt ('Table name: ', :tabname)

            !
            ! Prompt for form name. Check forms errors reported
            ! through INQUIRE_FRS.
            !
            exec frs prompt ('Form name: ', :formname)
            exec frs message 'Loading form ...'
            exec frs forminit :formname
            exec frs inquire_frs frs (:er = ERRORNO)
            if (er > 0) then
                exec frs message 'Could not load form. Exiting.'
                exec frs endforms
                exec sql disconnect
                exit program
            end if
            ! Commit any work done so far - access of forms catalogs
            exec sql commit
            ! Describe the form and build the SQL statement strings
            if (not Describe_Form
                (formname, tabname, sel_buf, ins_buf)) then
                exec frs message 'Could not describe form. Exiting.'
                exec frs endforms
                exec sql disconnect
                exit program
            end if
            !
```

```
! PREPARE the SELECT and INSERT statements that correspond
! to the menu items Browse and Insert. If the Save menu item
! is chose the statements are reprepared.
!
exec sql prepare sel_stmt from :sel_buf
er = sqlcode
exec sql prepare ins_stmt from :ins_buf
if ((er < 0) or (sqlcode < 0)) then
    exec frs message
            'Could not prepare SQL statements. Exiting.'
    exec frs endforms
    exec sql disconnect
    exit program
end if
!
! Display the form and interact with user, allowing browsing
! and the inserting of new data.
!

exec frs display :formname fill
exec frs initialize
exec frs activate menuitem 'Browse'
exec frs begin
    !
    ! Retrieve data and display the first row on the form,
    ! allowing the user to browse through successive rows. If
    ! data types from the database table are not consistent
    ! with data descriptions obtained from the form, a
    ! retrieval error will occur. Inform the user of this or
    ! other errors.
    !
    ! Note that the data will return sorted by the first
    ! field that was described, as the SELECT statement,
    ! sel_stmt, included an
    ! order by clause.
    !
    exec sql open csr
    ! Fetch and display each row
    while (sqlcode = 0)

            exec sql fetch csr using descriptor :sqlda
            if (sqlcode <> 0) then
                    exec sql close csr
                    exec frs prompt noecho ('No more rows :', :ret)
                    exec frs clear field all
                    exec frs resume
            end if
            exec frs putform :formname using descriptor :sqlda
            exec frs inquire_frs frs (:er = ERRORNO)
            if (er > 0) then
                    exec sql close csr
                    exec frs resume
            end if
            ! Display data before prompting user with submenu
            exec frs redisplay
            exec frs submenu
            exec frs activate menuitem 'Next', FRSKEY4
            exec frs begin
                    ! Continue with cursor loop
                    exec frs message 'Next row ...'
                    exec frs clear field all
            exec frs end
            exec frs activate menuitem 'End', FRSKEY3
            exec frs begin
                    exec sql close csr
                    exec frs clear field all
```

```
                        exec frs resume
                exec frs end
        next                    ! While there are more rows
    exec frs end
    exec frs activate menuitem 'Insert'
    exec frs begin
        exec frs getform :formname using descriptor :sqlda
        exec frs inquire_frs frs (:er = errorno)
        if (er > 0) then
                exec frs clear field all
                exec frs resume
        end if
        exec sql execute ins_stmt using descriptor :sqlda
        if ((sqlcode < 0) or (sqlerrd(2) = 0)) then
                exec frs prompt noecho ('No rows inserted :', :ret)
        else
                exec frs prompt noecho ('One row inserted :', :ret)
        end if
    exec frs end
    exec frs activate menuitem 'Save'
    exec frs begin
        !
        ! COMMIT any changes and then re-PREPARE the SELECT
        !       and INSERT statements as the COMMIT statements
        !       discards them.
        !
        exec sql commit
        exec sql prepare sel_stmt from :sel_buf
        er = sqlcode
        exec sql prepare ins_stmt from :ins_buf
        if ((er < 0) or (sqlcode < 0)) then
                exec frs prompt noecho                  &
                        ('Could not reprepare SQL statements :', :ret)
                exec frs breakdisplay
        end if
    exec frs end
    exec frs activate menuitem 'Clear'
        exec frs begin
                                exec frs clear field all
        exec frs end
        exec frs activate menuitem 'Quit', FRSKEY2
        exec frs begin
                exec sql rollback
                exec frs breakdisplay
                exec frs end
        exec frs finalize
        exec frs endforms
        exec sql disconnect
end program                                     ! Dynamic_FRS
!
! Function: Describe_Form
! Purpose:  Profile the specified form for name and data
!           type information.
!           Using the DESCRIBE FORM statement, the SQLDA is
!           loaded with field information from the form. This
!           procedure processes this information to allocate
!           result storage, point at storage for dynamic FRS
!           data retrieval and assignment, and build SQL
!           statements strings for subsequent dynamic SELECT and
!           INSERT statements. For example, assume the form
!           (and table) 'emp' has the following fields:
!
!               Field Name      Type            Nullable?
!               ----------      ----            ---------
!               name            char(10)        No
!               age             integer4        Yes
```

```
!               salary          money           Yes
!
!               Based on 'emp', this procedure will construct the
!               SQLDA.  The procedure allocates variables from a
!               result variable pool (integers, floats and a large
!               character string buffer).
!               The SQLDATA and SQLIND fields are pointed at
!               the addresses of the result variables in the pool.
!               The following SQLDA is built:
!
!               sqlvar(0)
!                       sqltype         = IISQ_CHA_TYPE
!                       sqllen          = 10
!                       sqldata         = pointer into characters array
!                       sqlind          = null
!                       sqlname         = 'name'
!               sqlvar(1)
!                       sqltype         = -IISQ_INT_TYPE
!                       sqllen          = 4
!                       sqldata         = address of integers(1)
!                       sqlind          = address of indicators(1)
!                       sqlname         = 'age'
!               sqlvar(2)
!                       sqltype         = -IISQ_FLT_TYPE
!                       sqllen          = 8
!                       sqldata         = address of floats(2)
!                       sqlind          = address of indicators(2)
!                       sqlname         = 'salary'
!
!               This procedure also builds two dynamic SQL statements
!               strings. Note that the procedure should be extended
!               to verify that the statement strings do fit into the
!               statement buffers (this was not done in this
!               example). The above example would construct the
!               following statement strings:
!
!               'SELECT name, age, salary FROM emp ORDER BY name'
!               'INSERT INTO emp (name, age, salary) VALUES (?, ?, ?)'
!
! Parameters:
!               formname - Name of form to profile.
!               tabname - Name of database table.
!               sel_buf - Buffer to hold SELECT statement string.
!               ins_buf - Buffer to hold INSERT statement string.
! Returns:
!               TRUE (-1) - Success/failure - will fail on error
!               FALSE (0) or upon finding a table field.
!

200     function byte Describe_Form
                (string formname, tabname, sel_buf, ins_buf)

        ! Declare the global SQLCA and SQLDA records
        exec sql include sqlca
        exec sql include sqlda
        common (sqlda_area) IISQLDA sqlda
        !
        ! Global result data storage pool for integer data, floating-point
        ! data, indicator variables, and character data. The character
        ! data is a large buffer from which sub-strings are chosen.
        !
        declare word constant CHAR_MAX = 2500
        common (result_area) integer integers(IISQ_MAX_COLS),           &
                 double floats(IISQ_MAX_COLS),                          &
                 word indicators(IISQ_MAX_COLS),                        &
                 string characters(CHAR_MAX) = 1
```

```
declare integer char_cnt                ! Character counter
declare integer char_cur                ! Current character length
declare integer i                       ! Index into SQLVAR
declare integer base_type               ! Base type w/o nullability
declare byte nullable                   ! Is nullable (SQLTYPE < 0)

declare string names                    ! Names for SQL statements
declare string name_cur                 ! Current column name
declare string marks                    ! Place holders for INSERT
declare integer er                      ! Error status
declare string ret                      ! Prompt error buffer
!
! Initialize the SQLDA and DESCRIBE the form. If we cannot fully
! describe the form (our SQLDA is too small) then report an error
! and return.
!
sqlda::sqln = IISQ_MAX_COLS
exec frs describe form :formname all into :sqlda
exec frs inquire_frs frs (:er = errorno)
if (er > 0) then
    Describe_Form = 0           ! Error already displayed
    exit function
end if
if (sqlda::sqld > sqlda::sqln) then
    exec frs prompt noecho ('SQLDA is too small for form :', :ret)
    Describe_Form = 0
    exit function
end if
if (sqlda::sqld = 0) then       ! No fields
        exec frs prompt noecho
                ('There are no fields in the form :', :ret)
                Describe_Form = 0
                exit function
end if
!
! For each field determine the size and type of the result data
! area. This data area will be allocated out of the result
! variable pool (integers, floats and characters) and will be
! pointed at by SQLDATA and SQLIND. Note that the index into
! SQLVAR begins at 0 and not 1 because the array is zero-based.
!
! If a table field type is returned then issue an error.
!
! Also, for each field add the field name to the 'names' buffer
! and the SQL place holders '?' to the 'marks' buffer, which
! will be used to build the final SELECT and INSERT statements.
!

char_cnt = 1
for i = 0 to sqlda::sqld - 1            ! For each column
    ! Find the base-type of the result (non-nullable).
    if (sqlda::sqlvar(i)::sqltype > 0) then
        base_type = sqlda::sqlvar(i)::sqltype
        nullable = 0                    ! False
    else
        base_type = -sqlda::sqlvar(i)::sqltype
        nullable = -1 ! True
    end if
    !
    ! Collapse all different types into one of 4-byte integer,
    ! 8-byte floating-point, or fixed length character. Figure
    ! out where to point SQLDATA and SQLIND - which member
    ! of the result variable pool is compatible with the data.
    !
    select base_type
        case IISQ_INT_TYPE              ! Use 4-byte integer
```

```
                sqlda::sqlvar(i)::sqltype = IISQ_INT_TYPE
                sqlda::sqlvar(i)::sqllen = 4
                sqlda::sqlvar(i)::sqldata = loc(integers(i))

        case IISQ_FLT_TYPE, IISQ_MNY_TYPE ! Use 8-byte float
                sqlda::sqlvar(i)::sqltype = IISQ_FLT_TYPE
                sqlda::sqlvar(i)::sqllen = 8
                sqlda::sqlvar(i)::sqldata = loc(floats(i))

        case IISQ_CHA_TYPE, IISQ_VCH_TYPE, IISQ_DTE_TYPE
        !
        ! Determine the length of the sub-string required
        ! from the large character buffer. If we have enough
        ! space left then point at the start of the corresponding
        ! sub-string, otherwise print an error and return.
        !
        ! Note that for DATE types we must set the length.
        !
        if (base_type = IISQ_DTE_TYPE) then
                char_cur = IISQ_DTE_LEN
        else
                char_cur = sqlda::sqlvar(i)::sqllen
        end if
        if ((char_cnt + char_cur) > CHAR_MAX) then
                exec frs prompt noecho                          &
                        ('Character pool buffer overflow :', :ret)
                Describe_Form = 0
        exit function
        end if                          ! If too many characters
        !
        ! Grab space out of the large character buffer and
        ! keep track of the amount of space used so far.
        !
        sqlda::sqlvar(i)::sqltype = IISQ_CHA_TYPE
        sqlda::sqlvar(i)::sqllen  = char_cur
        sqlda::sqlvar(i)::sqldata = loc(characters(char_cnt))
        char_cnt                  = char_cnt + char_cur
    case IISQ_TBL_TYPE                          ! Table field
        exec frs prompt noecho                                  &
                ('Table field found in form :', :ret)
        Describe_Form = 0
        exit function
    case else                                   ! Bad data type
        exec frs prompt noecho ('Invalid field type :', :ret)
        Describe_Form = 0
        exit function
end select                                      ! Of checking types
! If nullable then point at a null indicator and negate type id
if (nullable) then
    sqlda::sqlvar(i)::sqlind  = loc(indicators(i))
    sqlda::sqlvar(i)::sqltype = -sqlda::sqlvar(i)::sqltype
else
    sqlda::sqlvar(i)::sqlind  = 0
end if
!
! Store field names and place holders (separated by commas)
! for the SQL statements.
!
name_cur =                                                      &
    left$(sqlda::sqlvar(i)::sqlnamec, sqlda::sqlvar(i)::sqlnamel)
if (i = 0) then
    names = name_cur
    marks = '?'
else
    names = names + ',' + name_cur
    marks = marks + ',?'
```

```
end if
next i                                   ! End of column processing
!
! Create final SELECT and INSERT statements. For the SELECT
! statement ORDER BY the first field.
!
name_cur =                                                        &
    left$(sqlda::sqlvar(0)::sqlnamec, sqlda::sqlvar(0)::sqlnamel)
sel_buf = 'select ' + names + ' from ' + tabname &
                + ' order by ' + name_cur
ins_buf = 'insert into ' + tabname + ' (' + names &
                + ') values (' + marks + ')'

Describe_Form = -1                               ! True


end function                                     ! Describe_Form
```

Chapter 7: Embedded SQL for Pascal

This chapter describes the use of Embedded SQL with the Pascal programming language.

Embedded SQL Statement Syntax for Pascal

This section describes the language-specific issues inherent in embedding SQL database and forms statements in a Pascal program. An Embedded SQL database statement has the following general syntax:

[*margin*] **exec sql** *SQL_statement terminator*

The syntax of an Embedded SQL/FORMS statement is almost identical:

[*margin*] **exec frs** *SQL/FORMS_statement terminator*

For information on SQL statements, see the *SQL Reference Guide*. For information on SQL/FORMS statements, see the *Forms-based Application Development Tools User Guide*.

The sections below describe the various syntactical elements of these statements as implemented in Pascal.

Margin

There are no specified margins for Embedded SQL statements in Pascal. The **exec** keyword can begin anywhere on the source line. It can be preceded only by white space (blanks and tabs) and/or a label.

Terminator

The terminator for Pascal is the semicolon (;). For example, a **select** statement embedded in a Pascal program would look like:

```
exec sql select ename
         into :namevar
         from employee
         where eno = :numvar;
```

An embedded statement cannot be followed on the same line by another embedded statement or a Pascal statement. Doing so will cause preprocessor syntax errors on the second statement. Following the Pascal terminator, only comments and white space are allowed to the end of the line.

Even though some Pascal statements, such as the last statement before a Pascal **else** clause, do not allow a semicolon, Embedded SQL requires the semicolon. For more details on this and other coding requirements, see Advanced Processing in this chapter.

Labels

Like Pascal statements, Embedded SQL statements can have a label prefix. The label must begin with a digit, an alphabetic character, or an underscore, must be the first word on the line (optionally preceded by white space), and must be terminated with a colon. For example:

```
close_cursor: exec sql close cursor1;
```

The label can appear anywhere a Pascal label can appear. As in standard Pascal, the label must be declared before it is used. This declaration must occur outside any Embedded SQL declaration section. Even though the preprocessor will accept a label in front of any **exec sql** or **exec frs** prefix, it may not be appropriate to code a label on some lines. For example, the following, although acceptable to the preprocessor, causes a compiler error because labels are not allowed before declarations:

```
include_sqlca: exec sql include sqlca;
```

As a general rule, use labels only with executable statements.

Line Continuation

There are no line continuation rules for Embedded SQL statements in Pascal. Statements can continue across multiple lines, extending to the Pascal terminator. Blank lines can be included in a statement.

Comments

Embedded SQL/Pascal comments can be either of the two standard Pascal comments, delimited by "(*" and "*)" or by "{" and "}". For example:

```
exec frs message 'No permission ...';(*No user access *)
exec frs sleep 2; { Let the user read it }
```

Note that you cannot mix delimiters: a comment starting with "{" must end with "}" and not with "*)". You cannot nest comments, but you can extend them over multiple lines. As a convention, comments in this document will normally be delimited by "{" and "}".

You can include an Embedded SQL/Pascal comment anywhere in an Embedded SQL statement that a blank is allowed, with the following exceptions:

- Between the margin and the word **exec** (whether or not you have a Pascal label prefix).
- Between the word **exec** and the word **sql** or **frs**. In the following example, comments cause both statements to be interpreted as Pascal host code:

```
{ Initial comment } exec sql include sqlca;
 exec { Between } sql help employee;
```

- Between words that are reserved when they appear together. For a list of these double reserved words, see the list of Embedded SQL keywords in the *SQL Reference Guide*.
- In string constants.
- In parts of statements that are dynamically defined. For example, a comment in a string variable specifying a form name is interpreted as part of the form name.
- Between component lines of Embedded SQL/FORMS block-type statements. All block-type statements (such as **activate** and **unloadtable**) are compound statements that include a statement section delimited by **begin** and **end**. Comment lines must not appear between the statement and its section. The preprocessor would interpret such comments as Pascal host code and generate preprocessor syntax errors. (Note, however, that comments can appear *on the same line* as the statement.) For example, the following statement would cause a syntax error on the Pascal comment:

```
exec frs unloadtable empform
            employee (:namevar = ename);
{Illegal comment before statement body}
exec frs begin; {Comment legal here}
            msgbug := namevar;
exec frs end;
```

- Statements made up of more than one compound statement, such as the **display** statement, which typically consists of the **display** clause, an **initialize** section, **activate** sections and a **finalize** section, cannot have Pascal comments between any of the components. These comments would be translated as host code and would cause syntax errors on subsequent statement components.

You can also use the SQL comment delimiter "--". Everything between this delimiter and the end of the line is considered a comment. For example:

```
exec sql delete -- Delete all employees
     from employee;
```

Note: Because Pascal assumes that "(*" is the beginning of a comment, when you want to use the aggregate function, count, to count the number of rows in a table, that is count (*), you must put a space between the left parenthesis and the asterisk, count (*).

String Literals

Embedded SQL string literals are delimited by single quotes. To embed a single quote in a string literal you should double it, as in:

```
exec sql insert
        into people (age, surname)
        values (15, 'O''Hara');
```

String literals cannot be continued over multiple lines.

String Literals and Statement Strings

The Dynamic SQL statements **prepare** and **execute immediate** both use statement strings, which specify an SQL statement. The statement string can be specified by a string literal or character string variable, as in:

```
exec sql execute immediate 'drop employee';
str = 'drop employee';
exec sql execute immediate :str;
```

As with regular Embedded SQL string literals, the statement string delimiter is the single quote. However, quotes embedded in statement strings must conform to the runtime rules of SQL when the statement is executed. Notice the doubling of the single quote in the following Dynamic **insert** statement.

```
exec sql prepare s1 from
   'Insert into t1 values (''single''''double" '')';
```

The runtime evaluation of the above statement string is:

```
Insert into t1 values ('single''double" ')
```

The Create Procedure Statement

As mentioned in the *SQL Reference Guide*, the **create procedure** statement has language-specific syntax rules for line continuation, string literal continuation, comments, and the final terminator. These syntax rules follow the rules discussed in this section. For example, the final terminator is a semicolon. Although the preprocessor treats the **create procedure** statement as a single statement, all statements *in* the body of the procedure are terminated with a semicolon as is an Embedded SQL/Pascal statement.

The following example shows a **create procedure** statement that follows the Embedded SQL/Pascal syntax rules:

```
exec sql
         create procedure proc (parm integer) as
         declare
                var integer;
         begin
                if parm > 10
                then { use pascal comment delimiters }
                         message 'pascal strings cannot
```

```
                        continue over lines';
                        insert into tab values (:parm);
                endif;
        end;
```

Decimal Literals

The preprocessor distinguishes between decimal and floating-point literals in SQL and Forms Runtime System (FRS) statements according to the following rules:

- A literal containing a decimal point with no E notation is a decimal literal.
- A literal with E notation is a floating-point literal.

For example:

```
exec sql insert
     into mytable (salary) values (23000.12)
exec sql insert
     into mytable (number) values (1.4E4)
```

A numeric literal with or without the E notation is treated as a float if it is in the host declaration section.

In addition, the preprocessor treats integer literals greater than MAXINT as decimals. This allows host programs to input large integer values.

Ingres will treat '23000.00' as a decimal literal and '1.4E2' as a float literal.

However, applications will continue to use host language rules for interpreting literals appearing in host declarations. For example:

```
exec sql begin declare section
    integer 2 i (1.234)
exec sql end declare section
```

The literal '1.234' is interpreted according to the Pascal compiler rules.

This is consistent with the Ingres convention of interpreting SQL statements according to SQL rules and host statements according to host language compiler rules.

Pascal Variables and Data Types

This section describes how to declare and use Pascal program variables in Embedded SQL.

Embedded SQL/Pascal Declarations

The following sections describe SQL/Pascal declarations.

Embedded SQL Variable Declaration Sections

Embedded SQL statements use Pascal variables to transfer data from the database or a form into the program and *vice versa*. You must declare Pascal variables and constants to Embedded SQL before using them in any Embedded SQL statements. Pascal variables, types, and constants are declared to Embedded SQL in a *declaration section*. This section has the following syntax:

exec sql begin declare section;
Pascal constant, type and variable declarations
exec sql end declare section;

Note that placing a label in front of the **exec sql end declare section** statement causes a preprocessor syntax error.

Embedded SQL variable declarations are global to the program file from the point of declaration onwards. Multiple declaration sections can be incorporated into a single program, as would be the case when a few different Pascal procedures issue embedded statements using local variables. Each procedure can have its own declaration section. For more information on the declaration of variables that are local to Pascal procedures, see The Scope of Objects in this chapter.

Reserved Words in Declarations

All Embedded SQL keywords are reserved. Therefore, you cannot declare variables with the same names as ESQL keywords. You can only use them in quoted string literals. These words are:

array	**file**	**packed**	**ref**	**varying**
case	**function**	**procedure**	**static**	
const	**label**	**range**	**type**	
def	**otherwise**	**record**	**var**	

Note that not all Pascal compilers reserve every keyword listed. However, the Embedded SQL/Pascal preprocessor does reserve all these words.

Data Types and Constants

The Embedded SQL/Pascal preprocessor accepts the data types that are shown in the following table. The table maps these types to their corresponding Ingres type categories. For a description of the exact type mapping, see Data Type Conversion in this chapter.

Pascal Data Types and Corresponding Ingres Types

Pascal Type	Ingres Type
boolean	integer
integer	integer
unsigned	integer
real	float
single	float
double	float
char	character
indicator	indicator
real	decimal

Your program should not redefine any of the above types.

The table below maps the Pascal constants to their corresponding Ingres type categories.

Constants and Corresponding Ingres Types

Pascal Constant	Ingres Type
maxint	**integer**
true	**integer**
false	**integer**

The Integer Data Types

Several Pascal types are considered as integer type by the preprocessor as shown in the following table.

The Integer Data Types

Description	Example
integer	Integer
4-byte subrange of integer	1..127
2-byte subrange of integer	[word] 0..32767
1-byte subrange of integer	[byte] 0..63
enumeration	(red, blue, green)
boolean	Boolean

The preprocessor can accept all **integer** types. Even though some integer types have Pascal constraints, such as the subranges and enumerations, Embedded SQL does not check these constraints, either during preprocessing or at runtime.

The type **boolean** is handled as a special type of **integer**. Embedded SQL treats the **boolean** type as an enumerated type and generates the correct code in order to use this type to interact with an Ingres integer. Enumerated types are described in more detail later.

The Indicator Type

An *indicator type* is a 2-byte integer type. There are three ways to use indicator types in an application:

- In a statement that retrieves data from Ingres, you can use an indicator type to determine if its associated host variable was assigned a null.
- In a statement that sets data to Ingres, you can use an indicator type to assign a null to the database column, form field, or table field column.
- In a statement that retrieves character data from Ingres, you can use the indicator type as a check that the associated host variable was large enough to hold the full length of the returned character string.

Embedded SQL/Pascal predefines the 2-byte integer type **indicator**. As with other types, you should not redefine the **indicator** type. This type definition is in the file that is included when preprocessing the Embedded SQL statement **include sqlca**. The type declaration syntax is:

```
type
        Indicator = [word] -32768..32767;
```

Because the type definition is in the referenced **include** file, you can only declare variables of type **indicator** after you have issued **include sqlca**. This declaration does not preclude you from declaring indicator variables of other 2-byte integer types.

The Floating-Point Data Types

The preprocessor accepts three floating-point types. These are **single** and **real**, which are 4-byte floating-point types, and **double**, which is the 8-byte floating-point type. Note that, although the preprocessor accepts **quadruple** data type declarations, it does not accept references to variables of type **quadruple**. For more information, see Record Type Definition in this chapter.

The Double Storage Format

Embedded SQL requires that the storage representation for double variables be d_floating, because the Embedded SQL runtime system uses that format for floating-point conversions. If your Embedded SQL program has double variables that interact with the Embedded SQL runtime system, you must make sure they are stored in the d_floating format. Because the default Pascal format is d_floating, your program will automatically use the correct storage representation unless you use the g_floating compiler option. Any module compiled with this option must not use double variables or **float** literals to interact with Ingres. **Float** literals are treated as double precision numbers by Ingres. Note that Embedded SQL recognizes only single, and not double or quadruple, exponential notation for real constants. Thus, any real constants passed to Ingres are always single precision and are unaffected by the g_floating compiler option.

The Character Data Types

Three Pascal data types are compatible with Ingres string objects: **char**, **packed array of char**, and **varying of char**. Note that literal string constants are of type **packed array of char**. Embedded SQL allows only regular Pascal string literals: sequences of printing characters enclosed in single quotes. The VMS Pascal extensions of parenthesized string constructors and of nonprinting characters represented by their ASCII values in parentheses are not allowed.

The **char** data type does have some restrictions. Because of the mechanism used to pass string-valued arguments to the Embedded SQL runtime library, you cannot use a member of a **packed array of char** or **varying of char** to interact with Ingres. Also, a plain **array of char** (that is, not **packed** or **varying**) is not compatible with Ingres string objects; an element of such an array, however, is a **char** and as such *is* compatible.

For example, given the following legal declarations:

```
exec sql begin declare section;
type
    Alpha = 'a'..'z';                {1 character}
    Packed_6 = packed array[1..6]
               of Char;             {6-char string}
    Vary_6 = varying[6] of Alpha;   {6-char string}
    Array_6 = array[1..6]
```

```
                of Char;          {1-dimensional array}


var
    letter: Alpha; {1 character}
    p_str_arr: array[1..5]
               of Packed_6;       {Array of strings}
    chr_arr: array[1..6]
               of Char;           {1-dimensional array}
    two_arr: array[1..5]
               of Array_6;        {2-dimensional array of char}
    v_string : Vary_6;            {String}
exec sql end declare section;
```

these usages are legal:

```
exec frs message letter;          {A char is a string}
exec frs message chr_arr[3];      {A char is a string}
exec frs message two_arr[2][5];   {A char is a string}
exec frs message v_string;        {A varying array is a string}
exec frs message p_str_arr[2];
                                  {A packed array is a string}
```

but these usages are illegal:

```
exec frs message
            chr_arr;          {An array of chars is not a string}
exec frs message
            v_string[2];      {Cannot index a varying array}
exec frs message
            p_str_arr[2][3]; {Cannot index a packed array}
```

Declaration Syntax

This section describes the syntax for variable, type, and constant declarations. It also describes how to declare labels.

Attributes

In type definitions, Embedded SQL allows VMS Pascal attributes both at the beginning of the definition and just before the type name. The only attributes the preprocessor recognizes in type definitions are **byte**, **word**, and **long**. The preprocessor ignores any optional storage unit constant "(*n*)" appearing with the attribute. The preprocessor also ignores all other attributes, although it allows them.

The following example shows how to use the **byte** attribute in order to convert a 4-byte integer subrange into a 1-byte variable.

```
exec sql begin declare section;
var
            v_i1 : [byte] -128..127;
exec sql end declare section;
```

Note that Pascal requires that a size attribute be at least as large as the size of its type. Therefore, the following declaration would be illegal, because 400 will not fit into one byte:

```
exec sql begin declare section;
var
            v_i1 : [byte] 0..400;
exec sql end declare section;
```

Embedded SQL/Pascal does not allow explicit attribute size conflicts, as, for example:

```
exec sql begin declare section;
type
        i1 = [byte] -128..127; {i1 is a 1-byte integer type}
var
        v_i2 : [word] i1; {i1 cannot be extended to 2 bytes}
exec sql end declare section;
```

Label Declarations

An Embedded SQL block-structured statement is a statement delimited by the **begin** and **end** clauses. The **select** loop and the forms statements **display**, **unloadtable**, **submenu**, **formdata**, and **tabledata** are examples of these block-structured statements. All these statements generate Pascal labels in order to handle the complex control flow implicit in the statement. Because Pascal requires that all labels be declared before their use, Embedded SQL/Pascal requires that you issue an **exec sql label** statement in the Pascal declaration section of every routine (program, procedure, or function) that issues one of these statements. You must also end the routine with the Embedded SQL **exec sql end** statement, rather than the Pascal **end** statement, so that the preprocessor will know the scope of the label declaration.

The syntax for a label declaration is:

> **exec sql label** [*label_name* {, *label_name*}*];*
> ...
>
> **exec sql end ; | .**

Syntax Notes:

1. You can use **exec frs** and **exec sql** interchangeably with the Embedded SQL **label** and **end** statements.
2. The preprocessor ignores *label_name*s, except that they will appear in the generated Pascal **label** statement.
3. The terminating semicolon of the Embedded SQL **label** statement is required, even if there are no *label_name*s.
4. Only one Embedded SQL **label** statement can occur in each routine.

5. Each Embedded SQL **label** statement must have a matching Embedded SQL **end** statement. This **exec sql end** statement replaces the Pascal **end** statement and can be terminated with a semicolon or a period.
6. The **label** statement must appear in a Pascal declaration section, and *not* in an Embedded SQL **declare** section.

The following example illustrates the use of label declarations:

```
procedure Unload_Table;
exec frs label; {Must include this statement or exec sql
                  label,because unloadtable uses labels}
exec sql begin declare section;
var
        age : integer;
exec sql end declare section;
begin {unload_table}
    exec frs unloadtable 'form' 'table' (:age = emp_age);
    exec frs begin;
         ...
    exec frs end;
exec frs end; {Unload_Table}
```

Constant Declarations

The syntax for a constant declaration is:

> **const** *constant_name* = *constant_expr;*
> *{constant_name* = *constant_expr;}*

where a *constant_expr* is one of the following:

> [**+**|**-**] *constant_number*
> [**+**|**-**] *constant_name*
> *string_constant*

Constants can be used to set Ingres values but cannot be assigned values from Ingres.

Syntax Notes:

1. A *constant_name* must be a legal Pascal identifier beginning with an underscore or alphabetic character.
2. A *constant_number* can be either an integer or real number.
3. A variable or type name must begin with an alphabetic character, which can be followed by alphanumeric characters or underscores.
4. Embedded SQL/Pascal recognizes only **single**, and not **double** or **quadruple**, exponential notation for constants of type **real**.
5. The type of a *constant_name* is determined from the type of its *constant_expr*.

6. If a "+" or a "-" precedes a *constant_name* that is used as a *constant_expr*, the *constant_name* must be numeric.
7. Embedded SQL/Pascal does not support the declaration of arbitrary constant expressions.

The following example illustrates the use of constants declarations:

```
exec sql begin declare section;
const
            min_sal     = 15000.00;     {Real}
            pi          = 3.14159;      {Real}
            max_emps    = +99;          {Integer}
            max_credit  = 100000.00;    {Real}
            max_debt    = -max_credit;  {Real}
            yes         = 'y';          {Char}
exec sql end declare section;
```

Type Declarations

An Embedded SQL/Pascal type declaration has the following syntax:

type *type_name = type_definition;*
{type_name = type_definition;}

where *type_definition* is any of the following:

Syntax	Category
type_name	renaming
(enum_identifier {,enum_identifier})	enumeration
***[*+**|**-**] *constant* **..** [**+**|**-**] *constant*	numeric or character subrange
^type_name	pointer
varying **[***upper_bound***]** **of***char_type_name*	varying length string
[*packed] array* **[***dimensions]* **of** type_definition	array
record *field_list* **end**	record
file of *type_definition*	file
set of *type_definition*	set

Each of these type definitions is discussed in its own section below. All type names must be legal Pascal identifiers beginning with an alphabetic or underscore character.

Renaming Type Definition

The declaration for the renaming of a type uses the following syntax:

type *new_type_name* = *type_name;*

Syntax Notes:

1. The *type_name* must be either an Embedded SQL/Pascal type or a type name already declared to Embedded SQL (such as **Integer** or **Real**).
2. The *new_type_name* cannot be **Integer**, **Real** or **Char** or any other type listed at the beginning of this section.

The following example illustrates how to use this declaration:

```
exec sql begin declare section;
type
        NaturalInt = Integer;       {A "natural" sized integer}
exec sql end declare section;
```

Enumeration Type Definition

The declaration for an enumeration type definition has the following syntax:

type *type_name* = (*enum_identifier {, enum_identifier});*

Syntax Notes:

1. An *enum_identifier* must be a legal Pascal identifier beginning with an alphabetic or underscore character.
2. The *enum_identifiers* are treated as 4-byte integer constant identifiers.
3. The *type_name* maps to a 1-byte integer if there are fewer than 257 enumerated identifiers. Otherwise, it maps to a 2-byte integer.
4. When using an enumerated identifier as a value in an Embedded SQL statement, only the ordinal position of the identifier in the original enumerated list is important. In assigning a value to a variable of enumeration type, Embedded SQL passes the variable by address and assumes that the value is a legal one for the variable.

The following example illustrates the use of this declaration:

```
exec sql begin declare section;
type
        Table_Field_States =
                (undefined, newrow, unchanged, changed, deleted);
exec sql end declare section;
```

Subrange Type Definition

The syntax for declaring a subrange type definition is either:

> **type** *type_name* **=** [**+**|**-**]*integer_const* **..** [**+**|**-**]*integer_const***;**

or

> **type** *type_name = string_const .. string_const;*

Syntax Notes:

1. An *integer_const* can be either an integer literal or a named integer constant.
2. A *string_const* must be either a string literal or the name of a string constant. Although the preprocessor accepts any length string constant, the compiler requires the constant to be a single character.

The following example illustrates the use of this declaration:

```
exec sql begin declare section;
type
     alpha = 'a' .. 'z';
     months = 1 .. 12;
     minmax = -value .. value; {"value" is an integer constant}
     updated_states = changed .. deleted; {from previous example}
exec sql end declare section;
```

Pointer Type Definition

The declaration for a pointer type definition has the following syntax:

> **type** *pointer_name* = ^*type_name***;**

Syntax Notes:

The *type_name* can be either a previously defined type, or a type not yet defined. If the type has not yet been defined, the pointer type definition is a *forward pointer* definition. In that case, Embedded SQL requires that you define the *type_name* before using a variable of type *pointer_name* in an Embedded SQL statement.

The following example illustrates the use of this declaration:

```
exec sql begin declare section;
type
        empptr = ^emprecord;     {forward pointer declaration}
        emprecord = record
                e_name           : varying[40] of char;
                e_salary         : real;
                e_id             : integer;
                e_next           : empptr;
        end;
var
```

```
        empnode = empptr;
exec sql end declare section;
        ...

exec sql select name, salary, id
        into    :empnode^.e_name,
                :empnode^.e_salary,
                :empnode^.e_id
        from emp;
```

Varying Length String Type Definition

The declaration for a varying length string type definition has the following syntax:

> **type** *varying_type_name* **=** ***varying*** [*upper_bound*] **of** *char_type_name;*

Syntax Notes:

1. The *upper_bound* of a varying array specification is not parsed by the Embedded SQL preprocessor. Consequently, an illegal upper bound (such as a non-numeric expression) will be accepted by the preprocessor but will later cause Pascal compiler errors. For example, both of the following type declarations are accepted, even though only the first is legal in Pascal:

   ```
   exec sql begin declare section;
   type
           string20    = varying[20] of char;
           what        = varying['upperbound'] of char;
   exec sql end declare section;
   ```

2. Embedded SQL/Pascal treats a variable of type **varying of char** as a string, not an array.

The following example illustrates the use of this declaration:

```
exec sql begin declare section;
type
        pname = varying[100] of char;
var
        user_name : pname;
exec sql end declare section;
        ...
exec sql insert into person (name)
        values (:user_name);
```

Array Type Definition

The declaration for an array type definition has the following syntax:

> **type** *type_name* **=** [**packed**] **array** [*dimensions*] **of** *type_definition;*

Syntax Notes:

1. The *dimensions* of an array specification are not parsed by the Embedded SQL preprocessor. Consequently, an illegal dimension (such as a non-numeric expression) will be accepted by the preprocessor but will later cause Pascal compiler errors. For example, both of the type declarations shown below are accepted, even though only the first is legal in Pascal.

   ```
   exec sql begin declare section;
   type
           square      = array[1..10, 1..10] of integer;
           what        = array['dimensions'] of real;
   exec sql end declare section;
   ```

   The preprocessor only verifies that an array variable is followed by brackets when used (except **packed array of char**—see below).

2. ESQL/Pascal treats a variable of type **packed array of char** as a string, not an array. Thus, it is not followed by brackets when used.

3. Components of a **packed array** cannot be passed to the Embedded SQL runtime routines. Therefore, you should not declare **packed arrays** to Embedded SQL, except for **packed arrays of char**, which are passed as a whole (for example, as character strings).

The following example illustrates the use of the array type definition:

```
exec sql begin declare section;
type
         ssid = packed array [1..9] of char;
var
         user_ssid : ssid;
exec sql end declare section;
         ...

exec sql insert into person (ssno)
         values (:user_ssid);
```

Record Type Definition

The declaration for a record type definition has the following syntax:

> **type** *record_type_name* =
>   **record**
>   *field_list* [**;**]
>   **end;**

where *field_list* is:

> *field_element {; field_element}*
> [**case** [*tag_name* **:**] *type_name* ***of***
>   [*case_element {; case_element}]*
>   [**otherwise (** *field_list*)]]

where *field_element* is:

field_name {**,** *field_name*} : *type_definition*

and *case_element* is:

case_label **{,** *case_label*} : **(** *field_list* **)**

Syntax Notes:

1. All clauses of a record component have the same rules and restrictions as they do in a regular type declaration. For example, as with regular declarations, the preprocessor does not check dimensions for correctness.
2. In the **case** list, the *case_labels* can be numbers or names. Embedded SQL need not know the names.
3. ESQL/Pascal **record** declarations must be entirely contained in a **declaration** section; consequently all of the record components will be declared to the preprocessor. To minimize the effect of this restriction, the types **quadruple** and **set of** are allowed as legal types in an Embedded SQL record declaration. It is, however, an error to use variables of those types in Embedded SQL statements.
4. Components of a **packed** record cannot be passed to the runtime ESQL routines. Thus, do not declare **packed** records to ESQL.

The following example illustrates the use of the record type definition:

```
exec sql begin declare section;
type
         addressrec = record
                         street: packed array[1..30] of char;
                         town: packed array[1..10] of char;
                         zip: 1 .. 9999;
         end;

         employeerec = record
                         name:            packed array[1..20] of char;
                         age:             [byte] 0 .. 128;
                         salary:          real;
                         address:         addressrec;
                         checked:         boolean;
                         scale:           Quadruple;     {Cannot be used
                                                          by Embedded SQL}
        end;
exec sql end declare section;
```

File Type Definition

The declaration for a file type definition, has the following syntax:

type *type_name* **= file of** *type_definition;*

Syntax Notes:

1. A variable of type **file** can only be used with Embedded SQL through the file buffer. A file buffer for a given *type_definition* is referenced in the same manner as a pointer to the same type.
2. Components of a **packed** file cannot be passed to the Embedded SQL runtime routines. Do not declare **packed** files to ESQL.

The following example illustrates the use of the file type definition:

```
exec sql begin declare section;
var
        myfile : file of integer;
    exec sql end declare section;
        ...

get (myfile);
exec sql insert into emp (floor)
        values (:myfile^);
        ...

exec sql select floor
        into :myfile^;
        from emp;
put (myfile);
```

Set Type Definition

The declaration for a set type definition has the following syntax:

type *type_name* **= set of** *type_definition;*

Syntax Note:

Although the preprocessor accepts set definitions, no set variables can be used in Embedded SQL statements. As stated in the section on **record** declarations, **set** declarations are accepted only because all record components must be declared to Embedded SQL.

Variable Declarations

An Embedded SQL/Pascal variable declaration has the following syntax:

var *var_name {, var_name} : type_definition [:= initial_value];*
{var_name {, var_name} : type_definition [:= initial_value];}

Syntax Notes:

1. See the previous sections for information on the *type_definition*.

2. The *initial_value* is not parsed by the preprocessor. Consequently, any initial value is accepted, even if it may later cause a Pascal compiler error. Furthermore, the preprocessor accepts an initial value with any variable declaration, even where not allowed by the compiler. For example, both of the following initializations are accepted, even though only the first is legal in Pascal:

```
exec sql begin declare section;
var
         rowcount: integer := 1;
         msgbuf: packed array[1..100] of char := 2;
exec sql end declare section;
```

The following example illustrates the use of variable declarations:

```
exec sql begin declare section;
var
         rows, records:         0..500 := 0;
         was_error:             boolean;
         msgbuf:                varying[100] of char := ' ';
         operators:             array[1..6] of packed array[1..2] :=
                                ('= ', '!=', '< ', '> ', '<=', '>=');
         employees:             array[1..100] of employeerec;

         emp_ptr:               ^employeerec;
         work_days:             (mon, tue, wed, thu, fri);
         day_name:               varying[8] of char;
         random_ints:           file of integer;
         ind_set:                array[1...10] of indicator;
exec sql end declare section;
```

Formal Parameter Declarations

Most VAX/VMS Pascal formal parameter declarations are acceptable to Embedded SQL.

An Embedded SQL/Pascal formal parameter declaration has the following syntax:

formal_param_section {; *formal_param_section*}

where *formal_param_section* is:

formal_var | *formal_routine* [**:=** [*%mechanism*] *default_value*]

A *formal_var* has the syntax:

[**var** | *%mechanism] identifier {, identifier} : typename_or_schema*

where *typename_or_schema* is one of the following:

type_name
varying [*upper_bound_identifier*] **of** *type_name*
packed array [*schema_dimensions*] **of** *typename_or_schema*
array [*schema_dimensions* {**;** *schema_dimensions*}] **of**

typename_or_schema

where *schema_dimensions* is:

lower_bound_identifier *.. upper_bound_identifier :*
scalar_type_name

A *formal_routine* has the syntax:

*[%mechanism] routine*_header

where *routine_header* is one of the following:

procedure *proc_name ([formal_parameter_declaration*] **)**
function *func_name ([formal_parameter_declaration*] **)**
:*return_type_name*

In a subprogram declaration, the syntax of a formal parameter declaration is:

procedure *proc_name*
exec sql begin declare section;
(*formal_parameter_declaration* ***)***
exec sql end declare section;
;
...

or:

function *func_name*
exec sql begin declare section;
(*formal_parameter_declaration* ***)***
exec sql end declare section;
: *return_type_name**;***
...

Syntax Notes:

1. The Embedded SQL preprocessor ignores the names of procedures and functions used as formal parameters, but checks *their* formal parameters for legality.
2. The *default_value* is not parsed by the preprocessor. Consequently, any default value is accepted, even if it may later cause a Pascal compiler error. For example, both of the parameter default values shown below are accepted, even though only the first is legal in Pascal:

```
    procedure Load_table
    exec sql begin declare section;
         (clear_it: boolean := true;
            var is_error: boolean := 'false')
    exec sql end declare section;
                ;
                ...
```

3. Any *mechanism* specification is ignored.

The following example illustrates the use of these declarations:

```
    function Getesqlerror
exec sql begin declare section;
             ( buf : varying[ub] of char )
exec sql end declare section;
                        : boolean;

procedure Handleerror
exec sql begin declare section;
             ( procedure errorhandle(err : integer); var
                 errnum : integer )
exec sql end declare section;
                        ;

function Doappend
exec sql begin declare section;
             ( emp_id, floor : integer;
                 name : varying[ub] of char;
                 salary : real )
exec sql end declare section;
                                : integer;
```

The DCLGEN Utility

DCLGEN (Declaration Generator) is a record-generating utility that maps the columns of a database table into a record that can be included in a variable declaration. You invoke DCLGEN from the operating system level with the following command:

dclgen ***l****anguage dbname tablename filename recordname*

where

- *language* is the Embedded SQL host language, in this case, "pascal."
- *dbname* is the name of the database containing the table.
- *tablename* is the name of the database table.
- *filename* is the output file into which the record declaration is placed.
- *recordname* is the name of the Pascal record variable that the command creates. The command generates a record type definition named *recordname*, followed by "_rec." The command also generates a variable declaration for *recordname*.

This command creates the declaration file *filename*. The file contains a record type definition corresponding to the database table and a variable declaration of that record type. The file also includes a **declare table** statement that serves as a comment and identifies the database table and columns from which the record was generated.

After generating the file, you can use an Embedded SQL **include** statement to incorporate it into the variable declaration section. The following example demonstrates how to use DCLGEN in a Pascal program.

Assume the Employee table was created in the Personnel database as:

```
exec sql create table employee
            (eno        smallint not null,
             ename      char(20) not null,
             age        integer1,
             job        smallint,
             sal        decimal   not null,
             dept       smallint);
```

and the DCLGEN system-level command is:

```
dclgen pascal personnel employee employee.dcl emprec
```

The employee.dcl file created by this command contains a comment and three statements. The first statement is the **declare table** description of "employee," which serves as a comment. The second statement is a declaration of the Pascal record type definition "emprec_rec." The last statement is a declaration, using the "emprec_rec" type, for the record variable "emprec." The contents of the employee.dcl file are shown below.

```
{Description of table employee from database personnel}
exec sql declare employee TABLE
            (eno                smallint not null,
             ename              char(20) not null,
             age                integer1,
             job                smallint,
             sal                decimal  not null,
             dept               smallint);

type emprec_rec = record
            eno:                [word] -32768 .. 32767;
            ename:              packed array[1..20] of Char;
            age:                [byte] -128 .. 127;
            job:                [word] -32768 .. 32767;
            sal:                Double;
            dept:               [word] -32768 .. 32767;
end;
var emprec: emprec_rec;
```

This file should be included, by means of the Embedded SQL **include** statement, in an Embedded SQL declaration section:

```
exec sql begin declare section;
        exec sql include 'employee.dcl';
exec sql end declare section;
```

The emprec record can then be used in a **select**, **fetch**, or **insert** statement.

DCLGEN and Large Objects

You can use DCLGEN to generate an appropriate **declare table** statement with Ada variables for tables that contain **long varchar** columns. For columns that have a limited length, the variables generated will be identical to the variables generated for the Ingres **varchar** datatype. For columns with unlimited length, such as:

```
create table long_obj_table(blob_col long varchar);
```

DCLGEN will issue an error message and generate a character string variable with zero length. You can modify the length of the generated variable before attempting to use the variable in an application.

For example the following table definition:

```
create tablelongobj_table
        (long_column     long varchar));
```

results in the following DCLGEN generated output for Pascal compilers that support structures:

```
exec sql declare long_obj_table table
        (long_column             long varchar)

type blobs_rec_rec = record
        long_column : varying[0] of char;
end;
var blobs_rec : blobs_rec_rec;
```

Predeclared Identifiers

Embedded SQL predeclares all the standard Pascal types and constants in a scope enclosing the entire program (see Data Types and Constants in this chapter). You should not redefine any of these identifiers, because the runtime library expects the standard definitions.

Program Syntax

The syntax for an Embedded SQL/Pascal program definition is:

> **program** *program_name [(identifier {, identifier}***)**]**;**
> [**exec sql begin declare section;**
> *declarations*
> **exec sql end declare section;**]
> [*procedures, functions, etc.*]
> **begin**
>   [*statements*]
> **end.**

or:

program *program_name [(identifier {, identifier}* **)**]**;**
exec sql label [*label_declarations*]**;**
[**exec sql begin declare section;**
declarations
exec sql end declare section;]
[*procedures, functions, etc.*]
begin
[*statements*]
exec sql end.

where *declarations* can include any of the following:

const *constant_declarations*
type *type_declarations*
var *variable_declarations*

See the previous sections for descriptions of the various types of declarations.

Syntax Notes:

1. The *program_name* and the *identifiers* are not processed by ESQL.
2. The declaration sections can be in any order and can be repeated.

The following example illustrates the above points:

```
program Test;
exec sql label;
exec sql begin declare section;
var
    curformname, curfieldname, curcolname :
        varying[12] of char;
    curtablerow : integer;
exec sql end declare section;
begin
    {Embedded SQL and Pascal statements}
exec sql end.
```

The Procedure

The syntax for an Embedded SQL/Pascal procedure is:

> **procedure** *procedure_name*
> [**exec sql begin declare section;**
>   **(***formal_parameters)*
> **exec sql end declare section;**]
>   **;**
> [**exec sql begin declare section;**
>   *declarations*
> **exec sql end declare section;**]
> **begin**
>   [*statements*]
> **end;**

or:

> **procedure** *procedure_name*
> [**exec sql begin declare section;**
>   **(***formal_parameters)*
> **exec sql end declare section;**]
>   **;**
> **exec sql label;**
> [**exec sql begin declare section;**
>   *declarations*
> **exec sql end declare section;**]
> **begin**
>   [*statements*]
> **exec sql end;**

Syntax Notes:

1. The *procedure_name* is not processed by Embedded SQL.
2. Formal parameters and variables declared in a procedure are visible globally to the end of the source file.
3. For a description of formal parameters and their syntax, see Formal Parameter Declarations in this chapter.

The following is an example of an Embedded SQL/Pascal procedure:

```
procedure AppendRow
exec sql begin declare section;
    (   name : varying[20] of Char;
        age : Integer;
        salary : Real )
exec sql end declare section;
    ;
begin
    exec sql insert into emp (name, age, salary)
        values (:name, :age, :salary);
end;
```

The Function

The syntax for an Embedded SQL/Pascal function is:

function *function_name*
[**exec sql begin declare section;**
    (*formal_parameters)*
exec sql end declare section;]
    : return_type_name;
[**exec sql begin declare section;**
    declarations
exec sql end declare section;]
begin
    [*statements*]
end;

or:

function *function_name*
[**exec sql begin declare section;**
    (*formal_parameters)*
exec sql end declare section;]
    : return_type_name;
exec sql label;
[**exec sql begin declare section;**
    declarations
exec sql end declare section;]
begin
    [*statements*]
exec sql end;

Syntax Notes:

1. The *function_name* is not processed by Embedded SQL.
2. Formal parameters and variables declared in a function are globally visible to the end of the source file.
3. For a description of formal parameters and their syntax, see Formal Parameter Declarations in this chapter.

The following is an example of an Embedded SQL/Pascal function:

```
exec sql begin declare section;
var
    errorbuf : varying[100] of char;
exec sql end declare section;
    ...

function wasdeadlock : boolean;
exec sql begin declare section;
const
    EsqlDeadlock = -4700;
var
    errnum : Integer;
```

```
exec sql end declare section;
begin
    errnum := sqlca.sqlcode;
    if errnum = EsqlDeadLock then
    begin
            SetErrorBuf( errnum, errorbuf );
            WasDeadlock := TRUE;
    end else
    begin
            errorbuf := ' ';
            WasDeadlock := FALSE;
    end;
end;
```

Assembling and Declaring External Compiled Forms

You can pre-compile your forms in the Visual Forms Editor (VIFRED). Doing this saves the time otherwise required at runtime to extract the form's definition from the database forms catalogs. When you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED prompts you for the name of the file with the MACRO description. After the file is created, you can use the following VMS command to assemble it into a linkable object module:

macro *filename*

This command produces an object file containing a global symbol with the same name as your form. Before the Embedded SQL/FORMS statement **addform** can refer to this global object, you must declare it in an Embedded SQL declaration section. The Pascal compiler requires that this be an *external* declaration. The syntax for a compiled form declaration is:

exec sql begin declare section;
var
*formname***:** **[external] Integer;**
exec sql end declare section;

Syntax Notes:

1. The *formname* is the actual name of the form. VIFRED gives this name to the address of the external object. The *formname* is also used as the title of the form in other Embedded SQL/FORMS statements.
2. The **external** attribute associates the object with the external form definition.

The example below shows a typical form declaration and illustrates the difference between using the form's object definition and the form's name.

```
exec sql begin declare section;
var
    empform: [external] integer;
exec sql end declare section;
    ...
exec frs addform :empform; {The global object}
```

```
exec frs display empform;  {The name of the form}
    ...
```

Concluding Example

The following example demonstrates some simple Embedded SQL/Pascal declarations:

```
program Concluding_Example( input, output );
exec sql include sqlca;                   {Include error handling}
exec sql begin declare section;
const
         max_persons = 1000;
type
         shortshortinteger = [byte] -128 .. 127;
         shortinteger      = [word] -32768 .. 32767; {same as indicator type}
         string9           = packed array[1..9] of char;
         string12          = packed array[1..12] of char;
         string20          = packed array[1..20] of char;
         string30          = packed array[1..30] of char;
         varstring         = varying[40] of char;

record datatypes_rec = {Structure of all types}
         d_byte :                shortshortinteger;
         d_word :                shortinteger;
         d_long :                integer;
         d_single :              real;
         d_double :              double;
         d_string :              string20;
    end;

record Persontype_rec = {variant record}
         age :                   shortshortinteger;
         flags :                 integer;
         case married :          boolean of
            true :               (spouse_name : string30);
            false :              (dog_name : string12);
    end;
var
    empform, deptform : [external] integer;
                {compiled forms}
    dbname : String9;
    formname, tablename, columnname : String12;
    salary : Real;

    d_rec : Datatypes_Rec;
    person : Persontype_Rec;
    person_store : array[1..MAX_PERSONS] of Persontype_Rec;
    person_null: array[1..10] of Indicator;

    exec sql include 'employee.dcl'; {From DCLGEN}
exec sql end declare section;

begin
        dbname := 'personnel';
        ...

end. {Concluding_Example}
```

The Scope of Objects

All constants, types, and variables declared in an Embedded SQL declaration section can be referenced, and are accepted by the preprocessor, from the point of declaration to the end of the file, regardless of the Pascal scope of the declaration. This holds true for local variables and formal parameters. Once an object has been declared to Embedded SQL, it should not be redeclared to Embedded SQL for use in a different Pascal scope; the preprocessor will use the type information supplied by the original declaration. The object must, however, be redeclared to Pascal in the second scope to avoid errors from the Pascal compiler.

In the following program fragment, the variable "dbname" is passed as a parameter to the second procedure. In the first procedure, "dbname" is a local variable. In the second procedure, it is a formal parameter passed as a string to be used with the **connect** statement. The declaration of "dbname" as a formal parameter to the second procedure should not occur in an Embedded SQL declaration section. In both procedures, the preprocessor uses the type information from the variable's declaration in the first procedure.

```
program Decl_Test( input, output );
exec sql include sqlca;
exec sql begin declare section;
type
     String15 = packed array[1..15] of Char;
exec sql end declare section;

     procedure Open_Db( dbname: String15 ); forward;

     procedure Access_Db;
    exec sql begin declare section;
    var
            dbname: String15;
    exec sql end declare section;
    begin
    {"Dbname" is local to this procedure.}
            exec frs prompt ('Database: ', :dbname);
            Open_Db( dbname );
            Process_Db;
    end;

    { procedure Open_Db(dbname: String15); }
    procedure Open_Db;
    begin
            exec sql whenever sqlerror stop;
            {"Dbname" is known from the local declaration
                in      "       Access_Db".}
            exec sql connect :dbname;
                        ...

    end;

    begin {Decl_Test}
                ...

                Access_Db;
                ...
    end. {Decl_Test}
```

Note that you can declare record components with the same name if they are in different record types. The following example declares two records, each of which has the components "firstname" and "lastname":

```
exec sql begin declare section;
type
         Child = record
                firstname: varying[20] of Char;
                lastname: varying[20] of Char;
                age: Integer;
            end;
            Mother = record
                firstname: varying[20] of Char;
                lastname: varying[20] of Char;
                num_child: 1..10;
                children: array[1..10] of Child;
            end;
exec sql end declare section;
```

Special care should be taken when using variables with a **declare cursor** statement. The scope of the variables used in such a statement must also be valid in the scope of the **open** statement for that same cursor. The preprocessor actually generates the code for the **declare** at the point that the **open** is issued, and, at that time evaluates any associated variables. For example, in the following program fragment, even though the variable "number" is valid to the preprocessor at the point of both the **declare cursor** and **open** statements, it is not a valid variable name for the Pascal compiler at the point that the **open** is issued.

```
program Bad_Cursors( input, output );
{This example contains an error}
        procedure Init_Csr1 (num: Integer);
        exec sql begin declare section;
        var
            number: Integer;
        exec sql end declare section;
        begin
            number := num;
            exec sql declare cursor1 CURSOR FOR
                         select ename, age
                         from employee
                         where eno = :number;

            {Initialize "number" to a particular value}
            ...

        end; {Init_Csr1}

        procedure Process_Csr1;
        exec sql begin declare section;
        var
             ename: varying[15] of Char;
             age: Integer;
        exec sql end declare section;
        begin
             {Illegal evaluation of "number"}
             exec sql open cursor1;

             exec sql fetch cursor1 INTO :ename, :age;
             ...

        end; {Process_Csr1}
begin
```

```
    ...
end. {Bad_Cursors}
```

Variable Usage

Pascal variables declared to Embedded SQL can substitute for most non key-word elements of Embedded SQL statements. Of course, the variable and its data type must make sense in the context of the element. To use a Pascal variable (or named constant) in an Embedded SQL statement, you must precede it with a colon. You must further verify that the statement using the variable is in the scope of the variable's declaration. As an example, the following **select** statement uses the variables "namevar" and "numvar" to receive data, and the variable "idnovar" as an expression in the **where** clause:

```
exec sql select name, num
     into :namevar, :numvar
     from employee
     where idno = :idnovar;
```

You should not use the Pascal type-cast operator (::) in Embedded SQL statements. The preprocessor ignores it and does not change the type of the variable.

Various rules and restrictions apply to the use of Pascal variables in Embedded SQL statements. The sections below describe the usage syntax of different categories of variables and provide examples of such use.

Simple Variables

A simple scalar-valued variable (integer, floating-point, or character string) is referred to by the syntax:

:*simplename*

Syntax Notes:

1. If you use the variable to send data to Ingres, it can be any scalar-valued variable, constant, or enumerated literal.
2. If you use the variable to receive data from Ingres, it can only be a scalar-valued variable.
3. Packed or varying arrays of characters (for example, character strings) are referenced as simple variables.

The following program fragment demonstrates a typical message-handling routine that uses two scalar-valued variables, "buffer" and "seconds":

```
exec sql begin declare section;
var
     buffer : packed array[1..80] of char;
     seconds : integer;
```

```
exec sql end declare section;
begin
     ...

     exec frs message :buffer;
     exec frs sleep :seconds;
end;
```

A special case of a scalar type is the enumerated type. As mentioned in the section describing declarations, Embedded SQL treats all enumerated literals and any variables declared with an enumerated type as integers. When used in an Embedded SQL statement, only the ordinal position of the value in relation to the original enumerated list is relevant. When assigning into an enumerated variable, Embedded SQL will pass the object by address and assume that the value being assigned into the variable will not raise a runtime error. For example, the following enumerated type declares the states of a table field row, and the variable of that type will always receive one of those values:

```
exec sql begin declare section;
type
         Table_field_states =
            (undefined, newrow, unchanged, changed, deleted);
    var
         tbstate: table_field_states;
         ename: varying[20] of char;
exec sql end declare section;
             ...

tbstate := undefined;
exec frs getrow empform employee
         (:ename = name, :tbstate = _state);

case tbstate of
        undefined:
            ...

        deleted:
            ...
end;
```

Another example retrieves the value TRUE (a predefined constant of type **boolean**) into a variable when a database qualification is successful:

```
exec sql begin declare section;
var
     found: boolean;
exec sql end declare section;
     ...
found := false;
exec sql select :true
     into :found
     from emp
     where age > 62;
if not found then
begin
    ...

end;
```

Note that a colon precedes the Pascal constant "TRUE." The colon is required before all Pascal named objects—constants and enumerated literals, as well as variables—used in Embedded SQL statements.

Array Variables

An array variable is referred to by the syntax:

*:arrayname***[***subscript*{*,subscript*}**]** {**[***subscript*{*,subscript*}**]**}

Syntax Notes:

1. The variable must be subscripted because only scalar-valued elements (integers, floating-point and character strings) are legal Embedded SQL values.
2. When the array is declared, the array bounds specification is not parsed by the Embedded SQL preprocessor. Consequently, illegal bounds values will be accepted. Also, when an array is referenced, the subscript is not parsed, allowing the use of illegal subscripts. The preprocessor only confirms the use of an array subscript for an array variable. You must make sure that the subscript is legal and that the correct number of indices are used.
3. An array of characters is not a string unless it is **packed** or **varying**.
4. A **packed** or **varying** array of characters is considered a simple variable, not an array variable, in its usage. It therefore cannot be subscripted in order to reference a single character. For example, assuming the following variable declaration and subsequent assignment:

   ```
   exec sql begin declare section;
   var
        abc : packed array[1..3] of char;
   exec sql end declare section;
        ...
        abc := 'abc';
   ```

   you could not reference

   ```
   :abc[1]
   ```

   to access the character "a." To perform such a task, you should declare the variable as a plain (not **packed** or **varying**) array, as, for example:

   ```
   exec sql begin declare section;
   var
        abc : array[1..3] of char;
   exec sql end declare section;
        ...
        abc := ('a', 'b', 'c');
   ```

5. Arrays of indicator variables used with structure assignments should not include subscripts when referenced.

Record Variables

You can use a record variable in two different ways. First, you can use the record as a simple variable, implying the use of all its components. This would be appropriate in the Embedded SQL **select**, **fetch** and **insert** statements. Second, you can use a component of a record to refer to a single element. Of course, this component must be a scalar value (integer, floating-point or character string).

Using a Record as a Collection of Variables

The syntax for referring to a complete record is the same as referring to a simple variable:

:recordname

Syntax Notes:

1. The *recordname* can refer to a main or nested record. It can be an element of an array of records. Any variable reference that denotes a record is acceptable. For example:

```
:emprec                      {A simple record}
:record_array[i]     {An element of an array of records}
:record.minor2.minor3        {A nested record at level 3}
```

2. In order to be used as a collection of variables, the final record in the reference must have no nested records or arrays. All the components of the record will be enumerated by the preprocessor and must have scalar values. The preprocessor generates code as though the program had listed each record component in the order in which it was declared.

3. You must not use a record with a variant part as a complete record. The preprocessor generates explicit references to each of its components, including the components of the variant. Because the preprocessor generates references to all variant components, the use of a record with a variant part results in either a "wrong number of values" preprocessor error or a runtime error.

The example below uses the employee.dcl file generated by DCLGEN to retrieve values into a record.

```
exec sql begin declare section;
     exec sql include 'employee.dcl';
            {see above for description}
exec sql end declare section;
     ...
exec sql select *
     into :emprec
     from employee
     where eno = 123;
```

The example above generates code as though the following statement had been issued instead:

```
exec sql select *
     into    :emprec.eno, :emprec.ename, :emprec.age,
             :emprec.job, :emprec.sal, :emprec.dept
     from employee
     where eno = 123;
```

The example below fetches the values associated with all the columns of a cursor into a record.

```
exec sql begin declare section;
     exec sql include 'employee.dcl';
                          {see above for description}
exec sql begin declare section;

exec sql declare empcsr cursor for
     select *
     from employee
     order by ename;
     ...

exec sql fetch empcsr into :emprec;
```

The example below inserts values by looping through a locally declared array of records whose elements have been initialized:

```
exec sql begin declare section;
exec sql declare person table
        (pname          char(30),
         page           integer1,
         paddr          varchar(50));

type
        person_rec = record
        name:            packed array[1..30] of char;
        age:            [byte] -128 .. 127;
        addr:           varying[50] of char;
    end;
var
        person: array[1..10] of person_rec;
exec sql end declare section;
        ...

for i := 1 to 10 do
begin
        exec sql insert into person
            values (:person[i]);
end;
```

The **insert** statement in the example above generates code as though the following statement had been issued instead:

```
exec sql insert into person
        values (:person[i].name, :person[i].age,
         :person[i].addr);
```

Using Record Components

The syntax Embedded SQL uses to refer to a record component is the same as in Pascal:

:*record_name*{**^** |**[*s***ubscript*]*}**.***component*{**^**|**[***subscript*]}
{**.***component*{**^** | **[***subscript*]}}

that is, the name of the record, followed by any number of pointer dereference operators or array subscripts, followed by one or more field names (with any number of pointer dereference operators or array subscripts attached).

Syntax Notes:

1. The last record *component* denoted by the above reference must be a scalar value (integer, floating-point or character string). There can be any combination of arrays and records, but the last object referenced must be a scalar value. Thus, the following references are all legal:

```
{Assume correct declarations for "employee", "person" and other records.}
:employee.sal                {Component of a record}
:person[3].name              {Component of an element of an array}
:rec1.mem1.mem2.age  {Deeply nested component}
```

2. Any array subscripts or pointer references referred to in the record reference, and not at the very end of the reference, are not checked by the preprocessor. Consequently, both of the following references are accepted, even though one must be wrong, depending on whether “person” is an array:

```
:person[1].age
:person.age
```

The following example uses the array of records “emprec” to load values into the table field “emptable” in form “empform.”

```
exec sql begin declare section;
type
     EmployeeRec = record
            ename: packed array[1..20] of Char;
            eage: [word] -32768 .. 32767;
            eidno: Integer;
            ehired: packed array[1..25] of Char;
            edept: packed array[1..10] of Char;
            esalary: Real;
    end;
var
    emprec: array[1..100] of EmployeeRec;
    i: Integer;
exec sql end declare section;
        ...

for i := 1 to 100 do
begin
     exec frs loadtable empform emptable
        (name = :emprec[i].ename, age = :emprec[i].eage,
         idno = :emprec[i].eidno, hired = :emprec[i].ehired,
         dept = :emprec[i].edept, salary = :emprec[i].esalary);
end;
```

Pointer Variables

A pointer variable references an object in the same way as in Pascal—the name of the pointer is followed by a caret (^):

:*pointer_name*^

Any further referencing required to fully qualify an object, such as a member of a pointed-to record, follows the usual Pascal syntax.

Syntax Notes:

1. The final object denoted by the pointer reference must be a scalar value (integer, floating-point or character string) or a record (if this is a legal simple record reference). There can be any combination of arrays, records or pointer variables, as long as the last object referenced has a scalar value or is a legal simple record.
2. The pointer reference is also used with **file** type variables (see the example under Formal Parameter Declarations in this chapter).

In the following example, a pointer to an employee record is used to load a linked list of values into the database table "employee":

```
exec sql begin declare section;
type
     EmpLink = ^EmployeeRec;
     EmployeeRec = record
            ename: packed array [1..20] of Char;
            eage: Integer;
            eidno: Integer;
            enext: EmpLink;
    end;
var
    elist: EmpLink;
exec sql end declare section;
    ...

while (elist <> nil) do
begin
    exec sql insert into employee (name, age, idno)
            values (:elist^.ename, :elist^.eage,
                 :elist^.eidno);
    elist := elist^.enext;
end;
```

Indicator Variables

The syntax for referring to an *indicator* variable is the same as for a simple variable, except that an indicator variable is always associated with a host variable:

:*host_variable:indicator_variable*

or

:host_variable ***indicator :indicator_variable***

Syntax Notes:

1. The indicator variable can be a simple variable, an array element or a record component that yields a 2-byte integer. The type **indicator** has already been declared by the preprocessor. For example:

```
ind_var, ind_arr[5] : Indicator;
:var_1:ind_var
:var_2:ind_arr[2]
```

2. If the host variable associated with the indicator variable is a record, the indicator variable should be an array of 2-byte integers. In this case the array should *not* be dereferenced with a subscript.

3. When using an indicator array, the first element of the array corresponds to the first member of the record, the second element with the second member, and so on. Indicator array elements begin at subscript 1, regardless of the lower bound with which the array was declared.

The following example uses the employee.dcl file generated by DCLGEN to retrieve values into a structure and null values into the array “empind”.

```
exec sql include sqlca;
exec sql begin declare section

     exec sql include 'employee.dcl';
var
     empind : array[1..10] of Indicator;

exec sql end declare section;

exec sql select *
     into :emprec:empind
     from employee;
```

The above example generates code as though the following statement had been issued:

```
exec sql select *
     into :emprec.eno:empind[1], :emprec.ename:empind[2],
            :emprec.age:empind[3], :emprec.job:empind[4],
            :emprec.sal:empind[5], :emprec.dept:empind[6],
     from employee;
```

Data Type Conversion

A Pascal variable declaration must be compatible with the Ingres value it represents. Numeric Ingres values can be set by and retrieved into numeric variables, and Ingres character values can be set by and retrieved into character string variables.

Data type conversion occurs automatically for different numeric types, as follows:

- From floating-point Ingres database column values into integer Pascal variables

- From decimal to floating-point
- From floating-point to decimal
- For different length character strings, such as from varying-length Ingres character fields into fixed-length Pascal character string variables

Ingres does *not* automatically convert between numeric and character types. You must use the Ingres type conversion functions, the Ingres **ascii** function, or a Pascal conversion procedure for this purpose.

The following table shows the default type compatibility for each Ingres data type. Note that some Pascal types do not match exactly and, consequently, can go through some runtime conversion.

Ingres and Pascal Data Type Correspondence

Ingres Type	Pascal Type
char(*N)*	packed array[*1..N*] *of char*
char(*N)*	varying[*N] of char*
varchar(*N)*	packed array[*1..N*] *of char*
varchar(*N)*	varying[*N*] *of char*
integer1	[byte] -128..127
smallint	[word] -32768..32767
integer	integer
float4	real
float4	single
float	double
date	packed array[*1..25*] of char
money	double
table_key	packed array[*1..8*] of char
object_key	packed array[*1..16*] of char
decimal	real
long varchar	packed array

Runtime Numeric Type Conversion

The Ingres runtime system provides automatic data type conversion between numeric-type values in the database and forms system and numeric Pascal variables. The standard type conversion rules (according to standard VAX rules) are followed. For example, if you assign a **real** variable to an integer-valued field, the digits after the decimal point of the variable's value are truncated. Runtime errors are generated for overflow on conversion when assigning Ingres numeric values into Pascal variables. Overflow caused by assigning Pascal numeric variables into Ingres numeric objects is likely to result in inconsistent data, but does not by default generate a runtime error. Using the **-x** flag on the Ingres statement changes this default behavior by generating errors at runtime.

The Ingres **money** type is represented as **double**, an 8-byte floating-point value.

Runtime Character and Varchar Type Conversion

Automatic conversion occurs between Ingres character string values and Pascal character string variables. There are string-valued Ingres objects that can interact with character string variables. These are:

Ingres names, such as form and column names

- database columns of type **character**
- database columns of type **varchar**
- form fields of type **character**
- database columns of type **long varchar**

Several considerations apply when dealing with character string conversions, both to and from Ingres.

The conversion of Pascal character string variables used to represent Ingres names is simple: trailing blanks are truncated from the variables, because the blanks make no sense in that context. For example, the string literals "empform " and "empform" refer to the same form.

The conversion of other Ingres objects is a bit more complicated. First, the storage of character data in Ingres differs according to whether the medium of storage is a database column of type **character**, a database column of type **varchar** or a **character** form field. Ingres pads columns of type **character** with blanks to their declared length. Conversely, it does not add blanks to the data in columns of type **varchar** or **long varchar** in form fields.

Second, the storage of character data in Pascal differs according to whether the character variable is of fixed or of varying length. The Pascal convention is to blank-pad fixed-length character strings, but not to pad varying-length character strings. For example, the character string "abc" coming from an Ingres object will be stored in a Pascal **packed array[1..5] of char** variable as the string "abc " followed by two blanks. However, the same string would be stored in a **varying[5] of char** variable as "abc" without any trailing blanks.

When retrieving character data from an Ingres database column or form field into a Pascal variable, you should always ensure that the variable is at least as long as the column or field, in order to avoid truncation of data. Furthermore, take note of the following conventions:

- Data stored in a database column of type **character** is padded with blanks to the length of the column. The variable receiving such data, be it of fixed or varying length, will contain those blanks. Following Pascal rules, if a fixed-length variable is longer than the database column, the data retrieved into it is further padded with blanks to the length of the variable. In the case of a varying-length variable, no further padding takes place. If the variable is shorter than the database column, truncation of data occurs.
- Data stored in a database column of type **varchar** is not padded with blanks. If a fixed-length variable is longer than the data in the **varchar** column, when retrieved the data is padded with blanks to the length of the variable. In the case of a varying-length variable, no padding takes place. If the variable is shorter than the database column, truncation of data occurs.
- Data stored in a **character** form field contains no trailing blanks. If a fixed-length variable is longer than the data in the field, when retrieved the data is padded with blanks to the length of the variable. In the case of a varying-length variable, no padding takes place. If the variable is shorter than the field, truncation of data occurs.

When inserting character data into an Ingres database column or form field from a Pascal variable, note the following conventions:

- When data is inserted from a Pascal variable into a database column of type **character** and the column is longer than the variable, the column is padded with blanks. If the column is shorter than the variable, the data is truncated to the length of the column.

- When data is inserted from a Pascal variable into a database column of type **varchar** or **long varchar** and the column is longer than the variable, no padding of the column takes place. Furthermore, by default, all trailing blanks in the data are truncated before the data is inserted into the **varchar** column. For example, when a string "abc" stored in a Pascal **packed array[1..5] of char** variable as "abc " (see above) is inserted into the **varchar** column, the two trailing blanks are removed and only the string "abc" is stored in the database column. To retain such trailing blanks, you can use the Embedded SQL **notrim** function.

  It has the following syntax:

  notrim(:*stringvar*)

  where *stringvar* is a character string variable. An example demonstrating this feature follows later. If the **varchar** column is shorter than the variable, the data is truncated to the length of the column.

- When data is inserted from a Pascal variable into a **character** form field and the field is longer than the variable, no padding of the field takes place. In addition, all trailing blanks in the data are truncated before the data is inserted into the field. If the field is shorter than the data (even after all trailing blanks have been truncated), the data is truncated to the length of the field.

When comparing character data in an Ingres database column with character data in a Pascal variable, note the following convention:

- When comparing data in **character** or **varchar** database columns with data in a character variable, all trailing blanks are ignored. Initial and embedded blanks are significant.

Note: As described above, the conversion of character string data between Ingres objects and Pascal variables often involves the trimming or padding of trailing blanks, with resultant change to the data. If trailing blanks have significance in your application, give careful consideration to the effect of any data conversion. For a complete description of the significance of blanks in string comparisons, see the *SQL Reference Guide*.

The Ingres **date** data type is represented as a 25-byte character string.

The program fragment in the example below demonstrates the **notrim** function and the truncation rules explained above.

```
exec sql include sqlca;
    ...
exec sql begin declare section;

exec sql declare textchar table
    (row integer,
     data varchar(10));          {Note the varchar data type}

var
    row:    Integer;
    p_data: packed array[1..7] of Char;
    v_data: varying[7] of Char;
```

```
    ...
exec sql end declare section;

begin
    p_data := 'abc    ';                   {Holds "abc "}
    v_data := 'abc';                       {Holds "abc"}

    {The following insert adds the string "abc" (blanks truncated)}
    exec sql insert into textchar (row, data)
         values (1, :p_data);

    {The following insert adds the string "abc" (never had blanks)}
    exec sql insert into textchar (row, data)
         values (2, :v_data);

    {
    | This statement adds the string "abc ", with 4 trailing
    | blanks left intact by using the NOTRIM function.
    }

    exec sql insert into textchar (row, data)
         values (3, notrim(:p_data));

    {
    | The following FETCH retrieves rows #1 and #2, because trailing
    | blanks were suppressed when those rows were inserted.
    }

    exec sql declare csr cursor for
         select row
         from textchar
         wherE length(data) = 3;

    exec sql open csr;

    while (sqlca.sqlcode = 0) do
    begin
        exec sql fetch csr into :row;
        if (sqlca.sqlcode = 0) then
            writeln( 'Row found = ', row );
    end;

    exec sql close csr;

    {
    | The following FETCH retrieves row #3, because the NOTRIM
    | function left trailing blanks in the "p_data" variable
    | in the last INSERT statement.
    }

    exec sql declare csr2 cursor for
         select row
         from textchar
         where length(data) = 7;

            exec sql open csr2;

    while (sqlca.sqlcode = 0) do
    begin
        exec sql fetch csr2 into :row;
        if (sqlca.sqlcode = 0) then
            writeln( 'Row found = ', row );
    end;
```

```
    exec sql close csr2;
end;
```

The SQL Communications Area

This section describes the SQL Communications Area (SQLCA) as implemented in Pascal.

The Include SQLCA Statement

You should issue the **include sqlca** statement in the outermost scope of your Pascal program:

```
program Emp_Update( input, output )

exec sql include sqlca;

{Declarations, procedures, etc.}
begin
            {Host language and embedded statements}

end.
```

The **include sqlca** statement generates a Pascal **include** directive to make certain calls generated by the preprocessor acceptable to the compiler. The **include sqlca** statement also generates a Pascal **include** directive to define the SQLCA (SQL Communications Area) record, used for error handling and defining the **indicator** type used for null indicators.

Whether or not you intend to use the SQLCA for error handling, you must issue an **include sqlca** statement. If you do not issue it, the Pascal compiler will generate errors about undeclared built-in function and procedure names. Note that some error handling mechanism should be included before all executable Embedded SQL database statements, as the default action is to ignore errors, which is rarely desirable.

Contents of the SQLCA

One of the results of issuing the **include sqlca** statement is the declaration of the SQLCA record, which can be used for error handling in the context of database statements. You should only issue the statement once in a particular Pascal scope, because it generates an external record variable definition. The nested record declaration for the SQLCA is:

```
type
    IISQLCA = record
        sqlcaid: packed array[1..8] of Char;
        sqlcabc: Integer;
        sqlcode: Integer;
        sqlerrm: varying[70] of Char;
```

```
        sqlerrp: packed array[1..8] of Char;
        sqlerrd: array[1..6] of Integer;
        sqlwarn: record
            sqlwarn0: Char;
            sqlwarn1: Char;
            sqlwarn2: Char;
            sqlwarn3: Char;
            sqlwarn4: Char;
            sqlwarn5: Char;
            sqlwarn6: Char;
            sqlwarn7: Char;
        end;
        sqlext: packed array[1..8] of Char;
    end;
var
    sqlca: [common] IISQLCA;
```

The record member **sqlerrm** is a varying length character string which Pascal stores as if it were declared as:

```
sqlerrm: record
    length  : [word] 0..70;
    body    : packed array[1..70] of Char;
end;
```

Here "length" corresponds to the standard SQLCA variable **sqlerrml** and "body" corresponds to the standard SQLCA variable **sqlerrmc**. For a full description of all the SQLCA record members, see the *SQL Reference Guide*.

The SQLCA is initialized at load-time. The fields **sqlcaid** and **sqlcabc** are initialized to the string "SQLCA " and the constant 136, respectively.

Note that the preprocessor is not aware of the record declaration. Therefore, you cannot use members of the record in an Embedded SQL statement. For example, the following statement, attempting to **insert** the string "SQLCA" into a table, would generate an error:

```
exec sql insert into employee (ename)            {This statement is illegal}
     values (:sqlca.sqlcaid);
```

All modules written in Pascal and other embedded languages share the same SQLCA.

Using the SQLCA for Error Handling

Error handling with the SQLCA can be done implicitly by using **whenever** statements, or explicitly by checking the contents of the SQLCA fields **sqlcode**, **sqlerrd**, and **sqlwarn0**.

Error Handling with the Whenever Statement

The syntax of the **whenever** statement is as follows:

exec sql whenever *condition action;*

condition is **dbevent**, **sqlwarning**, **sqlerror**, **sqlmessage**, or **not found**. *action* is **continue**, **stop**, **goto** a label or **call** a Pascal procedure. For a detailed description of this statement, see the *SQL Reference Guide*.

In Embedded SQL/Pascal, all labels and procedure names must be legal Pascal label identifiers, beginning with a digit, an alphabetic character, or an underscore. If the label is an Embedded SQL reserved word, it should be specified in quotes. Note that the label targeted by the **goto** action must be in the scope of all subsequent Embedded SQL statements until another **whenever** statement is encountered for the same action. This is necessary because the preprocessor can generate the Pascal statement:

if (*condition*) *then goto label;*

after an Embedded SQL statement. If the scope of the label is invalid, the Pascal compiler will generate an error.

The same scope rules apply to procedure names used with the **call** action. Note that the reserved procedure **sqlprint**, which prints errors or database procedure messages and then continues, is always in the scope of the program. When a **whenever** statement specifies a **call** as the action, the target procedure is called, and after its execution, control returns to the statement following the statement that caused the procedure to be called. Consequently, after handling the **whenever** condition in the called procedure, you may want to take some action, instead of merely returning from the Pascal procedure. Returning from the Pascal procedure will cause the program to continue execution with the statement following the Embedded SQL statement that generated the error.

The following example demonstrates use of the **whenever** statements in the context of printing some values from the Employee table. The comments do not relate to the program but to the use of error handling.

```
program Db_Test( input, output );
label
        Close_Csr,
        Exit_Label;

exec sql begin declare section;
var
     eno:        [word] -32768 .. 32767;
     ename:      varying[20] of Char;
     age:        [byte] -128 .. 127;
exec sql end declare section;
     exec sql include sqlca;

     exec sql declare empcsr cursor for
        select eno, ename, age
        from employee;

    {
    | Clean_Up: Error handling procedure (print error and disconnect).
    }

    procedure Clean_Up;
    exec sql begin declare section;
```

```
    var
             errmsg: varying[200] of Char;
    exec sql end declare section;
    begin       {Clean_Up}
         exec sql whenever sqlerror stop;
         inquire_sql (:errmsg = errortext) ;
         writeln( 'Aborting because of error: ' );
         writeln( errmsg );
         exec sql disconnect;
         goto Exit_Label;
    end; {Clean_Up}

begin           {Db_Test}
    {
    | An error when opening the personnel database
    | will cause the error to be printed and the
    | program to abort.
    }

    exec sql whenever sqlerror stop;
    exec sql connect personnel;

    {
    |   Errors from here on will cause the program to clean up.
    }
    exec sql whenever sqlerror call Clean_Up;

    exec sql open empcsr;

    writeln( 'Some values from the "employee" table.' );

    {When no more rows are fetched, close the cursor.}
    exec sql whenever not found goto Close_Csr;

    {
    | The last executable Embedded SQL statement
    |   was an OPEN,so we know that the value of
    |   "sqlcode" cannot be SQLERROR or NOT FOUND.
    }

    while (sqlca.sqlcode = 0) do
    {Loop is broken by NOT FOUND}
    begin
        exec sql fetch empcsr
             into :eno, :ename, :age;

        {
        | This writeln statement does not execute
        |   after the previous FETCH returns the
        |   NOT FOUND condition.
        }
        writeln( eno, ', ', ename, ', ', age );
    end; {while}
    {
    | From this point in the file onwards, ignore
    |   all errors. Also turn off the NOT FOUND
    | condition, for consistency.
    }

    exec sql whenever sqlerror continue;
    exec sql whenever not found continue;
Close_Csr:
    exec sql close empcsr;
    exec sql disconnect;
```

```
Exit_Label::
end; {Db_Test}
```

The Whenever Goto Action in Embedded SQL Blocks

An Embedded SQL block-structured statement is a statement delimited by the **begin** and **end** clauses. For example, the **select** loop and the **unloadtable** loops are both block-structured statements. These statements can be terminated only by the methods specified for the particular statement in the *SQL Reference Guide*. For example, the **select** loop is terminated either when all the rows in the database result table have been processed or by an **endselect** statement, and the **unloadtable** loop is terminated either when all the rows in the forms table field have been processed or by an **endloop** statement.

Therefore, if you use a **whenever** statement with the **goto** action in an SQL block, you must avoid going to a label outside the block. Such a **goto** would cause the block to be terminated without issuing the runtime calls necessary to clean up the information that controls the loop. (For the same reason, you must not issue a Pascal **goto** statement that causes control to leave or enter the middle of an SQL block.) The target label of the **whenever goto** statement should be a label in the block. If, however, it is a label for a block of code that cleanly exits the program, the above precaution need not be taken.

The above information does not apply to error handling for database statements issued outside an SQL block, nor to explicit hard-coded error handling. For an example of hard-coded error handling, see The Table Editor Table Field Application in this chapter.

Explicit Error Handling

The program can also handle errors by inspecting values in the SQLCA record at various points. For further details, see the *SQL Reference Guide*.

The following example is functionally the same as the previous example, except that the error handling is hard-coded in Pascal statements.

```
program Db_Test( input, output );
label
    Exit_Label;
exec sql begin declare section;
const
    not_found = 100;
var
    eno:    [word] -32768 .. 32767;
    ename:  varying[20] of Char;
    age:    [byte] -128 .. 127;
exec sql end declare section;
    exec sql include sqlca;

    exec sql declare empcsr cursor for
         select eno, ename, age
         from employee;
```

```
    {
    | Clean_Up: Error handling procedure (print error and disconnect).
    }

    procedure Clean_Up( str : varying[ub] of Char );
    exec sql begin declare section;
    var
         errmsg: varying[200] of Char;
        err_stmt: varying[40] of Char;
    exec sql end declare section;
    begin {Clean_Up}
        err_stmt := str;
        exec sql inquire_sql (:errmsg = ERRORTEXT);
        writeln('Aborting because of error in ', err_stmt, ': ');
        writeln( errmsg );
        exec sql disconnect;

        goto Exit_Label;
    end; {Clean_Up}

begin                   {Db_Test}
        {Exit if the database cannot be opened.}
        exec sql connect personnel;
        if (sqlca.sqlcode < 0) then
        begin
            writeln( 'Cannot access database.' );
            goto Exit_Label;
        end;

        {Errors if cannot open cursor.}
        exec sql open empcsr;
        if (sqlca.sqlcode < 0) then
                Clean_Up( 'OPEN "empcsr"' ); {No return}

        writeln( 'Some values from the "employee" table.' );

        {
        | The last executable Embedded SQL statement was an OPEN,
        | so we know that the value of "sqlcode" cannot be SQLERROR
        | or NOT FOUND.
        }

            while (sqlca.sqlcode = 0) do {
                                         | Loop is broken by NOT FOUND
                                         }
            begin
                exec sql fetch empcsr
                        into :eno, :ename, :age;

            {Do not print the last values twice.}
            if (sqlca.sqlcode < 0) then
                        Clean_Up( 'FETCH "empcsr"' )
            else if (sqlca.sqlcode <> NOT_FOUND) then
                writeln( eno, ', ', ename, ', ', age );
        end; {while}

{
| From this point in the file onwards, ignore all errors.
}
    exec sql close empcsr;
    exec sql disconnect;

Exit_Label:;
end; {Db_Test}
```

Determining the Number of Affected Rows

The third element of the SQLCA array **sqlerrd** indicates how many rows were affected by the last row-affecting statement. The following program fragment, which deletes all employees whose employee numbers are greater than a given number, demonstrates how to use **sqlerrd**:

```
procedure Delete_Rows( lower_bound: Integer );
exec sql begin declare section;
var
     lower_bound_num: Integer;
exec sql end declare section;
begin
    lower_bound_num := lower_bound;
    exec sql delete from employee
         where eno > :lower_bound_num;

    {Print the number of employees deleted.}
    writeln( sqlca.sqlerrd[3], ' (rows) were deleted.' );
end; {Delete_Rows}
```

Using the SQLSTATE Variable

You can use the **SQLSTATE** variable in an ESQL/Pascal program to return status information about the last SQL statement that was executed. **SQLSTATE** must be declared in a declaration section. Also, it is valid across all sessions, so you only need to declare one **SQLSTATE** per application.

To declare this variable, use:

```
character 5 SQLSTATE
```

or :

```
character 5 SQLSTA
```

Dynamic Programming for Pascal

Ingres provides Dynamic SQL and Dynamic FRS to allow you to write generic programs. Dynamic SQL allows a program to build and execute SQL statements at runtime. For example, an application can include an expert mode in which the runtime user can type in select queries and browse the results at the terminal. Dynamic FRS allows a program to interact with any form at runtime. For example, an application can load in any form, allowing the runtime user to retrieve new data from the form and insert it into the database.

The Dynamic SQL and Dynamic FRS statements are described in the *SQL Reference Guide* and the *Forms-based Application Development Tools User Guide,* respectively. This section discusses the Pascal-dependent issues of dynamic programming. For a complete example of using Dynamic SQL to write an SQL Terminal Monitor application, see The SQL Terminal Monitor Application in this chapter. For an example of using both Dynamic SQL and Dynamic FRS to browse and update a database using any form, see A Dynamic SQL/Forms Database Browser in this chapter.

This section is written exclusively for VAX/VMS Pascal and makes use of the VMS extensions to the Pascal language, in particular the ability to point at any object using the built-in **address** functions.

The SQLDA Record

The SQLDA (SQL Descriptor Area) is used to pass type and size information about an SQL statement, an Ingres form, or Ingres table field, between Ingres and your program.

In order to use the SQLDA, you should issue the **include sqlda** statement at the proper scope of the source file, from where the SQLDA will be referenced. The **include sqlda** statement generates a Pascal **include** directive to a file that defines the SQLDA record type. The file does *not* declare an SQLDA variable; your program must declare a variable of the specified type. You can also code this record variable directly instead of using the **include sqlda** statement. You can choose any name for the record. The definition of the SQLDA (as specified in the **include** file) is:

```
const
                                          { Sizes }
    IISQ_MAX_COLS = 1024;                 { Maximum number of columns }
    IISQ_DTE_LEN = 25;                    { Date length }
                                          { Data type codes }
    IISQ_DTE_TYPE = 3;                    { Date - Output }
    IISQ_MNY_TYPE = 5;                    { Money - Output }
    IISQ_DEC_TYPE = 10;                   { Decimal - Output)
    IISQ_CHA_TYPE = 20;                   { Char - Input, Output }
    IISQ_VCH_TYPE = 21;                   { Varchar - Input, Output }
    IISQ_INT_TYPE = 30;                   { Integer - Input, Output }
    IISQ_FLT_TYPE = 31;                   { Float - Input, Output }
    IISQ_TBL_TYPE = 52;                   { Table field - Output }

type
    II_int2 = [word] -32768..32767; { 2-byte integer }

    IIsqlvar = record                     { Single SQLVAR element }
        sqltype:        II_int2;
        sqllen:         II_int2;
        sqldata:        Integer;          { Address of any type }
        sqlind:         Integer;          { Address of 2-byte integer }
        sqlname:        Varying[34] of Char;
 end;

IIsqlda = record                          { Full SQLDA definition }
        sqldaid: packed array[1..8] of Char;
        sqldabc: Integer;
```

```
        sqln:    II_int2;
        sqld:    II_int2;
        sqlvar:  array[1..IISQ_MAX_COLS] of IIsqlvar;
end;
```

Record Definition and Usage Notes:

- The record type definition of the SQLDA is called IISQLDA. This is done so that an SQLDA variable can be called "SQLDA" without causing a Pascal compile-time conflict.
- The **sqlvar** array is an array of IISQ_MAX_COLS (1024) elements. If an SQLDA record variable of type IISQLDA is declared, then the program will have a record with IISQ_MAX_COLS elements.
- Note that the **sqlvar** array begins at subscript 1.
- The **sqldata** and **sqlind** record components are declared as 4-byte integers. These integers actually contain addresses and must be set to point at other global or dynamically allocated variables using the **address** or **iaddress** built-in Pascal functions.
- If your program defines its own SQLDA type, you must verify that the internal record layout is identical to that of the IISQLDA record type, although you can declare a different number of **sqlvar** elements.
- The **sqlname** component is a varying length character string consisting of a length and data area. This varying length name contains the name of a result field or column after a **describe** (or **prepare into**) statement. The length of the name is implicit with varying length data type. The varying length name can also be set by the program using Dynamic FRS.
- The list of type codes represents the types that will be returned by the **describe** statement, and the types used by the program when retrieving or setting data using an SQLDA. The type code IISQ_TBL_TYPE indicates a table field and is set by the FRS when describing a form that contains a table field.

Declaring an SQLDA Record Variable

Once the SQLDA type definition has been included (or hard-coded), the program can declare an SQLDA record variable. This variable must be declared outside of a **declare section**, as the preprocessor does not understand the special meaning of the components of the SQLDA. When the variable is used, the preprocessor will accept any object name, and assume that the variable refers to a legally declared SQLDA record.

If a program requires a statically declared SQLDA with the same number of **sqlvar** variables as the IISQLDA type, then it can accomplish this as in the following example:

```
exec sql include sqlda;
var
    sqlda: IIsqlda;             { Outside of a DECLARE SECTION }
```

```
...
sqlda.sqln := IISQ_MAX_COLS; { Set the size }
...
exec sql describe s1 into :sqlda;
```

Recall that you must confirm that the SQLDA object being used is a valid SQLDA record variable.

If a program requires a statically declared SQLDA with a *different* number of variables (not IISQ_MAX_COLS), it can declare its own type. For example:

```
const
    NUM_COLS = 20;

type
    My_Sqlda = record
        my_sqid:        packed array[1..8] of Char;
        my_sqbc:        Integer;
        my_vars:        [word] 0..500;
        res_vars:       [word] 0..500;
        col_vars:       array[1..NUM_COLS] of IIsqlvar;
    end;

var
    my_sq: My_Sqlda;

...

my_sq.my_vars := NUM_COLS; { Set the size }

...

exec sql describe s1 into :my_sq;
```

In the above declaration the names of the record components are not the same as those of the IISQLDA record, but their layout is identical.

If the variable in the above example was declared as a pointer to an SQLDA record type, then it can be dynamically allocated and used as in the following example:

```
{ Assume My_Sqlda is declared as above }

var
    ptr_sq: ^My_Sqlda;

...

new(ptr_sq);

ptr_sq^.my_vars := NUM_COLS; { Set the size }

...

exec sql describe s1 into :ptr_sq^;
```

Using the SQLVAR

The *SQL Reference Guide* discusses the legal values of the **sqlvar** array. The **describe** and **prepare into** statement assigns type, length, and name information to the SQLDA. This information refers to the result columns of a prepared **select** statement, the fields of a form, or the columns of a table field. When the program uses the SQLDA to retrieve or set Ingres data, it must assign the type and length information that now refers to the variables being pointed at by the SQLDA.

Pascal Variable Type Codes

The type codes shown in The SQLDA Record in this chapter are the types that describe Ingres result fields or columns. For example, the SQL types **date** and **money** do not describe a program variable, but rather data types that are compatible with the Pascal character and numeric types. IISQ_LVCH_TYPE is SQL only character compatible too. When these types are returned by the **describe** statement, the type code must be a change to a compatible Pascal or ESQL/Pascal type.

The following table describes the type codes to use with Pascal variables that will be pointed at by the **sqldata** pointers.

The SQLDA Type Codes

Pascal Type	SQL Type Codes (sqltype)	SQL Length (sqllen)
[byte] -128..127	30 (integer)	1
[word] -32768..32767	30 (integer)	2
Integer	30 (integer)	4
Real	31 (float)	4
Double	31 (float)	8
Packed array[1..LEN] of Char	20 (char)	LEN
Varying[LEN] of Char	21 (varchar)	LEN
Real	31 (float)	10

Nullable data types (those variables that are associated with a null indicator) are specified by assigning the negative of the type code to the **sqltype** component. If the type is negative, a null indicator must be pointed at by the **sqlind** component. The type of the null indicator must be a 2-byte integer (or the SQL-defined **indicator** type). For information on how to declare and use a null indicator variable in Pascal, see Pascal Variables and Data Types in this chapter.

Character data and the SQLDA have the exact same rules as character data in regular Embedded SQL statements. For details of character string processing in SQL, see Pascal Variables and Data Types in this chapter.

Pointing at Pascal Variables

In order to fill an element of the **sqlvar** array, you must set the type information and assign a valid address to **sqldata**. The address must be that of a legal variable address. If the element is nullable, the corresponding **sqlind** component must point at a legally declared null indicator.

Because both the **sqldata** and **sqlind** components of the IISQLDA record are declared as integers, you must assign integer values to them. This requires the use of the built-in **iaddress** function (as shown in Appendices E and F), or other pointer and address operations. The Pascal compiler requires you to declare the target variables with the **volatile** attribute in order to use the **iaddress** and **address** functions.

For example, the following fragment sets the type information of and points at a 4-byte integer variable, an 8-byte nullable floating-point variable, and an **sqllen**-specified character substring. This example demonstrates how a program can maintain a pool of available variables, such as large arrays of the few different typed variables, and a large string space. The next available spot is chosen from the pool, as in the following example:

```
{
| Assume sqlda has been declared, as well as
| the following VOLATILE numeric arrays and
| large array of characters: int4_store,
| float8_store, indicator_store, char_store
}

sqlda.sqlvar[1].sqltype  := IISQ_INT_TYPE;       { 4-byte integer }
sqlda.sqlvar[1].sqllen   := 4;
sqlda.sqlvar[1].sqldata  := iaddress(int4_store[current_int]);
sqlda.sqlvar[1].sqlind   := 0;
current_int := current_int + 1; { Update integer pool }

sqlda.sqlvar[2].sqltype  := -IISQ_FLT_TYPE;      { 8-byte nullable float }
sqlda.sqlvar[2].sqllen   := 8;
sqlda.sqlvar[2].sqldata  :=iaddress
                                (float8_store[current_float]);
sqlda.sqlvar[2].sqlind
                         := iaddress(indicator_store[current_ind]);
current_float            := current_float + 1; { Update float and }
current_ind := current_ind + 1; { indicator pool }
```

```
{
| SQLLEN has been assigned by DESCRIBE to be the length of a specific result
| column. This length is used to pick off a substring from a large string space.
}
needlen                  := sqlda.sqlvar[3].sqllen;
sqlda.sqlvar[3].sqltype  := IISQ_CHA_TYPE;
sqlda.sqlvar[3].sqldata
                         := iaddress(char_store[current_char]);
sqlda.sqlvar[3].sqlind   := 0;
current_char := current_char + needlen;          { Update char pool }
```

Of course, in the above example, verification of enough pool storage must be made before referencing each cell of the different arrays in order to prevent **sqldata** and **sqlind** from pointing at undefined storage. Appendices E and F demonstrate this method.

The IISQ_HDLR_TYPE is a host language type that is used for transmitting data to and from Ingres. Because it is not an Ingres data type, it will never be returned as a data type from the **describe** statement.

If you code your own SQLDA, and, in place of **sqldata**, you declare a variant record of pointers to a subset of different data types, you may find that you can use dynamic allocation routines and simple pointer assignments. For example, you can declare a type:

```
type
        Data_Pointer = record
                case Integer of
                IISQ_INT_TYPE: (int_ptr: ^Integer);
                IISQ_FLT_TYPE: (flt_ptr: ^Double);
                IISQ_CHA_TYPE: (str_ptr: ^Char);
         end;
```

and use this type instead of the **sqldata** component. If you confirm that the layout of the variant record of different pointers is the same as that of a 4-byte integer (**sqldata**), then you may use this method. This approach is not discussed further in this manual.

Setting SQLNAME for Dynamic FRS

When using the **sqlvar** with Dynamic FRS statements there are a few extra steps that are required. These extra steps relate to the differences between Dynamic FRS and Dynamic SQL and are described in the *SQL Reference Guide*.

When using the SQLDA in a forms input or output **using** clause, the value of **sqlname** must be set to a valid field or column name. If this name was set by a previous **describe** statement, it must be retained or reset by the program. If the name refers to a hidden table field column, it must be directly set by the program. The varying-length name need not be padded with blanks.

For example, a dynamically named table field has been described, and the application always initializes any table field with a hidden 6-byte character column called "rowid." The code used to retrieve a row from the table field including the hidden column and **_state** variable would have to construct the two named columns:

```
...

rowid: [volatile] packed array[1..6] of Char;
rowstate: [volatile] Integer;

...

exec frs describe table :formname :tablename into :sqlda;

...

sqlda.sqld      := sqlda.sqld + 1;
col_num         := sqlda.sqld;

{ Set up to retrieve rowid }
sqlda.sqlvar[col_num].sqltype    := IISQ_CHA_TYPE;
sqlda.sqlvar[col_num].sqllen     := 6;
sqlda.sqlvar[col_num].sqldata    := iaddress(rowid);
sqlda.sqlvar[col_num].sqlind     := 0;
sqlda.sqlvar[col_num].sqlname    := 'rowid';

sqlda.sqld := sqlda.sqld + 1;
 col_num := sqlda.sqld;

{ Set up to retrieve _STATE }
sqlda.sqlvar[col_num].sqltype    := IISQ_INT_TYPE;
sqlda.sqlvar[col_num].sqllen     := 4;
sqlda.sqlvar[col_num].sqldata    := iaddress(rowstate);
sqlda.sqlvar[col_num].sqlind     := 0;
sqlda.sqlvar[col_num].sqlname    := '_state';

...

exec frs getrow :formname :tablename using descriptor :sqlda;
```

Advanced Processing

This section describes user-defined handlers. It includes information about user-defined error, dbevent, and message handlers as well as data handlers for large objects.

User-Defined Error, DBevent, and Message Handlers

You can use user-defined handlers to capture errors, messages, or events during the processing of a database statement. Use these handlers instead of the **sql whenever** statements with the SQLCA when you want to do the following:

- Capture more than one error message on a single database statement.

- Capture more than one message from database procedures fired by rules.

Trap errors, events, and messages as the DBMS raises them. If an event is raised when an error occurs during query execution, the WHENEVER mechanism detects only the error and defers acting on the event until the next database statement is executed.

User-defined handlers offer you flexibility. If, for example, you want to trap an error, you can code a user-defined handler to issue an **inquire_sql** to get the error number and error text of the current error. You can then switch sessions and log the error to a table in another session; however, you must switch back to the session from which the handler was called before returning from the handler. When the user handler returns, the original statement continues executing. User code in the handler cannot issue database statements for the session from which the handler was called.

The handler must be declared to return an integer. However, the preprocessor ignores the return value.

Syntax Notes:

The following syntax describes the three types of handlers:

```
exec sql set_sql (errorhandler = error_routine|0);
exec sql set_sql (dbeventhandler = event_routine|0);
exec sql set_sql (messagehandler = message_routine|0);
```

1. Errorhandler, dbeventhandler, and messagehandler denote a user-defined handler to capture errors, events, and database messages respectively, as follows:

   — error_routine is the name of the function the Ingres runtime system calls when an error occurs.

   — event_routine is the name of the function the Ingres runtime system calls when an event is raised.

   message_routine is the name of the function the Ingres runtime system calls whenever a database procedure generates a message.

   Errors that occur in the error handler itself do not cause the error handler to be re-invoked. You must use **inquire_sql** to handle or trap any errors that may occur in the handler.

2. Unlike regular variables, the handler must not be declared in an ESQL declare section; therefore, do not use a colon before the handler argument. (However, you must declare the handler to the compiler.)

3. If you specify a zero (0) instead of a name, the zero will unset the handler.

User-defined handlers are also described in the *SQL Reference Guide*.

Declaring and Defining User-Defined Handlers

The following example shows how to declare a handler for use in the **set_sql errorhandler** statement for ESQL/Pascal:

```
program TestProg(input, output);
exec sql include SQLCA;

     function Error_Func: Integer;
     exec sql begin declare section;
     var
     errnum : Integer;
     exec sql end declare section;

     begin
        exec sql inquire_sql (:errnum = ERRORNO);
        write ('Error number is ');
        writeln (errnum);
        Error_Func :=1;  {return value ignored}
     end;

begin
     exec sql connect dbname;
     exec sql set_sql (ERRORHANDLER = Error_Func);
     {                  }
     { ESQL will generate                           }
     {  IILQshSetHandler ( 1, %immed Error_Func);}
     {                                              }
. . .
end.
```

Sample Programs

The programs in this section are examples of how to declare and use user-defined data handlers in an ESQL/Pascal program. There are examples of a handler program, a Put Handler program, a Get Handler program and a dynamic SQL handler program.

Handler Program

This program inserts a row into the book table using the data handler Put_Handler to transmit the value of column chapter_text from a text file to the database. Then it selects the column chapter_text from the table book using the data handler Get_Handler to process each row returned.

```
program handler(input,output);
    exec sql include sqlca

-- Do not declare the data handlers nor the data handler argument
-- to the ESQL preprocessor

type

    String100 = packed array [1..100] of char;
        hdlr_rec = record

            argstr: String100;
            argint: Integer;
```

```
        end;

var

        hdlr_rec: hdlr_arg;

        exec sql begin declare section;
            indvar;      II_int2;
            seg_buf;    packed array [1...1000] of char;
            seg_len;    integer;
            data_end;   integer;
            max_len;    integer;
        exec sql end declare section;
```

Put Handler

This user defined handler shows how an application can use the put data handler to enter a chapter of a book from a text file into a database.

```
function Put_Handler(info: hdlr_rec) : Integer;

begin

        process information passed in via the info record...
        open file ...

data_end := 0;

        while (not end-of-file) do begin

                read segment from file into seg_buf...

            if (end-of-file) then begin
                data_end := 1;
            end;

            exec sql put data (segment = :seg_buf,
                        segmentlength = :seg_len,
                         dataend = :data_end);

end; {while}

. . .
close file...
set info record to return appropriate values...
....
Put_Handler := 0 {return value ignored}

end {Put Handler }
```

Get Handler

This user defined datahandler shows how an application can use the get data handler to enter a chapter of a book from a text file into a database.

```
function Get_Handler(info: hdlr_rec) :Integer;
begin
    ...
    process information passed in via the info record...
    open file ....
```

```
data_end := 0;
while (data_end = 0) do
begin
    exec sql get data (:seg_buf=segment,
            :seg_len = segmentlength,
             :data_end = dataend)
         with maxlength = :max_len;
        write segment to file...
end;
. . .
set info record to return appropriate values...
...
            Get_Handler := 0; {return value ignored }
    end;
begin
-- INSERT a long varchar value chapter_text into
-- the table book using the datahandler Put_Handler
-- The argument passed to the datahandler the record
-- hdlr_arg.
--
                ...
        ...
        exec sql insert into book (chapter_num, chapter_name, chapter_text)
                values (5, ‘One Dark and Stormy Night’,
                datahandler(Put_Handler(hdlr_arg)));
        ...

-- Select the long varchar column chapter_text from the table book.
-- The Datahandler (Get_handler) will be invoked for each non-null value of
-- column chapter_text retrieved. For null values the indicator variable
-- will be set to “-1” and the datahandler will not be called.
        ...
            ...
            exec sql select chapter_text into
                 datahandler(Get_Handler(hdlr)arg)):indvar
                 from book;
            exec sql begin;
                 process row....
            exec sql end;
            ...
end.
```

User-Defined Data Handlers for Large Objects

Use the following definitions when you code user-defined data handlers for large objects in Dynamic SQLprograms that use the **exec sql include sqlda** statement:

```
    constant IISQ_LVCH_TYPE = 22
    constant IISQ_HDLR_TYPE = 22

      type IIsqlhdlr = record
            sqlarg: [volatile] Integer;
            sqlhdlr: [volatile] Integer;

end;
```

Dynamic SQL Handler Program

The following is an example of a dynamic SQL handler program:

```
program dynamic_hdlr(input,output):

        exec sql include sqlca;
        exec sql include sqlda;

-- Do not declare the data handlers nor the data handler argument
-- to the ESQL preprocessor

type

         String100 = packed array [1..100] of char;
         hdlr_rec = record
                argstr: String100;
                argint: Integer;
            endr;

var

            function Put_Handler(hdlr_arg: hdrlr_rec):
                         integer;external;
            function Get_Handler(hdlr_arg: hdlr_rec):
                        integer;external;
            hdlr_rec:   hdlr_arg;

-- Declare SQLDA and IISQLHDLR

         sqlda:          IIsqlda;
         data_handler:   IIsqlhdlr;
         base_type:      integer;
         col_num:        integer;

-- Declare null indicator to ESQL

        exec sql begin declare section;
            ind_var:    integer;
            stmt_buf:   String100;
        exec sql end declare section;

        .   .

begin

-- Set the IISQLHDLR structure with the appropriate datahandler and
-- datahandler argument.

    data_handler.sqlhdlr = iaddress(Get_Handler)
    data_handler.sqlarg = iaddress(hdlr)arg)

-- Describe the statment into the SQLDA.

    stmt_buf = 'select * from book'.
    exec sql prepare stmt from :stmt_buf;
    exec sql describe stmt into SQLDA;

    . . .

-- Determine the base_type of the SQLDATA variables.

    col_num := 1;
    while (col_num <= sqlda.sqld) do begin

        with sqlda.sqlvar[col_num] do begin
```

```
                if (sqltype > 0) then
                        base_type := sqltype;
                else
                        base_type := -sqltype;

-- Set the sqltype, sqldata and sqlind for each column.
-- The Long Varchar Column chapter_text will be set to use a datahandler.

        if (base_type = IISQ_LVCH_TYPE) the
            sqltype = IISQ_HDLR_TYPE;
            sqldata = iaddress(data_handler_;
            sqlind = iaddress(indvar);
        else

        . . .
     end;

    end;

-- The Datahandler (Get_Handler) will be invoked for each non-null value
-- of column chapter_text retrieved.
-- For null values the indicator variable will be set to "-1" and
-- the datahandler will not be called.

    ...

    exec sql execute immediate :stmt_buf using :SQLDA
    exec sql begin
         process row...
    exec sql end;
    ...

end.
```

Preprocessor Operation

This section describes the operation of the Embedded SQL preprocessor for Pascal and the steps required to create, compile, and link an Embedded SQL program.

Command Line Operations

The following sections describe how to turn an embedded ESQL/Pascal source program into an executable program. These sections include commands that preprocess, compile, and link a program.

The Embedded SQL Preprocessor Command

The Pascal preprocessor is invoked by the following command line:

esqlp {*flags*} {*filename*}

where *flags* are

Flag	Description
-d	Adds debugging information to the runtime database error messages generated by Embedded SQL. The source file name, line number and statement in error will be printed with the error message.
-f[*filename*]	Writes preprocessor output to the named file. If no filename is specified, the output is sent to standard output, one screen at a time.
-l	Writes preprocessor error messages to the preprocessor's listing file, as well as to the terminal. The listing file includes preprocessor error messages and your source text in a file named *filename**.lis***, where *filename* is the name of the input file.
-lo	Like **-l**, but the generated Pascal code also appears in the listing file.
-o.*ext*	Specifies the extension given by the preprocessor to both the translated **include** statements in the main program and the generated output files. If this flag is not provided, the default extension is ".pas."If you use this flag in combination with the **-o** flag, then the preprocessor generates the specified extension for the translated include statements, but does not generate new output files for the include statements.
-o	Directs the preprocessor not to generate output files for include files. This flag does not affect the translated **include** statements in the main program. The preprocessor will generate a default extension for the translated **include** file statements unless you use the **-o.*ext*** flag.
-?	Shows what command line options are available for **esqlp**.
-s	Reads input from standard input and generates Pascal code to standard output. This is useful for testing statements you are not familiar with. If the **-l** option is specified with this flag, the listing file is called "stdin.lis." To terminate the interactive session, type **Ctrl Z**.
-sqlcode	Indicates the file declares ANSI SQL code. The ANSI-92 specification describes **SQLCODE** as a "deprecated feature" and recommends using the **SQLSTATE** variable.

Flag	Description
-[no]sqlcod	Tells the preprocessor not to assume a declared SQLCODE is for ANSI status information.
-w	Prints warning messages.
-wopen	This flag is identical to **-wsql=open**. However, **-wopen** is supported only for backwards capability. For more information, see **-wsql=open**.
-wsql= entry_SQL92open	Prints warning messages that indicate all non-entry SQL92 compliant syntax. Use *open* only with OpenSQL syntax. **-wsql = open** generates a warning if the preprocessor encounters an Embedded SQL statement that does not conform to OpenSQL syntax. (OpenSQL syntax is described in the *OpenSQL Reference Guide*.) This flag is useful if you intend to port an application across different Ingres Gateways. The warnings do not affect the generated code and the output file may be compiled. This flag does not validate the statement syntax for any SQL Gateway whose syntax is more restrictive than that of OpenSQL.

The Embedded SQL/Pascal preprocessor assumes that input files are named with the extension ".sp." You can override this default by specifying the file extension of the input file(s) on the command line. The output of the preprocessor is a file of generated Pascal statements with the same name and the extension ".pas."

If you enter the command without specifying any flags or a filename, Ingres displays a list of flags available for the command.

The following table present examples of the options available with **esqlp**.

Esqlp Command Examples

Command	Comment
esqlp file1	Preprocesses "file1.sp" to "file1.pas"
esqlp file2.xp	Preprocesses "file2.xp" to "file2.pas"
esqlp -l file3	Preprocesses "file3.sp" to "file3.pas" and creates listing "file3.lis"
esqlp -s	Accepts input from standard input

Command	Comment
esqlp -ffile4.out file4	Preprocesses "file4.sp" to "file4.out"
esqlp	Displays a list of flags available for this command

The Pascal Compiler

As mentioned above, the preprocessor generates Pascal code. You should use the VMS **pascal** command to compile this code. You can use most of the **pascal** command line options. You must not use the **g_floating** qualifier if real variables in the file are interacting with Ingres floating-point objects. You should also not use the **old_version** qualifier, because the preprocessor generates code for Version 3. Note, too, that many of the statements that the Embedded SQL/Pascal preprocessor generates are non-standard extensions provided by VAX/VMS. Consequently, you should not use the **standard** qualifier.

The following example preprocesses and compiles the file "test1." Note that both the Embedded SQL preprocessor and the Pascal compiler assume the default extensions.

```
$ esqlp test1
$ pascal/list test1
```

VMS

As of Ingres II 2.0/0011 (axm.vms/00) Ingres uses member alignment and IEEE floating-point formats. Embedded programs must be compiled with member alignment turned on. In addition, embedded programs accessing floating-point data (including the MONEY data type) must be compiled to recognize IEEE floating-point formats.

Note: Check the Readme file for any operating system specific information on compiling and linking ESQL/Pascal programs.

Linking an Embedded SQL Program

Embedded SQL programs require procedures from several VMS shared libraries in order to run properly. Once you have preprocessed and compiled an Embedded SQL program, you can link it. Assuming the object file for your program is called "dbentry," use the following link command:

```
$ link dbentry,-
  ii_system:[ingres.files]esql.opt/opt
```

Assembling and Linking Pre-Compiled Forms

The technique of declaring a pre-compiled form to the FRS is discussed in the *SQL Reference Guide* and in The SQL Communications Area section in this chapter. To use such a form in your program, you must also follow the steps described here.

In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED lets you select the name for the file. Once you have created the MACRO file this way, you can assemble it into linkable object code with the VMS command

macro *filename*

The output of this command is a file with the extension ".obj". You then link this object file with your program by listing it in the link command, as in the following example:

```
$ link formentry,-
  empform.obj,-
  ii_system:[ingres.files]esql.opt/opt
```

Linking an Embedded SQL Program without Shared Libraries

While the use of shared libraries in linking Embedded SQL programs is recommended for optimal performance and ease-of-maintenance, non-shared versions of the libraries have been included in case you require them. Non-shared libraries required by Embedded SQL are listed in the esql.noshare options file. The options file must be included in your link command *after* all user modules. Libraries must be specified in the order given in the options file.

The following example demonstrates the link command of an Embedded SQL program called "dbentry" that has been preprocessed and compiled:

```
$ link dbentry,-
  ii_system:[ingres.files]esql.noshare/opt
```

Placing User-written Embedded SQL Routines in Shareable Images

When you plan to place your code in a shareable image, note the following about the **psect** attributes of your global or external variables.

- As a default, some compilers mark global variables as shared (SHR: every user who runs a program linked to the shareable image sees the same variable) and others mark them as not shared (NOSHR: every user who runs a program linked to the shareable image gets their own private copy of the variable).

- Some compilers support modifiers you can place in your source code variable declaration statements to explicitly state which attributes to assign a variable.
- The attributes that a compiler assigns to a variable can be overridden at link time with the **psect_attr** link option. This option overrides attributes of all variables in the **psect**.

Consult your compiler reference manual for further details.

Include File Processing

The Embedded SQL **include** statement provides a means to include external files in your program's source code. Its syntax is:

exec sql include *filename;*

filename is a quoted string constant specifying a file name, or a logical name that points to the file name. If no extension is given to the filename (or to the file name pointed at by the logical name), the default Pascal input file extension ".sp" is assumed.

This statement is normally used to include variable declarations, although it is not restricted to such use. For more details on the **include** statement, see the *SQL Reference Guide*.

The included file is preprocessed and an output file with the same name but with the default output extension ".pas" is generated. You can override this default output extension with the **-o.*ext*** flag on the command line. The preprocessed output of the include statement is the Pascal **%include** directive. If you use the **-o** flag (without an extension), then the output file is not generated for the **include** statement. This is useful for program libraries that use VMS MMS dependencies.

For example, assume that no overriding output extension was explicitly given on the command line. The Embedded SQL statement:

```
exec sql include 'employee.dcl';
```

is preprocessed to the Pascal statement:

```
%include 'employee.pas'
```

and the file "employee.dcl" is translated into the Pascal file "employee.pas".

As another example, assume that a source file called "inputfile" contains the following **include** statement:

```
exec sql include 'mydecls';
```

You can define the name "mydecls" as a system logical name pointing to the file "dra1:[headers]myvars.sp" by means of the following command at the system level:

```
$ define mydecls dra1:[headers]myvars
```

Because the extension ".sp" is the default input extension for Embedded SQL **include** files, it need not be specified when defining a logical name for the file.

Assume now that "inputfile" is preprocessed with the command:

```
$ esqlp -o.inc inputfile
```

The command line specifies ".inc" as the output file extension for include files. As the file is preprocessed, the **include** statement shown earlier is translated into the Pascal statement:

```
%include 'dra1:[headers]myvars.inc'
```

and the Pascal file "dra1:[headers]myvars.inc" is generated as output for the original include file, "dra1:[headers]myvars.sp".

You can also specify include files with a relative path. For example, if you preprocess the file "dra1:[mysource]myfile.sp," the Embedded SQL statement:

```
exec sql include '[-.headers]myvars.sp';
```

is preprocessed to the Pascal statement:

```
%include '[-.headers]myvars.pas'
```

and the Pascal file "dra1:[headers]myvars.pas" is generated as output for the original include file, "dra1:[headers]myvars.sp."

Including Source Code with Labels

Some Embedded SQL statements generate labels. If you include a file containing such statements, you must be careful to include the file only once in a given Pascal scope. Otherwise, you may find that the compiler later complains that the generated labels are multiply defined in that scope.

The statements that generate labels are the Embedded SQL block-type statements, which are:

select-loop
display
formdata
tabledata
unloadtable
submenu

You must also issue the **exec sql label** statement in the same scope as the label-generating statement.

Coding Requirements for Writing Embedded SQL Programs

The following sections discuss coding requirements for writing Embedded SQL statements.

Comments Embedded in Pascal Output

Each Embedded SQL statement generates one comment and a few lines of Pascal code. You may find that the preprocessor translates 50 lines of Embedded SQL into 200 lines of Pascal. This can confuse the program developer who is trying to debug the original source code. To facilitate debugging, each group of Pascal statements associated with a particular statement is preceded by a comment corresponding to the original Embedded SQL source. (Note that only *executable* Embedded SQL statements are preceded by a comment.) Each comment is one line long and informs the reader of the file name, line number, and type of statement in the original source file.

One consequence of the generated comment is that you cannot comment out embedded statements by putting the opening comment delimiter on an earlier line. You have to put the delimiter on the same line, before the **exec** keyword, to cause the preprocessor to treat the complete statement as a Pascal comment.

Embedding Statements Inside Pascal If Blocks

As mentioned above, the preprocessor may produce several Pascal statements for a single Embedded SQL statement. However, all the statements generated by the preprocessor are enclosed in Pascal **begin** and **end** delimiters, composing a Pascal block. Thus the statement:

```
if (not dba) then
    exec sql select passwd
         into :passwd
         from security
         where usrname = :userid;
```

will produce legal Pascal code, even though the SQL **select** statement produces more than one Pascal statement. However, two or more Embedded SQL statements will generate multiple Pascal blocks, so you must delimit them yourself, just as you would delimit two Pascal statements in a single **if** block. For example:

```
if (not dba) then
begin
    exec frs message 'Confirming your user id';
    exec sql select passwd
```

```
              into :passwd
              from security
              where usrname = :userid;
end;
```

Note that, because the preprocessor generates a Pascal block for every Embedded SQL statement, the Pascal compiler may generate the error "Internal Table Overflow" when a single procedure has a very large number of Embedded SQL statements and local variables. You can correct this problem by splitting the file or procedure into smaller components.

All Embedded SQL statements must be terminated by a semicolon. Therefore, because Pascal does not permit semicolons before the **else** clause of an **if** statement, you must surround any single Embedded SQL statement that precedes an **else** clause with a Pascal **begin**-**end** block. For example, the following **if** statement will cause a Pascal error:

```
if error then
    exec frs message 'Error occurred';
            {Semicolon required by Embedded SQL}
else
    exec frs message 'No error occurred';
```

By delimiting the **then** clause with **begin**-**end**, you eliminate the error:

```
if error then
begin
    exec frs message 'Error occurred';
        {Semicolon required by Embedded SQL}
end {
    |   ... but that's okay because
    |   there's no semicolon here
    }
else
    exec frs message 'No error occurred';
```

Embedded SQL Statements That Do Not Generate Code

The following Embedded SQL declarative statements do not generate any Pascal code:

declare cursor
declare statement
declare table
whenever

These statements must not contain labels. Also, they must not be coded as the only statements in Pascal constructs that do not allow *null* statements. For example, coding a **declare cursor** statement as the only statement in a Pascal **if** statement not bounded by **begin** and **end** would cause compiler errors:

```
if (using_database) then
    exec sql declare empcsr cursor for
         select ename from employee;
else
```

```
    writeln('You have not accessed the database');
```

The code generated by the preprocessor would be:

```
if (using_database) then
else
    writeln('You have not accessed the database');
```

This is an illegal use of the Pascal **else** clause.

Embedded SQL/Pascal Preprocessor Errors

To correct most errors, you may wish to run the Embedded SQL preprocessor with the listing (**-l**) option on. The listing will be sufficient for locating the source and reason for the error.

For preprocessor error messages specific to Pascal, see Preprocessor Error Messages in this chapter.

Preprocessor Error Messages

The following is a list of error messages specific to Pascal.

E_DC000A

"Table 'employee' contains column(s) of unlimited length."

Explanation: Character strings(s) of zero length have been generated. This causes a compile-time error. You must modify the output file to specify an appropriate length.

E_E20001

"PASCAL attribute conflict in declaration of size for '%0c'."

Explanation: The program has specified conflicting size attributes for this object. For example, the following declaration is erroneous because of the attempt to extend the attribute size of the type:

```
'smaller': typesmaller = [byte] 1..100;
varbigger : [word] smaller;
```

E_E20002

"PASCAL subrange conflict. Upper and lower bounds are not the same type or they are not an ordinal type."

Explanation: Both bounds of a subrange declaration must be of the same ordinal type (single character or integer). If the subrange bounds types are different or if they are not ordinal types, the preprocessor will use the type of the second bound and accept the usage of variables declared with this subrange type. This will cause an error in later PASCAL compilation.

E_E20003

"Mismatching statement at end of PASCAL subprogram. Check balanced subprogram headers and END pairs."

Explanation: You may have an exec sql end statement that is not balanced by a exec sql label statement. These subprogram delimiters provide scoping for PASCAL labels generated by the preprocessor. If you had syntax errors on the exec sql label statement then correct those errors and preprocess the file again.

E_E20005

"PASCAL character array '%0c' must be PACKED or VARYING."

Explanation: A string referenced in an embedded statement must be either a PACKED ARRAY OF CHAR, a VARYING OF CHAR or a single CHAR. You have used a non-packed ARRAY OF CHAR as an embedded string variable. Convert the variable declaration to either PACKED or VARYING, or subscript the array to reference only one element.

E_E20006

"Extraneous semicolon in PASCAL declaration ignored."

Explanation: Only one semicolon is allowed between components of a record declaration. The preprocessor ignores the extra semicolons. You should delete the extra semicolon in your source code.

E_E20007

"PASCAL dimension of '%0c' is %1c, but subscripted %2c times."

Explanation: You have not referenced the specified variable with the same number of subscripts as the number of dimensions with which the variable was declared. This error indicates that you have failed to subscript an array, or you have subscripted a non-array. The preprocessor does not parse declaration dimensions or subscript expressions.

E_E20008

"Incorrect indirection of PASCAL variable '%0c'. Variable is declared with indirection of %1c, but dereferenced (^) %2c time(s)."

Explanation: This error occurs when the address or value of a variable is incorrectly expressed because of faulty indirection. For example, the name of an integer pointer has been given instead of the variable that the pointer was pointing at. Either redeclare the variable with the intended indirection (and check any implicit indirection in the type), or change its use in the current statement.

E_E20009

"PASCAL Pass 2 failure on INCLUDE file. The maximum INCLUDE nesting exceeded %0c."

Explanation: The PASCAL preprocessor must take a second pass in order to declare implicitly generated labels. If the source file referenced embedded INCLUDE files, then the second pass needs to generate labels into those files. Consequently there is a maximum nesting limit of INCLUDE files. Try reorganizing your files to create a flatter source file structure.

E_E2000B	"PASCAL Pass 2 open file failure. Cannot pass information from file '%0c' to '%1c'." **Explanation:** The PASCAL preprocessor must take a second pass in order to declare implicitly generated labels. Because there is a temporary file involved, and this file has a fixed name, you should avoid running the preprocessor more than once in the same directory. This error may also occur if the intermediate file disappeared, the system protections of the current directory are too restrictive or have changed, or if the original input file was moved between the first and second pass of the preprocessor.
E_E2000C	"PASCAL Pass 2 file inconsistency. Mismatching number of label markers in '%0c'." **Explanation:** The PASCAL preprocessor must take a second pass in order to declare implicitly generated labels. There was a difference between the number of label declaration sections the preprocessor expected to generate and the number of markers found in the intermediate file. This may be caused by an embedded INCLUDE statement that requires its own scope for label generation. If there were nested INCLUDE statements whose files required labels, try to flatten them out into larger source files.
E_E2000D	"Missing PASCAL keyword '%0c' in declaration." **Explanation:** You did not use the specified keyword, or you did not make the word known to the preprocessor. If there are no other errors the preprocessor will generate correct PASCAL code.
E_E2000F	"Can not use indirection (^) on an undeclared PASCAL variable '%0c'." **Explanation:** You have used pointer indirection on a name that was not declared as a PASCAL variable to the preprocessor. If this really is a variable you should make its declaration known to the preprocessor.
E_E20010	"Can not subscript ([]) an undeclared PASCAL variable '%0c'." **Explanation:** You have used array subscription on a name that was not declared as a PASCAL variable to the preprocessor. If this really is a variable you should make its declaration known to the preprocessor.
E_E20011	"Can not subscript VARYING PASCAL variable '%0c'." **Explanation:** Elements of a varying-length character string array cannot be passed to the runtime system. If you need to pass a single element then declare the array as a plain array (not PACKED nor VARYING).

E_E20012 "Scalar PASCAL type required for conformant schema bounds type."

Explanation: PASCAL requires that bounds expressions of conformant arrays be of a scalar type. You must choose a scalar type, such as a single character or an integer.

E_E20013 "PASCAL object '%0c' is not a variable."

Explanation: You have used the specified name as an embedded variable, but you have not declared it to the preprocessor. This may also be a scope problem. Make sure you have typed the name correctly, declared the variable to the preprocessor and have used it in its scope.

E_E20014 "Too many comma separated names in declaration. Maximum number of names is %0c."

Explanation: The declaration of a comma-separated list of names in a declaration is too long. For example: vara, b, ... N : Integer;Try breaking up the declaration into groups.

E_E20018 "Last PASCAL record member referenced in '%0c' is unknown."

Explanation: The last record member referenced is not a member of the current record. Make sure you have spelled the member name correctly, and that it is a member of the specified record.

E_E20019 "Unclosed PASCAL block. There are %0c unbalanced subprogram headers."

Explanation: The end of the file was reached with some program blocks left open. Make sure you have an END statement for each subprogram header or embedded LABEL statement.

E_E2001A "PASCAL %0c '%1c' is not yet defined. An INTEGER is assumed."

Explanation: The specified TYPE or CONST name has not yet been declared. Make sure that all types and constants are defined before use. Forward type declarations (such as pointers to undefined types) are an exception.

E_E2001B "Underflow of comma separated name list in declaration."

Explanation: The stack used to store comma-separated names in declarations has been corrupted. Try rearranging the list of names in the declaration.

E_E2001C "PASCAL variable '%0c' is of unsupported type SET or QUADRUPLE."

Explanation: You may declare variables of type SET And QUADRUPLE, but you may not use them in embedded statements. The declarations are only allowed so that you can declare records with components of those types. If those variables need to interact with INGRES, then declare the SET variable as an ARRAY OF BOOLEAN, and the QUADRUPLE variable as a DOUBLE.

E_E20022 "PASCAL variable '%0c' is a record, not a scalar value."

Explanation: The named variable refers to a record. It was used where a variable must be used to retrieve data from INGRES. This error may also cause a syntax error on any subsequent record components that are referenced.

E_E20023 "No embedded LABEL statement for current scope but labels have been generated."

Explanation: The PASCAL preprocessor must take a second pass in order to declare implicitly generated labels. If labels were implicitly generated then the preprocessor needs to know where to declare them on the second pass. That is why one must issue the embedded LABEL statement (and corresponding END statement) in each subprogram that issues an embedded block-structured statement. If you did not issue the EXEC SQL LABEL statement, the generated labels will be marked as undeclared by the PASCAL compiler.

Sample Applications

This section contains sample applications.

The Department-Employee Master/Detail Application

This application uses two database tables joined on a specific column. This typical example of a department and its employees demonstrates how to process two tables as a master and a detail.

The program scans through all the departments in a database table, in order to reduce expenses. Based on certain criteria, the program updates department and employee records. The conditions for updating the data are the following:

Departments:

- If a department has made less than $50,000 in sales, the department is dissolved.

Employees:

- If an employee was hired since the start of 1985, the employee is terminated.
- If the employee's yearly salary is more than the minimum company wage of $14,000 and the employee is not nearing retirement (over 58 years of age), the employee takes a 5% pay cut.

- If the employee's department is dissolved and the employee is not terminated, the employee is moved into a state of limbo to be resolved by a supervisor.

This program uses two cursors in a master-detail fashion. The first cursor is for the Department table, and the second cursor is for the Employee table. Both tables are described in **declare table** statements at the start of the program. The cursors retrieve all the information in the tables, some of which is updated. The cursor for the Employee table also retrieves an integer date interval whose value is positive if the employee was hired after January 1, 1985.

Each row that is scanned, from both the Department table and the Employee table, is recorded in an output file. This file serves both as a log of the session and as a simplified report of the updates that were made.

Each section of code is commented for the purpose of the application and also to clarify some of the uses of the Embedded SQL statements. The program illustrates table creation, multi-statement transactions, all cursor statements, direct updates and error handling.

```
program Departments( input, output );
exec sql include sqlca;

{The department table}
exec sql declare dept table
     (name          char(12)   not null,  {Department name}
      totsales      money      not null,  {Total sales}
      employees     smallint   not null); {Number of employees}

{The employee table}
exec sql declare employee table
     (name          char(20)   not null,  {Employee name}
      age           integer1   not null,  {Employee age}
      idno          integer1   not null,  {Unique employee id}
      hired         date       not null,  {Date of hire}
      dept          char(12)   not null,  {Department of work}
      salary        money      not null); {Yearly salary}

{"State-of-Limbo" for employees who lose their department}
exec sql declare toberesolved table
     (name          char(20)   not null,  {Employee name}
      age           integer1   not null,  {Employee age}
      idno          integer1   not null,  {Unique employee id}
      hired         date       not null,  {Date of hire}
      dept          char(12)   not null,  {Department of work}
      salary        money      not null); {Yearly salary}
label
      exit_program;
exec sql begin declare section;
type
     String12 = varying[12] of Char;
     String20 = varying[20] of Char;
     String25 = varying[25] of Char;
     String200 = varying[200] of Char;
     Short_Short_Integer = [byte] -128 .. 127;
     Short_Integer = [word] -32768 .. 32767;
exec sql end declare section;

{
```

```
| Procedure: Process_Expenses (MAIN)
| Purpose:   Main body of the application. Initialize the database,
|            process each department, and terminate the session.
| Parameters:
|            None
}

procedure Process_Expenses;
type
    File_type = Text;
var
    log_file: File_type; {Log file to which to write.}

    {
    | Procedure:  Init_Db
    | Purpose:    Initialize the database.
    |             Connect to the database and abort on error.
    |             Before processing departments and employees,
    |             create the table for employees who
    |             lose their department, "toberesolved".
    | Parameters: None
    }

procedure Init_Db;
begin

        exec sql whenever sqlerror stop;
        exec sql connect personnel;

        {Create the table.}
         writeln(log_file,
                'Creating ''To_Be_Resolved'' table.');
         exec sql create table toberesolved
                        (name    char(20) not null,
                         age     integer1 not null,
                         idno    integer  not null,
                         hired   date not null,
                         dept    char(12) not null,
                         salary  money not null);
end; {Init_Db}

{
| Procedure:  End_Db
| Purpose:    Commit the multi-statement transaction and
|             end access to the database.
| Parameters: None
}

procedure End_Db;
begin
    exec sql commit;
    exec sql disconnect;
end; {End_Db}

{
| Procedure:  Close_Down
| Purpose:    Error handler called any time after Init_Db has been
|             successfully completed. In all cases, print the
|             cause of the error and abort the transaction,
|             backing out changes. Note that disconnecting
|             from the database will implicitly close any
|             open cursors.
| Parameters: None.
}

procedure Close_Down;
```

```
        exec sql begin declare section;
        var
                errbuf: String200;
        exec sql end declare section;
begin
        {Turn off error handling here}
        exec sql whenever sqlerror continue;

        exec sql inquire_sql (:errbuf = ERRORTEXT);
        writeln( 'Closing Down because of database error.' );
        writeln( errbuf );

        exec sql rollback;
        exec sql disconnect;

        goto exit_program;                       {no return}
end; {Close_Down}

{
| Procedure: Process_Employees
| Purpose:   Scan through all the employees for a
|            particular department. Based on given
|            conditions, the employee may be terminated or
|            take a salary reduction.
|            1. If an employee was hired since 1985,
|               the employee is terminated.
|            2. If the employee's yearly salary is more
|               than the minimum company wage of $14,000
|               and the employee is not close to retirement
|               (over 58 years of age), the employee
|               takes a 5% salary reduction.
|            3. If the employee's department is dissolved
|               and the employee is not terminated,
|               the employee is moved into the
|               "toberesolved" table.
| Parameters:
|            dept_name    - Name of current department.
|            deleted_dept - Is department dissolved?
|            emps_term    - Set locally to record how many
|                           employees were terminated
|                           for the current department.
}

procedure Process_Employees
        (dept_name:     Varying[ub] of Char;
         deleted_dept:  Boolean;
         var emps_term: Integer);

    label
        Close_Emp_Csr;
    exec sql begin declare section;
    const
        salary_reduc = 0.95;
    type
        {Emp_Rec corresponds to the "employee" table}
         Emp_Rec = record
            name:                       String20;
            age:                Short_Short_Integer;
            idno:       Integer;
            hired:      String25;
            salary:     Real;
            hired_since_85: Integer;
        end;
    var
        erec:  Emp_Rec;
            dname: String12;
```

```
    exec sql end declare section;

    const
        min_emp_salary = 14000.00;
        nearly_retired = 58;
    var
        title:    String12; {Formatting values}
        descript: String25;

    {
    | Note the use of the INGRES function to find out
    | who has been hired since 1985
    }

    exec sql declare empcsr cursor for
        select name, age, idno, hired, salary,
            int4(interval('days',
                hired-date('01-jan-1985')))
        from employee
        where dept = :dname
        for direct update of name, salary;

begin {Process_Employees}
    dname := dept_name;

    {
    | All errors from this point on close down
    | the application
    }
    exec sql whenever sqlerror call Close_Down;
    exec sql whenever not found goto Close_Emp_Csr;

    exec sql open empcsr;

    emps_term := 0; {Record how many}
    while (sqlca.sqlcode = 0) do
    begin
        exec sql fetch empcsr into :erec;

        if (erec.hired_since_85 > 0) then
            begin
                exec sql delete from employee
                        where current of empcsr;
                title := 'Terminated: ';
                descript := 'Reason: Hired since 1985.';
                emps_term := emps_term + 1;
        end else if (erec.salary > min_emp_salary) then
        begin {Will reduce salary if not nearly retired}
            if (erec.age < nearly_retired) then
            begin
                exec sql update employee
                        set salary =
                                salary * :salary_reduc
                        where current of empcsr;
                title := 'Reduction: ';
                descript := 'Reason: Salary. ';
            end else
            begin
                {Do not reduce salary}
                title := 'No Changes: ';
                descript := 'Reason: Retiring. ';
            end;
        end else {Else leave employee as is}
        begin
            title := 'No Changes: ';
            descript := 'Reason: Salary. ';
```

```
        end;

        {Was employee's department dissolved?}
        if (deleted_dept) then
        begin
            exec sql insert into toberesolved
                select *
                from employee
                where idno = :erec.idno;

            exec sql delete from employee
                where current OF empcsr;
        end;

        {Log the employee's information}
        write(log_file, ' ', title, ' ');
        write(log_file, erec.idno:6);
        write(log_file, ', ', erec.name, ', ');
        write(log_file, erec.age:3);
        write(log_file, ', ');
        write(log_file, erec.salary:8:2);
        writeln(log_file, ' ; ', descript);
    end;

Close_Emp_Csr:
    exec sql whenever not found continue;
    exec sql close empcsr;
end;

{
| Procedure:  Process_Depts
| Purpose:    Scan through all the departments, processing each one.
|             If the department has made less than $50,000 in sales,
|             then the department is dissolved.
|             For each department, process all the employees
|             (they may even be moved to another database table).
|             If an employee was terminated, then update the department's
|             employee counter.
| Parameters: None
}


procedure Process_Depts;
    exec sql begin declare section;
    type
        {Dept_Rec corresponds to the "dept" table}
         Dept_Rec = record
                name:      String12;
                totsales:  Double;
                employees: Short_Integer;
        end;
    var
        dept:      Dept_Rec;
        emps_term: Integer;     {Employees terminated}
    exec sql end declare section;

    label
        Close_Dept_Csr;
    const
        min_tot_sales = 50000.00;
    var
        deleted_dept: Boolean;  {Was the dept deleted?}
        dept_format:  String20; {Formatting value}

        exec sql declare deptcsr cursor for
             select name, totsales, employees
```

```
                from dept
                for direct update of name, employees;
        begin {Process_Depts}
            emps_term := 0;

            {All errors from this point on close down the application}
            exec sql whenever sqlerror call Close_Down;
            exec sql whenever not found goto Close_Dept_Csr;

            exec sql open deptcsr;

            while (sqlca.sqlcode = 0) do
            begin
                exec sql fetch deptcsr into :dept;

                {Did the department reach minimum sales?}
                if (dept.totsales \ min_tot_sales) then
                begin
                    exec sql delete from dept
                            where current of deptcsr;
                    deleted_dept := TRUE;
                    dept_format := ' -- DISSOLVED --';
                end else
                begin
                    deleted_dept := FALSE;
                    dept_format := ' ';
                end;

                {Log what we have just done}
                write(log_file,
                    'Department: ', dept.name, ', Total Sales: ');
                write(log_file, dept.totsales:12:3);
                writeln(log_file, dept_format);

                {Now process each employee in the department}
                Process_Employees(dept.name,
                    deleted_dept, emps_term);

                {If employees were terminated, record this fact}
                if ((emps_term > 0) and (not deleted_dept)) then
                begin
                    exec sql update dept
                            set employees = :dept.employees - :emps_term
                            where current of deptcsr;
                    end;
                end;

    Close_Dept_Csr:
            exec sql whenever not found continue;
            exec sql close deptcsr;
    end;                            {Process_Depts}

    begin                           {Process_Expenses}
            writeln('Entering application to process expenses.');
            open(file_variable := log_file, file_name := 'expenses.log');
            rewrite( log_file );
            Init_Db;
            Process_Depts;
            End_Db;
            close(log_file);
            writeln('Completion of application.');
    end;                            {Process_Expenses}
```

```
begin                               {MAIN program}
        Process_Expenses;
exit_program:;
 end. {MAIN}
```

The Table Editor Table Field Application

This application edits the Person table in the Personnel database. It is a forms application that allows the user to update a person's values, remove the person, or add new persons. Various table field utilities are provided with the application to demonstrate how they work.

The objects used in this application are:

Object	Description
personnel	The program's database environment.
person	A table in the database, with three columns: name(**char(20)**) age (**smallint**) number (**integer**) Number is unique.
personfrm	The VIFRED form with a single table field.
persontbl	A table field in the form, with two columns: name (**char(20)**) age (**integer**) When initialized, the table field includes the hidden column: number **(integer)**
personrec	A local structure, whose members correspond in name and type to columns in the Person table and the persontbl table field.

When the application starts, a database cursor is opened to load the table field with data from the Person table. After the table field has been loaded, the user can browse and edit the displayed values. Entries can be added, updated, or deleted. When finished, the values are unloaded from the table field, and the user's updates are transferred back into the Person table.

```
program Table_Edit( input, output );
exec sql include sqlca;

exec sql declare person table
    (name char(20),      {Person name}
     age smallint,       {Age}
     number integer);    {Unique id number}
```

```
exec frs label exit_label;
exec sql begin declare section;
const
        not_found = 100; {SQLCA value for no rows}
type
        String1 = packed array [1..1] of Char;
        String13 = packed array [1..13] of Char;
        String20 = packed array [1..20] of Char;
        String100 = packed array [1..100] of Char;
        Short_Integer = [word] -32768 .. 32767;

        {Table field row states}
        Row_States = (
            row_undef,    {Empty or undefined row}
            row_new,      {Appended by user}
            row_unchange, {Loaded by program, not updated}
            row_change,   {Loaded by program and updated}
            row_delete    {Deleted by program}
        );
var
        {Person information corresponds to "person" table}
        pname:   String20;      {Full name}
        page:    Short_Integer; {Age}
        pnumber: Integer;       {Unique person number}
        pmaxid:  Integer;       {Maximum person id number}

        {Table field entry information}
        state:   Row_States;    {State of data set row}
        recnum,                 {Record number}
        lastrow: Integer;       {Last row in table field}

        {Utility buffers}
        search:   String20;     {Name to find in search loop}
        password: String13;     {Password buffer}

        msgbuf:   String100;    {Message buffer}
        respbuf:  String1;      {Response buffer}
exec sql end declare section;

var
        {Error handling variables for database updates}
        update_error: Boolean;  {Error in updates?}
        update_commit: Boolean; {Commit updates}

{
| Load the information from the "person" table into the person variables.
| Also save away the maximum person ID number.
}

function Load_Table : Integer;
        label
            Load_End;

        exec sql begin declare section;
        var
            {Person information}
            pname:      String20;       {Full name}
            page:       Short_Integer;  {Age}
            pnumber:    Integer;        {Unique person number}
            maxid:      Integer;        {Maximum person id number}
        exec sql end declare section;

        exec sql declare loadtab cursor for
            select name, age, number
            from person;
```

```
        {Set up error handling for loading procedure}
        exec sql whenever sqlerror goto Load_End;
        exec sql whenever not found goto Load_End;

begin                          {Load_Table}
        exec frs message 'Loading Person Information . . .';

        {Fetch the maximum person id number for later use}
        exec sql select max(number)
            into :maxid
            from person;

        exec sql open loadtab;

        while (sqlca.sqlcode = 0) do
        begin
            {Fetch data into record and load table field}
            exec sql fetch loadtab into :pname, :page, :pnumber;

            exec frs loadtable personfrm persontbl
                (name = :pname, age = :page, number = :pnumber);
        end;

Load_End:
        exec sql whenever sqlerror continue;
        exec sql close loadtab;

        Load_Table := maxid;
end; {Load_Table}

begin                  {Table_Edit}
        {Set up error handling for main program}
        exec sql whenever sqlwarning continue;
        exec sql whenever not found continue;
        exec sql whenever sqlerror stop;

        {Start up INGRES and the INGRES/FORMS system}

        exec sql connect 'personnel';

        exec frs forms;

        update_error := FALSE;
        update_commit := TRUE;

        {Verify that the user can edit the "person" table}
        exec frs prompt noecho
            ('Password for table editor: ', :password);
        if (password <> 'MASTER_OF_ALL') then
        begin
            exec frs message 'No permission for task. Exiting . . .';
            exec frs endforms;
            exec sql disconnect;
            goto exit_label;
        end;
        exec frs message 'Initializing Person Form . . .';
        exec frs forminit personfrm;

        {
        | Initialize "persontbl" table field with a data set
        | in FILL mode so that the runtime user can append rows.
        | To keep track of events occurring to original rows that
        | will be loaded into the table field, hide the unique
        | person number.
        }
```

```
        exec frs inittable personfrm persontbl fill (number = integer);

        pmaxid := Load_Table;

        {Display the form and allow runtime editing}

        exec frs display personfrm update;
        exec frs initialize;
        exec frs begin;
            {
            | Provide menu items, as well as system FRS keys,
            | to scroll to both extremes of the table field.
            }
            exec frs scroll personfrm persontbl to 1;
        exec frs end;

exec frs activate menuitem 'Top';
exec frs begin;
        exec frs scroll personfrm persontbl TO 1; {Backward}
exec frs end;

exec frs activate menuitem 'Bottom';
exec frs begin;
        exec frs scroll personfrm persontbl to end; {Forward}
exec frs end;

exec frs activate menuitem 'Remove';
exec frs begin;
        {
        | Remove the person in the row the user's cursor
        | is on. If there are no persons, exit operation
        | with message. Note that this check cannot
        | really happen, as there is always at least one
        | UNDEFINED row in FILL mode.
        }

        exec frs inquire_frs table personfrm
                (:lastrow = lastrow(persontbl));
        if (lastrow = 0) then
        begin
            exec frs message 'Nobody to Remove';
            exec frs sleep 2;
            exec frs resume field persontbl;
        end;

        exec frs deleterow personfrm persontbl; {Recorded for later}
exec frs end;

exec frs activate menuitem 'Find';
exec frs begin;
        {
        | Scroll user to the requested table field entry.
        | Prompt the user for a name, and if one is typed
        | in, loop through the data set searching for it.
        }

        search := ' ';
        exec frs prompt ('Person''s name : ', :search);
        if (search[1] = ' ') then
            exec frs resume field persontbl;

            exec frs unloadtable personfrm persontbl
                (:pname = name, :recnum = _record, :state = _state);
            exec frs begin;
                {Do not compare with deleted rows}
```

```
                if ((state <> row_delete) and (pname = search)) then
                  begin
                        exec frs scroll personfrm persontbl to :recnum;
                        exec frs resume field persontbl;
                  end;
            exec frs end;
            {Fell out of loop without finding name. Issue error.}
            msgbuf := 'Person ''' + search +
                 ''' not found in table [HIT RETURN] ';
            exec frs prompt noecho (:msgbuf, :respbuf);
exec frs end;

exec frs activate menuitem 'Exit';
exec frs begin;
     exec frs validate field persontbl;
     exec frs breakdisplay;
exec frs end;
exec frs finalize;

{
| Exit person table editor and unload the table field.
| If any updates, deletions or additions were made,
| duplicate these changes in the source table.
| If the user added new people, assign a unique person ID
| to each person before adding the person to the table.
| To do this, increment the previously-saved maximum ID number
| with each insert.
}

{Do all the updates in a transaction}
exec sql savepoint savept;

update_commit := TRUE;

{
| Hard code the error handling in the UNLOADTABLE loop,
| as we want to cleanly exit the loop.
}
exec sql whenever sqlerror continue;

exec frs message 'Exiting Person Application . . .';

exec frs unloadtable personfrm persontbl
        (:pname = name, :page = age,
         :pnumber = number, :state = _state);
exec frs begin;
        case state of
            row_new:
            begin
                {Filled by user. Insert with new unique id.}
                pmaxid := pmaxid + 1;
                exec sql insert into person (name, age, number)
                        values (:pname, :page, :pmaxid);
            end;

            row_change:
            begin
                {Updated by user. Reflect in table.}
                exec sql update person set
                        name = :pname, age = :page
                        where number = :pnumber;
            end;

            row_delete:
            {
            | Deleted by user, so delete from table. Note that
```

```
                | only original rows, not rows appended at runtime,
                | are saved by the program.
                }
                exec sql delete from person
                    where number = :pnumber;

                otherwise
                {
                | Else UNDEFINED or UNCHANGED --
                | No updates required.
                }
                ;
            end; {case}

            {
            | Handle error conditions -
            | If an error occurred, abort the transaction.
            | If no rows were updated, inform user and
            | prompt for continuation.
            }

    if (sqlca.sqlcode < 0) then {Error}
        begin
            exec sql inquire_sql (:msgbuf = errortext);
            exec sql rollback to savept;
            update_error := true;
            update_commit := false;
            exec frs endloop;
        end else if (sqlca.sqlcode = NOT_FOUND) then
        begin
            msgbuf := 'Person " + pname +
                 " not updated. Abort all updates? ';
            exec frs prompt noecho (:msgbuf, :respbuf);
            if ((respbuf = 'Y') or (respbuf = 'y')) then
            begin
                update_commit := false;
                exec sql rollback to savept;
                exec frs endloop;
            end;
        end;
        exec frs end;

        if (update_commit) then
            exec sql commit; {Commit the updates}

        exec frs endforms; {Terminate the FORMS and INGRES}
        exec sql disconnect;

        if (update_error) then
        begin
            writeln( 'Your updates were aborted because of error:' );
            writeln( msgbuf );
        end;
    exit_label:;
    exec frs end. {Table_Edit}
```

The Professor-Student Mixed Form Application

This application lets the user browse and update information about graduate students who report to a specific professor. The program is structured in a master/detail fashion, with the professor being the master entry, and the students the detail entries. The application uses two forms—one to contain general professor information and another for detailed student information.

The objects used in this application are:

Object	Description
personnel	The program's database environment.
professor	A database table with two columns: pname (**char(25)**) pdept (**char(10)**) See its **declare table** statement in the program for a full description.
student	A database table with seven columns: sname (**char(25)**) sage (**integer1**) sbdate (**char(25)**) sgpa (**float4**) sidno (**integer**) scomment (**varchar(200)** sadvisor (**char(25)**) See its **declare table** statement for a full description. The sadvisor column is the join field with the pname column in the Professor table.
masterfrm	The main form has fields pname and pdept, which correspond to the information in the Professor table, and table field studenttbl. The pdept field is display-only.
studenttbl	A table field in "masterfrm" with two columns, "sname" and "sage." When initialized, it also has five hidden columns corresponding to information in the Student table.
studentfrm	The detail form, with seven fields, which correspond to information in the Student table. Only the fields sgpa, scomment, and sadvisor are updatable. All other fields are display-only.
grad	A global structure, whose fields correspond in name and type to the columns of the Student database table, the studentfrm form and the studenttbl table field.

The program uses the "masterfrm" as the general-level master entry, in which data can only be retrieved and browsed, and the "studentfrm" as the detailed screen, in which specific student information can be updated.

The runtime user enters a name in the pname field and then selects the **Students** menu operation. The operation fills the table field "studenttbl" with detailed information of the students reporting to the named professor. This is done by the database cursor "studentcsr" in the procedure "Load_Students." The program assumes that each professor is associated with exactly one department.

The user can then browse the table field (in **read** mode), which displays only the names and ages of the students. More information about a specific student can be requested by selecting the **Zoom** menu operation. This operation displays the form "studentfrm" (in **update** mode). The fields of "studentfrm" are filled with values stored in the hidden columns of "studenttbl." The user can make changes to three fields ("sgpa," "scomment," and "sadvisor"). If validated, these changes will be written back to the database table (based on the unique student id), and to the table field's data set. This process can be repeated for different professor names.

```
{
| Procedure:  Prof_Student
| Purpose:    Main body of "Professor Student" Master-Detail application.
}

program Prof_Student( input, output );

exec sql include sqlca;

exec sql declare student table {Graduate student table}
    (sname      char(25),       {Name}
     sage       integer1,       {Age}
     sbdate     char(25),       {Birth date}
     sgpa       float4,         {Grade point average}
     sidno      integer,        {Unique student number}
     scomment   varchar(200),   {General comments}
     sadvisor   char(25));      {Advisor's name}

exec sql declare professor table {Professor table}
     (pname     char(25),       {Professor's name}
     pdept      char(10));      {Department}

exec sql begin declare section;
type
    Short_Short_Integer = [byte] -128..127;

    String1 = packed array[1..1] of Char;
    String10 = packed array[1..10] of Char;
    String25 = packed array[1..25] of Char;
    String100 = packed array[1..100] of Char;
    String200 = packed array[1..200] of Char;

    {Graduate student record maps to "student" database table }
    Student_Rec = record
        sname:    String25;
        sage:     Short_Short_Integer;
        sbdate:   String25;
        sgpa:     Real;
        sidno:    Integer;
```

```
        scomment: String200;
        sadvisor: String25;
    end;
var
    grad: Student_Rec;
    {Master and student compiled forms (imported objects)}
    masterfrm, studentfrm: [external] Integer;
exec sql end declare section;

{
| Procedure:    Load_Students
| Purpose:      Given an advisor name, load into the "studenttbl"
|               table field all the graduate students who report
|               to the professor with that name.
|               Columns "sname" and "sage" will be displayed, and
|               all other columns will be hidden.
| Parameters:   advisor - User specified professor name.
|                Uses the global student record "grad".
}

procedure Load_Students( var adv : String25 );
    label
        Load_End;
    exec sql begin declare section;
    var
        advisor : String25;
    exec sql end declare section;

    exec sql declare studentcsr cursor for
        select sname, sage, sbdate, sgpa,
            sidno, scomment, sadvisor
        from student
        where sadvisor = :advisor;

begin           {Load_Students}
    advisor := adv;

    {
    | Clear previous contents of table field. Load the table
    | field from the database table based on the advisor name.
    | Columns "sname" and "sage" will be displayed, and all
    | others will be hidden.
    }

    exec frs message 'Retrieving Student Information . . .';
    exec frs clear field studenttbl;
    exec frs redisplay; {Refresh for query}

    exec sql whenever sqlerror goto Load_End;
    exec sql whenever not found goto Load_End;

    exec sql open studentcsr;

    {
    | Before we start the loop, we know that the OPEN
    | was successful and that NOT FOUND was not set.
    }

    while (sqlca.sqlcode = 0) do
    begin
        exec sql fetch studentcsr into :grad;

    exec frs loadtable masterfrm studenttbl
        (sname = :grad.sname,
         sage = :grad.sage,
         sbdate = :grad.sbdate,
```

```
            sgpa = :grad.sgpa,
            sidno = :grad.sidno,
            scomment = :grad.scomment,
            sadvisor = :grad.sadvisor);
    end;

Load_End:       {Clean up on an error, and close cursors}
        exec sql whenever not found continue;
        exec sql whenever sqlerror continue;
        exec sql close studentcsr;
    end; {Load_Students}


    {
    | Function:    Student_Info_Changed
    | Purpose:     Allow the user to zoom into the details of a
    |              selected student. Some of the data can be
    |              updated by the user. If any updates were made,
    |              then reflect these back into the database table.
    |              The procedure returns TRUE if any changes were made.
    | Parameters:  None
    | Returns:     TRUE/FALSE - Changes were made to the database.
    |              Sets the global "grad" record with the new data.
    }

function Student_Info_Changed : Boolean;
    exec frs label;
    exec sql begin declare section;
    var
        changed: Integer; {Changes made to the form?}
        valid_advisor: Integer; {Is the advisor name valid?}
    exec sql end declare section;

begin           {Student_Info_Changed}
    {Local error handler just prints error and continues}
    exec sql whenever sqlerror call sqlprint;
    exec sql whenever not found continue;

    {Display the detailed student information}
    exec frs display studentfrm fill;
    exec frs initialize
        (sname = :grad.sname,
         sage = :grad.sage,
         sbdate = :grad.sbdate,
         sgpa = :grad.sgpa,
         sidno = :grad.sidno,
         scomment = :grad.scomment,
         sadvisor = :grad.sadvisor);

    exec frs activate menuitem 'Write';
    exec frs begin;

        {
        | If changes were made, then update the
        | database table. Only bother with the
        | fields that are not read-only.
        }
        exec frs inquire_frs form (:changed = change);

        if (changed = 1) then
        begin
            exec frs validate;
            exec frs message
                'Writing changes to database. . .';

            exec frs getform
```

```
                 (:grad.sgpa = sgpa,
                  :grad.scomment = scomment,
                  :grad.sadvisor = sadvisor);

            {Enforce integrity of professor name}
            valid_advisor := 0;
            exec sql select 1 into :valid_advisor
                from professor
                where pname = :grad.sadvisor;

            if (valid_advisor = 0) then
            begin
                exec frs message
                        'Not a valid advisor name';
                exec frs sleep 2;
                exec frs resume field sadvisor;
            end else
            begin
                exec sql update student set
                        sgpa = :grad.sgpa,
                        scomment = :grad.scomment,
                        sadvisor = :grad.sadvisor
                        where sidno = :grad.sidno;
            end;
        end;
        exec frs breakdisplay;
    exec frs end;       {"Write"}

    exec frs activate menuitem 'Quit';
    exec frs begin;
        {Quit without submitting changes }
        changed := 0;
        exec frs breakdisplay;
    exec frs end; {"Quit"}

    exec frs finalize;

    Student_Info_Changed := (changed = 1);
exec frs end; {Student_Info_Changed}

{
| Procedure:    Master
| Purpose:      Drive the application, by running "masterfrm" and
|               allowing the user to "zoom" into a selected student.
| Parameters:   None - Uses the global student "grad" record.
}

procedure Master;
    exec frs label;
    exec sql begin declare section;
    type
        {Professor record maps to "professor" database table }
         Prof_Rec = record
            pname: String25;
            pdept: String10;
         end;
    var
            prof: Prof_Rec;

            {Useful forms runtime information }
            lastrow,            {Lastrow in table field }
            istable: Integer;   {Is a table field? }

            {Utility buffers }
            msgbuf: String100;          {Message buffer }
            respbuf: String1;           {Response buffer }
```

```
            old_advisor: String25;       {Old advisor before ZOOM}
        exec sql end declare section;

    begin                                {Master}
        {
        | Initialize "studenttbl" with a data set in READ mode.
        | Declare hidden columns for all the extra fields that
        | the program will display when more information is
        | requested about a student. Columns "sname" and "sage"
        | are displayed. All other columns are hidden, to be
        | used in the student information form.
        }

        exec frs inittable masterfrm studenttbl read
            (sbdate = char(25),
             sgpa = float4,
             sidno = integer,
             scomment = char(200),
             sadvisor = char(20));

        {
        | Drive the application by running "masterfrm" and
        | allowing the user to "zoom" into a selected student.
        }
        exec frs display masterfrm update;

        exec frs initialize;
        exec frs begin;
            exec frs message 'Enter an Advisor name . . .';
            exec frs sleep 2;
        exec frs end;

        exec frs activate menuitem 'Students', field 'pname';
        exec frs begin;
            {Load the students of the specified professor }
            exec frs getform (:prof.pname = pname);

            {If no professor name is given, resume }
            if (prof.pname[1] = ' ') then
                exec frs resume field pname;

            {
            | Verify that the professor exists. If not print
            | print a message, and continue. Assume that
            | each professor has exactly one department.
            }

            exec sql whenever sqlerror call sqlprint;
            exec sql whenever not found continue;

            prof.pdept := ' ';
            exec sql select pdept
                into :prof.pdept
                from professor
                where pname = :prof.pname;

            {If no professor, report error}
            if (prof.pdept[1] = ' ') then
            begin
                msgbuf := 'No professor with name ''' +
                    prof.pname + ''' [return]';
                exec frs prompt noecho (:msgbuf, :respbuf);
                exec frs clear field all;
                exec frs resume field pname;
            end;
```

```
    {Fill the department field and load students }
    exec frs putform (pdept = :prof.pdept);
    Load_Students( prof.pname );

    exec frs resume field studenttbl;
exec frs end; {"Students" }

exec frs activate menuitem 'Zoom';
exec frs begin;
    {
    | Confirm that user is in "studenttbl" and that
    | the table field is not empty. Collect data from
    | the row and zoom for browsing and updating.
    }

    exec frs inquire_frs field masterfrm (:istable = table);
    if (istable = 0) then
    begin
        exec frs prompt noecho
            ('Select from the student table [return]',
                    :respbuf);
        exec frs resume field studenttbl;
    end;

    exec frs inquire_frs table masterfrm
            (:lastrow = lastrow);
    if (lastrow = 0) then
    begin
        exec frs prompt noecho
            ('There are no students [RETURN]',
                    :respbuf);
        exec frs resume field pname;
    end;

    {Collect all data on student into graduate record }
    exec frs getrow masterfrm studenttbl
        (:grad.sname = sname,
         :grad.sage = sage,
         :grad.sbdate = sbdate,
         :grad.sgpa = sgpa,
         :grad.sidno = sidno,
         :grad.scomment = scomment,
         :grad.sadvisor = sadvisor);

    {
    | Display "studentfrm", and if any changes were made,
    | make the updates to the local table field row.
    | Only make updates to the columns corresponding to
    | writable fields in "studentfrm." If the student
    | changed advisors, then delete the row from the
    | display.
    }

    old_advisor := grad.sadvisor;
    if (Student_Info_Changed) then
    begin
        if (old_advisor <> grad.sadvisor) then
        begin
            exec frs deleterow
                    masterfrm studenttbl;
        end else
        begin
            exec frs putrow masterfrm studenttbl
                    (sgpa = :grad.sgpa,
                     scomment = :grad.scomment,
                     sadvisor = :grad.sadvisor);
```

```
            end;
        end;
    exec frs end;               {"Zoom"}

    exec frs activate menuitem 'Exit';
    exec frs begin;
        exec frs breakdisplay;
    exec frs end;               {"Exit"}

    exec frs finalize;
  exec frs end; {Master}

begin                           {Prof_Student}
    {Start up Ingres and the Forms system }
    exec frs forms;

    exec sql whenever sqlerror stop;
    exec frs message 'Initializing Student Administrator . . .';
    exec sql connect personnel;

    exec frs addform :masterfrm;
    exec frs addform :studentfrm;

    Master;

    exec frs clear screen;
    exec frs endforms;
    exec sql disconnect;
end. {Prof_Student}
```

The SQL Terminal Monitor Application

This application executes SQL statements that are read in from the terminal. The application reads statements from input and writes results to output. Dynamic SQL is used to process and execute the statements.

When application starts, the user is prompted for the database name. The user is then prompted for an SQL statement. SQL comments and statement delimiters are not accepted. The SQL statement is processed using Dynamic SQL, and results and SQL errors are written to output. At the end of the results, an indicator of the number of rows affected is displayed. The loop is then continued and the user is prompted for another SQL statement. When end-of-file is typed in, the application rolls back any pending updates and disconnects from the database.

The user's SQL statement is prepared using **prepare** and **describe**. If the SQL statement is not a **select** statement, it is run using **execute** and the number of rows affected is printed. If the SQL statement *is* a **select** statement, a Dynamic SQL cursor is opened, and all the rows are fetched and printed. The routines that print the results do not try to tabulate the results. A row of column names is printed, followed by each row of the results.

Keyboard interrupts are not handled. Fatal errors, such as allocation errors, and boundary condition violations are handled by rolling back pending updates and disconnecting from the database session.

```
program SQL_Monitor (input, output);
{ Declare the SQLCA and the SQLDA records }
exec sql include sqlca;
exec sql include sqlda;

exec sql begin declare section;
var
    dbname: varying [50] of Char;         { Database name }
exec sql end declare section;

var
    sqlda: IIsqlda;                       { Global SQLDA record }

exec sql declare stmt statement;          { Dynamic SQL statement }
exec sql declare csr cursor for stmt;     { Cursor for dynamic statement}

{
|Constants and types needed to declare global storage
|for SELECT results
}

const
    { Length of large string pool from which sub-strings
    | will be allocated
    }
    max_string = 3000;

type
    { Different numeric types for result variables }
    Numerics = record
            n_int: Integer;              { 4-byte integers }
            n_flt: Double;               { 8-byte floating points }
            n_ind: Indicator;    { 2-byte null indicators }
        end;

    { Large string pool from which to allocate sub-strings }
    Strings = record
            s_len: Integer; { Length used, and data }
            s_data: array [1..MAX_STRING] of Char;
        end;

var
    {
    | Global result storage area - set up by Print_Header, filled when
    | executing the FETCH statement, and displayed by Print_Row.
    | Record is declared volatile so that the IADDRESS and ADDRESS
    | functions can correctly point SQLDATA and SQLIND at the various
    | components.
    }
    res:        [volatile] record
                        nums: array [1..IISQ_MAX_COLS] of Numerics;
                        str: Strings;
                end;

{ Forward defined procedures and functions }

{ Main body of monitor }
procedure Run_Monitor; forward;

{ Execute dynamic SELECT statements }
function Execute_Select: Integer; forward;

{ Print the column headers for a dynamic SELECT }
function Print_Header: Boolean; forward;

{ Print a result row for a dynamic SELECT }
```

```
procedure Print_Row; forward;

{ Print an error message }
procedure Print_Error; forward;

{ Read a statement from input }
function Read_Stmt(stmt_num: Integer;
        var stmt_buf: varying[len] of char): Boolean; forward;

    {
    | Procedure: Run_Monitor
    | Purpose:   Run the SQL monitor. Initialize the global
    |            SQLDA with the number of SQLVAR elements.
    |            Loop while prompting the user for input; if
    |            end-of-file is detected then return to the main program.
    |
    |            If the statement is not a SELECT statement
    |            then EXECUTE it, otherwise open a cursor and
    |            process a dynamic SELECT statement (using Execute_Select).
    }

    procedure Run_Monitor;

        label
            Exec_Error;                         { SQL error in statement }

        exec sql begin declare section;

        var
            stmt_buf: varying[1000] of Char;    { SQL statement input buffer }
            stmt_num: Integer;                  { SQL statement number }
            rows: Integer;                      { # of rows affected }
        exec sql end declare section;

        var
            reading: Boolean;                   { While reading statements }

    begin                               { Run_Monitor }

        sqlda.sqln := IISQ_MAX_COLS; { Initialize the SQLDA }

        { Now we are set for input }
        stmt_num := 0;
        reading  := TRUE;

        while (reading) do begin

        stmt_num := stmt_num + 1;

        {
        | Prompt and read the next statement. If Read_Stmt
        | returns FALSE then end-of-file was detected.
        }
        reading := Read_Stmt(stmt_num, stmt_buf);

        if (reading) then begin

            { Handle database errors }
            exec sql whenever sqlerror goto Exec_Error;

            {
            | Prepare and describe the statement. If the statement
            | is not a SELECT then EXECUTE it, otherwise inspect the
            | contents of the SQLDA and call Execute_Select.
            }
            exec sql prepare stmt from :stmt_buf;
```

```
            exec sql describe stmt into :sqlda;

            { If SQLD = 0 then this is not a SELECT }
            if (sqlda.sqld = 0) then begin

                exec sql execute stmt;
                rows := sqlca.sqlerrd[3];

            end else begin                      { This is a SELECT }

                { Are there enough result variables }
                if (sqlda.sqld < sqlda.sqln) then begin
                        rows := Execute_Select;
                end else begin                  { Too few result variables }
                        writeln('SQL Error: SQLDA requires ',
                                sqlda.sqld:1,
                                ' variables, but has only ',
                                sqlda.sqln:1, '.');
                        rows := 0;
                end;                    { If enough result variables }

            end;                        { If SELECT or not }

            { Display number of rows processed }
            writeln('[', rows:1, ' row(s)]');

        Exec_Error:
            exec sql whenever sqlerror continue;
            { If we have an error then display the error message }
            if (sqlca.sqlcode < 0) then
                Print_Error;
        end;                            { If reading a statement }

    end;                                { While reading statements }

end;                                    { Run_Monitor }

{
| Function:   Execute_Select
| Purpose:    Run a dynamic SELECT statement. The SQLDA has
|             already been described, so print the column header
|             (names), open a cursor, and retrieve and print the
|             results. Accumulate the number or rows processed.
| Returns:    Number of rows processed.
}

function Execute_Select;
            { : Integer; }

    label
        Select_Error;                   { SQL error in statement }
    var
        rows:   Integer;                { Counter of rows fetched }

begin                                   { Execute_Select }

    Execute_Select := 0;

    {
    | Print result column names, set up the result types and
    | variables.Print_Header returns FALSE if the dynamic
    |   set-up failed.
    }
    if (Print_Header) then begin

        exec sql whenever sqlerror goto Select_Error;
```

```
        exec sql open csr for readonly;  { Open the dynamic cursor }

        { Fetch and print each row }
        rows := 0;
        while (sqlca.sqlcode = 0) do begin

            exec sql fetch csr using descriptor :sqlda;
            if (sqlca.sqlcode = 0) then begin
                rows := rows + 1;        { Count the rows }
                Print_Row;
            end;

        end;                             { While there are more rows }

    Select_Error:
        {
        |If we got here because of an error then print
        |the error message
        }
        if (sqlca.sqlcode < 0) then
                Print_Error;
        exec sql whenever sqlerror continue;
        exec sql close csr;

        Execute_Select := rows;
    end;                                 { If Print_Header }
end;                                     { Execute_Select }

{
| Function:   Print_Header
| Purpose:    A statement has just been described so set up the
|             SQLDA for result processing. Print all the column
|             names and allocate (point at) result variables for
|             retrieving data. The result variables are chosen
|             out of a pool of variables (integers, floats and
|             a large character string space). The SQLDATA and
|             SQLIND fields are pointed at the addresses of the
|             result variables.
| Returns:    TRUE if successfully set up the SQLDA for result
|             variables, FALSE if an error occurred.
}

function Print_Header;
              { : Boolean; }

var
        col:    Integer;         { Index into SQLVAR }
        col_err: Boolean;        { Error processing column }
        col_null: Boolean;       { Null indicator required }
        cur_len: Integer;        { Current string length }

begin                            { Print_Header }

    res.str.s_len := 1;          { No strings used yet }
    col := 1;
    col_err := FALSE;

    while (col <= sqlda.sqld) and (not col_err) do begin

        with sqlda.sqlvar[col] do begin

            {
            | For each column display the number and name, ie:
            | [1] sal [2] name [3] age
            }
```

```
write('[', col:1, '] ', sqlname);
if (col < sqlda.sqld) then
    write(' ');     { Separator space }

{
| Determine the data type of the column and to
| where SQLDATA and SQLIND must point in order to
| retrieve data-compatible results. Use the global
| result storage area to allocate the result variables.
|
| Collapse all different types into Integers, Floats
| or Characters.
}

if (sqltype < 0) then       { Null indicator handled later }
    col_null := TRUE
else
    col_null := FALSE;

case (abs(sqltype)) of
    IISQ_INT_TYPE:          { Integers }
            begin
                    sqltype := IISQ_INT_TYPE;
                    sqllen := 4;
                    sqldata := iaddress(res.nums[col].n_int);
            end;

    IISQ_MNY_TYPE,          { Floating points }
    IISQ_FLT_TYPE:
            begin
                    sqltype := IISQ_FLT_TYPE;
                    sqllen := 8;
                    sqldata := iaddress(res.nums[col].n_flt);
            end;

    IISQ_DTE_TYPE, { Characters }
    IISQ_CHA_TYPE,
    IISQ_VCH_TYPE:
            begin
                    { First determine required length }
                    if (abs(sqltype) = IISQ_DTE_TYPE) then
                            cur_len := IISQ_DTE_LEN
                    else
                            cur_len := sqllen;

                    { Enough room in large string buffer ? }
                    if ((res.str.s_len + cur_len)
                                    <= MAX_STRING) then
                        begin
                        { Point at a sub-string in buffer }
                        sqltype := IISQ_CHA_TYPE;
                        sqllen := cur_len;
                        sqldata :=
                          iaddress(res.str.s_data[res.str.s_len]);
                        res.str.s_len := res.str.s_len + cur_len;
                    end else begin
                        writeln;
                        writeln('SQL Error: Character result
                             data','overflow.');
                        col_err := TRUE;
            end;                    { If room in string }
    end;
end;                                { Case of data types }

{ Assign pointers to null indicators and toggle type }
if (col_null) then begin
```

```
                sqltype := -sqltype;
                sqlind  := iaddress(res.nums[col].n_ind);
            end else begin
                sqlind := 0;
            end;

        end;                            { With current column }

        col := col + 1;

    end;                                { While processing columns }

    writeln;                            { Print separating line }
    writeln('---------------------------------------');

    Print_Header := not col_err;
end; { Print_Header }

{
| Procedure:  Print_Row
| Purpose:    For each element inside the SQLDA, print the value.
|             Print its column number too in order to identify it
|             with a column name printed earlier in Print_Header.
|             If the value is NULL print "N/A".
}

procedure Print_Row;

    var
        col:    Integer;        { Index into SQLVAR }
        ch:     Integer;        { Index into sub-strings }

begin                           { Print_Row }

    res.str.s_len := 1; { Reset string counter }
    col := 1;
    for col := 1 to sqlda.sqld do begin

        with sqlda.sqlvar[col] do begin

            { For each column display the number and value }
            write('[', col:1, '] ');

            if (sqltype < 0) and (res.nums[col].n_ind = -1) then begin

                write('N/A');

            end else begin

                {
                | Using the base type set up in Print_Header
                | determine how to print the results. All types
                | are printed using default formats.
                }

                case (abs(sqltype)) of
                        IISQ_INT_TYPE:
                                write(res.nums[col].n_int:1);

                IISQ_FLT_TYPE:
                        write(res.nums[col].n_flt);

                IISQ_CHA_TYPE:
                        begin
                                for ch := 0 to sqllen - 1 do begin
                                        write(res.str.s_data
```

```
                                        [res.str.s_len + ch]);
                                end;
                                res.str.s_len := res.str.s_len + sqllen;
                        end;
                    end;                        { Case of data types }

                end;                            { If null or not }

            end;                                { With current column }

            if (col < sqlda.sqld) then          { Add trailing space }
                write(' ');

        end;                                    { While processing columns }
        writeln; { Print end of line }

end; { Print_Row }

{
| Procedure:  Print_Error
| Purpose:    SQLCA error detected. Retrieve the error message and print it.
}
procedure Print_Error;

        exec sql begin declare section;
        var
            error_buf: varying[400] of Char; { SQL error text retrieval }
        exec sql end declare section;

begin

        exec sql inquire_sql (:error_buf = errortext);
        writeln('SQL Error:');
        writeln(error_buf);

end;                                    { Print_Error }

{
| Function:   Read_Stmt
| Purpose:    Reads a statement from standard input. This routine
|             prompts the user for input (using a statement number)
|             and returns the response. The routine can be extended
|             to scan for tokens that delimit the statement, such
|             as semicolons and quotes, in order to allow the
|             statement to be continued over multiple lines.
| Parameters:
|             stmt_num - Statement number for prompt.
|             stmt_buf - Buffer to fill for input.
| Returns:
|             TRUE if a statement was read,
|             FALSE if end-of-file typed.
}

function Read_Stmt;
            { (stmt_num:Integer;
               var stmt_buf: Varying of Char) : Boolean; }

begin

        write(stmt_num:1, '> ');                { Prompt for SQL statement }
        if (not eof) then begin
            readln(stmt_buf);
            Read_Stmt := TRUE;
        end else begin
            stmt_buf := '';
            Read_Stmt := FALSE;
```

```
        end;

end;                                            { Read_Stmt }

{
| Program: SQL_Monitor Main
| Purpose: Main entry of SQL Monitor application. Prompt for database
|          name and connect to the database. Run the monitor and
|          disconnect from the database. Before disconnecting roll
|          back any pending updates.
}

begin                                           { Main Program }

        open(output, record_length :=
                MAX_STRING);                    { For large result lines }

        write('SQL Database: ');                { Prompt for database name }
        readln(dbname);

        writeln(' -- SQL Terminal Monitor --');

        { Treat connection errors as fatal errors }
        exec sql whenever sqlerror stop;
        exec sql connect :dbname;

        Run_Monitor;

        exec sql whenever sqlerror continue;

        writeln('SQL: Exiting monitor program.');
        exec sql rollback;
        exec sql disconnect;

end. { Main Program }
```

A Dynamic SQL/Forms Database Browser

This program lets the user browse data from and insert data into any table in any database, using a dynamically defined form. The program uses Dynamic SQL and Dynamic FRS statements to process the interactive data. You should already have used VIFRED to create a Default Form based on the database table that you want to browse. VIFRED will build a form with fields that have the same names and data types as the columns of the specified database table.

When run, the program prompts the user for the name of the database, the table and the form. The form is profiled using the **describe form** statement, and the field name, data type and length information is processed. From this information the program fills in the SQLDA data and null indicator areas, and builds two Dynamic SQL statement strings to **select** data from and insert(b) data into the database.

The **Browse** menu item retrieves the data from the database using an SQL cursor associated with the dynamic **select** statement, and displays that data using the dynamic **putform** statement. A **submenu** allows the user to continue with the next row or return to the main menu. The **Insert** menu item retrieves the data from the form using the dynamic **getform** statement, and adds the data to the database table using a prepared **insert** statement. The **Save** menu item commits the user's changes and, because prepared statements are discarded, reprepares the **select** and **insert** statements. When the **Quit** menu item is selected, all pending changes are rolled back and the program is terminated.

```
program Dynamic_FRS;
    exec sql labeL exit_program;         { Exit on error }

    exec sql include sqlca;              { Declare the SQLCA and }
    exec sql include sqlda;              { and the SQLDA records }

    var
        sqlda: IIsqlda;                  { Global SQLDA record }

    const
        MAX_NAME = 50;                   { Input name size }
        MAX_STRING = 3000;               { Large string buffer size }
        MAX_STMT = 1000;                 { SQL statement string size }

    {
    | Result storage pool for Dynamic SQL and FRS statements.
    | This result pool consists of arrays of 4-byte integers,
    | 8-byte floating-points, 2-byte indicators, and a large
    | string buffer from which sub-strings will be allocated.
    | Each SQLDA SQLVAR sets its SQLDATA and SQLIND address pointers
    | to variables from this pool.
    |
    | Note that the arrays are declared as volatile so that the
    | IADDRESS and ADDRESS functions can correctly point SQLDATA
    | and SQLIND at the various elements.
    }
    var
            integers:     [volatile] array[1..IISQ_MAX_COLS] of Integer;
            floats:       [volatile] array[1..IISQ_MAX_COLS] of Double;
            indicators:   [volatile] array[1..IISQ_MAX_COLS] of Indicator;
            characters:   [volatile] array[1..MAX_STRING] of Char;

    exec sql begin declare section;
        type
            Statement_Buf = varying[MAX_STMT]
                of Char;                                 { Statement string }
            Input_Name = varying[MAX_NAME] of Char;      { Input name }
        var
            dbname: Input_Name;                  { Database name }
            formname: Input_Name;                { Form name }
            tabname: Input_Name;                 { Database table name }
            sel_buf: Statement_Buf;              { Prepared SELECT statement }
            ins_buf: Statement_Buf;              { Prepared INSERT statement }
            err: Integer;                        { Error status }
            ret: Char;                           { Prompt error buffer }
    exec sql end declare section;

    {
    | Function: Describe_Form
    | Purpose:  Profile the specified form for name and data
    |           type information. Using the DESCRIBE FORM statement,
    |           the SQLDA is loaded with field information from the
```

```
|           form. This procedure processes this information to
|           allocate result storage, point at storage for
|           dynamic FRS data retrieval and assignment, and build
|           SQL statements strings for subsequent dynamic
|           SELECT and INSERT statements. For example, assume the
|           form (and table) 'emp' has the following fields:
|
|           Field Name      Type            Nullable?
|           ----------      ----            ---------
|           name            char(10)        No
|           age             integer4        Yes
|           salary          money           Yes
|
|       Based on 'emp', this procedure will construct the
|       SQLDA. The procedure allocates variables from a
|       result variable pool (integers, floats and a large
|       character string space).
|       The SQLDATA and SQLIND fields are pointed at the
|       addresses of the result variables in the pool. The
|       following SQLDA is built:
|
|           sqlvar[1]
|           sqltype = IISQ_CHA_TYPE
|           sqllen = 10
|           sqldata = pointer into characters array
|           sqlind = null
|           sqlname = 'name'
|           sqlvar[2]
|           sqltype = -IISQ_INT_TYPE
|           sqllen = 4
|           sqldata = address of integers[2]
|           sqlind = address of indicators[2]
|           sqlname = 'age'
|           sqlvar[3]
|           sqltype = -IISQ_FLT_TYPE
|           sqllen = 8
|           sqldata = address of floats[3]
|           sqlind = address of indicators[3]
|           sqlname = 'salary'
|
|       This procedure also builds two dynamic SQL statements
|       strings. Note that the procedure should be extended to
|       verify that the statement strings do fit into the
|       statement buffers (this was not done in this example).
|       The above example would construct the following statement
|       strings:
|
|           'SELECT name, age, salary FROM emp ORDER BY name'
|           'INSERT INTO emp (name, age, salary) VALUES (?, ?, ?)'
|
| Parameters:
|           formname - Name of form to profile.
|           tabname - Name of database table.
|           sel_buf - Buffer to hold SELECT statement string.
|           ins_buf - Buffer to hold INSERT statement string.
| Returns:
|           TRUE/FALSE - Success/failure - will fail on error
|           or upon finding a table field.
}
function Describe_Form (formname, tabname: Input_Name;
        var sel_buf, ins_buf: Statement_Buf): Boolean;

var
    names:  Statement_Buf;          { Names for SQL statements }
    marks:  Statement_Buf;          { Place holders for INSERT }
    col:  Integer;                  { Index into SQLVAR }
```

```
    nullable: Boolean;              { Is nullable (SQLTYPE 0) }
    char_cnt: Integer;              { Total string length }
    char_cur: Integer;              { Current string length }
    described:Boolean;              { Return value }

begin                               { Describe_Form }

    {
    | Initialize the SQLDA and DESCRIBE the form. If we
    |   cannot fully describe the form (our SQLDA is too small)
    |   then report an error and return.
    }
    sqlda.sqln := IISQ_MAX_COLS;
    described := TRUE;

    exec frs describe form :formname all into :sqlda;
    exec frs inquire_frs frs (:err = ERRORNO);
    if (err > 0) then begin
        described := FALSE;  { Error already displayed }
    end else if (sqlda.sqld > sqlda.sqln) then begin
        exec frs prompt noecho ('SQLDA is too small for
                    form :', :ret);
        described := FALSE;
    end else if (sqlda.sqld = 0) then begin
        exec frs prompt noecho
                    ('There are no fields in the form :', :ret);
        described := FALSE;
    end;

    {
    | For each field determine the size and type of the
    | result data area. This data area will be allocated out
    | of the result variable pool (integers, floats and
    | characters) and will be pointed at by SQLDATA and SQLIND.
    |
    | If a table field type is returned then issue an error.
    |
    | Also, for each field add the field name to the 'names'
    | buffer and the SQL place holders '?' to the 'marks'
    | buffer, which will be used to build the final SELECT
    | and INSERT statements.
    }
    char_cnt := 1;                  { No strings used yet }
    col := 1;

    while (col <= sqlda.sqld) and (described) do begin

        with sqlda.sqlvar[col] do begin

            {
            | Collapse all different types into Integers, Floats
            | or Characters.
            }
            if (sqltype < 0) then { Null indicator handled later }
                    nullable := TRUE
            else
                    nullable := FALSE;
            case (abs(sqltype)) of
                    IISQ_INT_TYPE:          { Integers }
                            begin
                                    sqltype := IISQ_INT_TYPE;
                                    sqllen := 4;
                                    sqldata := iaddress(integers[col]);
                            end;

                    IISQ_MNY_TYPE,          { Floating-points }
```

```
        IISQ_FLT_TYPE:
                begin
                        sqltype := IISQ_FLT_TYPE;
                        sqllen := 8;
                        sqldata := iaddress(floats[col]);
                end;

        IISQ_DTE_TYPE,          { Characters }
        IISQ_CHA_TYPE,
        IISQ_VCH_TYPE:
                begin
                        { First determine required length }
                        if (abs(sqltype) = IISQ_DTE_TYPE) then
                                char_cur := IISQ_DTE_LEN
                        else
                                char_cur := sqllen;

                        { Enough room in large string buffer ? }
                        if ((char_cnt + char_cur) > MAX_STRING)
                                then begin
                                exec frs prompt noecho
                                        ('Character pool buffer
                                        overflow :', :ret);
                                        described := FALSE;
                        end else begin
                             { Point at a sub-string in buffer}
                                sqltype := IISQ_CHA_TYPE;
                                sqllen := char_cur;
                                sqldata :=iaddress
                                (characters[char_cnt]);
                                char_cnt := char_cnt + char_cur;
                        end; { If room in string }
                end;

        IISQ_TBL_TYPE:
                begin
                        exec frs prompt noecho
                        ('Table field found in form :', :ret);
                                described := FALSE;
                end;

        otherwise
                begin
                        exec frs prompt noecho
                        ('Invalid field type :', :ret);
                                described := FALSE;
                end;

end;                    { Case of data types }

{ Assign pointers to null indicators and toggle type }
if (nullable) then begin
        sqltype := -sqltype;
        sqlind := iaddress(indicators[col]);
end else begin
        sqlind := 0;
end;

{
| Store field names and place holders (separated by commas)
| for the SQL statements.
}
if (col = 1) then begin
        names := sqlname;
        marks := '?';
end else begin
```

```
                        names := names + ',' + sqlname;
                        marks := marks + ',?';
                end;

            end;                                    { With current column }
            col := col + 1;

end;                                                { While processing columns }
{
| Create final SELECT and INSERT statements. For the SELECT
| statement ORDER BY the first field.
}
if (described) then begin
        sel_buf := 'SELECT ' + names + ' FROM ' + tabname
                 + ' ORDER BY ' + sqlda.sqlvar[1].sqlname;
        ins_buf := 'INSERT INTO ' + tabname + ' (' + names
                + ') VALUES (' + marks + ')';
end;

Describe_Form := described;

end;                                                { Describe_Form }

{
| Program: Dynamic_FRS Main
| Purpose: Main body of Dynamic SQL forms application. Prompt for
|          database, form and table name. Call Describe_Form
|          to obtain a profile of the form and set up the SQL
|          statements. Then allow the user to interactively browse
|          the database table and append new data.
}

begin { Dynamic_FRS Main }

    exec sql declare sel_stmt
             statement;                     { Dynamic SQL SELECT statement }
    exec sql declare
             ins_stmt statement;            { Dynamic SQL INSERT statement }
    exec sql declare csr cursor
             for sel_stmt;                  { Cursor for SELECT statement }
    exec frs forms;

    { Prompt for database name - will abort on errors }
    exec sql whenever sqlerror stop;
    exec frs prompt ('Database name: ', :dbname);
    exec sql connect :dbname;

    exec sql whenever sqlerror call sqlprint;

    {
    | Prompt for table name - later a Dynamic sql select statement
    | will be built from it.
    }
    exec frs prompt ('Table name: ', :tabname);

    {
    | Prompt for form name. Check forms errors reported
    | through inquire_frs.
    }
    exec frs prompt ('Form name: ', :formname);
    exec frs message 'Loading form ...';
    exec frs forminit :formname;
    exec frs inquire_frs frs (:err = ERRORNO);
    if (err > 0) then begin
        exec frs message 'Could not load form. Exiting.';
        exec frs endforms;
```

```
        exec sql disconnect;
        goto exit_program;
    end;

    { Commit any work done so far - access of forms catalogs }
    exec sql commit;

    { Describe the form and build the SQL statement strings }
    if (not Describe_Form(formname, tabname, sel_buf, ins_buf))
        then begin
            exec frs message 'Could not describe form. Exiting.';
            exec frs endforms;
            exec sql disconnect;
            goto exit_program;
        end;

        {
        | PREPARE the SELECT and INSERT statements that correspond to the
        | menu items Browse and Insert. If the Save menu item is chosen
        | the statements are reprepared.
        }
        exec sql prepare sel_stmt from :sel_buf;
        err := sqlca.sqlcode;
        exec sql prepare ins_stmt from :ins_buf;
        if ((err < 0) or (sqlca.sqlcode < 0)) then begin
            exec frs message 'Could not prepare SQL statements. Exiting.';
            exec frs endforms;
            exec sql disconnect;
            goto exit_program;
        end;

        {
        | Display the form and interact with user, allowing browsing
        | and the inserting of new data.
        }
        exec frs display :formname FILL;
        exec frs initialize;
        exec frs activate menuitem 'Browse';
        exec frs begin;
            {
            | Retrieve data and display the first row on the form,
            | allowing the user to browse through successive rows.
            | If data types from the database table are not consistent
            | with data descriptions obtained from the form, a
            | retrieval error will occur. Inform the user of this or
            | other errors.
            |
            | Note that the data will return sorted by the first
            | field that was described, as the SELECT statement,
            | sel_stmt, included an ORDER BY clause.
            }
            exec sql open csr;

            { Fetch and display each row }
            while (sqlca.sqlcode = 0) do begin

            exec sql fetch csr using descriptor :sqlda;
            if (sqlca.sqlcode <> 0) then begin
                exec sql close csr;
                exec frs prompt noecho ('No more rows :', :ret);
                exec frs clear field all;
                exec frs resume;
            end;

            exec frs putform :formname using descriptor :sqlda;
            exec frs inquire_frs frs (:err = errorno);
```

```
            if (err > 0) then begin
                exec sql close csr;
                exec frs resume;
            end;

            { Display data before prompting user with submenu }
            exec frs redisplay;

            exec frs submenu;
            exec frs activate menuitem 'Next', FRSKEY4;
            exec frs begin;
                 { Continue with cursor loop }
                 exec frs message 'Next row ...';
                 exec frs clear field all;
            exec frs end;
            exec frs activate menuitem 'End', FRSKEY3;
            exec frs begin;
                 exec sql close csr;
                 exec frs clear field all;
                 exec frs resume;
            exec frs end;

        end;                             { While there are more rows }
exec frs end;

exec frs activate menuitem 'Insert';
exec frs begin;
    exec frs getform :formname using descriptor :sqlda;
    exec frs inquire_frs frs (:err = ERRORNO);
    if (err > 0) then begin
        exec frs clear field all;
        exec frs resume;
    end;
    exec sql execute ins_stmt using descriptor :sqlda;
    if ((sqlca.sqlcode < 0) or (sqlca.sqlerrd[3] = 0)) then begin
        exec frs prompt noecho ('No rows inserted :', :ret);
    end else begin
        exec frs prompt noecho ('One row inserted :', :ret);
    end;
exec frs end;

exec frs activate menuitem 'Save';
exec frs begin;
    {
    | COMMIT any changes and then re-PREPARE the SELECT and INSERT
    | statements as the COMMIT statements discards them.
    }
    exec sql commit;
    exec sql prepare sel_stmt from :sel_buf;
    err := sqlca.sqlcode;
    exec sql prepare ins_stmt from :ins_buf;
    if ((err < 0) or (sqlca.sqlcode < 0)) then begin
        exec frs prompt
             noecho ('Could not reprepare SQL statements :',:ret);
        exec frs breakdisplay;
    end;
exec frs end;

exec frs activate menuitem 'Clear';
exec frs begin;
    exec frs clear field all;
exec frs end;

exec frs activate menuitem 'Quit', FRSKEY2;
exec frs begin;
    exec sql rollback;
```

```
    exec frs breakdisplay;
exec frs end;
exec frs finalize;

exec frs endforms;
exec sql disconnect;

exit_program:;

exec sql end. { Dynamic_FRS Main }
```

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

T

U

V

W