InterBase
DSQL Programmer's Guide

Disclaimer

Borland International, Inc. (henceforth, Borland) reserves the right to make changes in specifications and other information contained in this publication without prior notice. The reader should, in all cases, consult Borland to determine whether or not any such changes have been made.

The terms and conditions governing the licensing of InterBase software consist solely of those set forth in the written contracts between Borland and its customers. No representation or other affirmation of fact contained in this publication including, but not limited to, statements regarding capacity, response-time performance, suitability for use, or performance of products described herein shall be deemed to be a warranty by Borland for any purpose, or give rise to any liability by Borland whatsoever.

In no event shall Borland be liable for any incidental, indirect, special, or consequential damages whatsoever (including but not limited to lost profits) arising out of or relating to this publication or the information contained in it, even if Borland has been advised, knew, or should have known of the possibility of such damages.

The software programs described in this document are confidential information and proprietary products of Borland.

Restricted Rights Legend. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subdivision (b) (3) (ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013.

© **Copyright 1993** by Borland International, Inc. All Rights Reserved. InterBase, GDML, and Pictor are trademarks of Borland International, Inc. All other trademarks are the property of their respective owners.

Corporate Headquarters: Borland International Inc., 100 Borland Way, P. O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-5300. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom.

Software Version: V3.0

Current Printing: October 1993
Documentation Version: v3.0.1

Reprint note

This documentation is a reprint of InterBase V3.0 documentation. It contains most of the information from *InterBase Previous Versions Documentation Corrections* and *InterBase Version* 3.2 *Documentation Corrections* and a new index. For information on features added since InterBase Version V3.0, consult the appropriate release notes.

Table Of Contents

Preface

Who Should Read this Book ix
Using this Book x
Text Conventions xi
Syntax Conventions xii
InterBase Documentation xiii

1 Introduction to Dynamic SQL

Overview 1-1
DSQL Statements 1-2
The SQL Descriptor Area 1-4
DSQL Considerations 1-8
For More Information 1-10

2 Processing DSQL Statements

Overview 2-1
Processing Non-Select Statements with No Associated Parameters 2-3
- Steps for C Programs 2-3
- C Program Example 2-3
- Steps for Pascal Programs 2-4
- Pascal Program Example 2-5

Processing Non-Select Statements with Associated Parameters 2-6
- Steps for C Programs 2-6
- C Program Example 2-7
- Steps for Pascal Programs 2-9
- Pascal Progam Example 2-10

Processing Select Statements with No Associated Parameters 2-13

Steps for C Programs. 2-13
C Program Example 2-14
Steps for Pascal Programs 2-16
Pascal Program Example 2-18
Processing Select Statements with Associated Parameters 2-21
Steps for C Programs. 2-21
C Program Example 2-23
Steps for Pascal Programs 2-25
Pascal Program Example 2-27
Processing Unknown Statements 2-30
Steps for C Programs. 2-30
C Program Example 2-33
Steps for Pascal Programs 2-42
Pascal Program Example 2-44
Checking for Null Values. 2-53
Steps for C Programs. 2-53
C Program Example 2-54
Steps for Pascal programs. 2-57
Pascal Example 2-58
For More Information 2-61

3 Accessing Special Field Types

Overview. 3-1
Accessing Blob Fields. 3-2
Steps for C Programs. 3-2
C Program Example 3-2
Steps for Pascal Programs 3-4
Pascal Program Example 3-4
Accessing Date Fields 3-6
Casting a Date Field 3-6
Using Gds Date Routines 3-10
For More Information 3-16

4 Setting up an SQLDA

Overview 4-1
Setting up an SQLDA in C 4-2
Setting up an SQLDA in Pascal 4-4
Setting up an SQLDA in PL/1 4-5
Setting up an SQLDA in Ada 4-6
Setting up an SQLDA in BASIC 4-10
Setting up an SQLDA in COBOL 4-11
Setting up an SQLDA in FORTRAN 4-12
For More Information 4-13

Preface

This manual contains information on Dynamic SQL, a capability for letting programs accept or generate SQL statements at runtime.

Who Should Read this Book

You should read this book if you are an applications programmer who uses an InterBase database. The book assumes programming knowledge, but does not assume specific knowledge of InterBase. This book is a companion to the *Programmer's Guide* and assumes you have already read that book.

Using this Book

This book contains the following chapters:

Chapter 1	Presents an overview of dynamic SQL (DSQL), describes dynamic SQL statements, and describes the SQL descriptor area (SQLDA).
Chapter 2	Describes how to process various types of DSQL statements.
Chapter 3	Describes how to access blob fields and date fields when using DSQL.
Chapter 4	Describes how to set up an SQLDA in C, Pascal, Pl/1, Ada, COBOL, BASIC, and FORTRAN programs.

Text Conventions

This book uses the following text conventions.

boldface

Indicates a command, option, statement, or utility. For example:

- Use the **commit** command to save your changes.
- Use the **sort** option to specify record return order.
- The **case_menu** statement displays a menu in the forms window.
- Use **gdef** to extract a data definition.

italic

Indicates chapter and manual titles; identifies filenames and pathnames. Also used for emphasis, or to introduce new terms. For example:

- See the introduction to SQL in the *Programmer's Guide*.
- /usr/interbase/lock_header
- Subscripts in RSE references *must* be closed by parentheses and separated by commas.
- C permits only *zero-based* array subscript references.

`fixed width font`

Indicates user-supplied values and example code:

- $run sys$system:iscinstall
- add field population_1950 long

UPPER CASE

Indicates relation names and field names:

- Secure the RDB$SECURITY_CLASSES system relation.
- Define a missing value of *X* for the LATITUDE_COMPASS field.

Syntax Conventions

This book uses the following syntax conventions.

{braces}

Indicates an alternative item:

- `option::= {`**`vertical`**`|`**`horizontal`**`|`**`trans-parent`**`}`

[brackets]

Indicates an optional item:

- `dbfield-expression[`**`not`**`]`**`missing`**

`fixed width font`

Indicates user-supplied values and example code:

- `$run sys$system:iscinstall`
- `add field population_1950 long`

commalist

Indicates that the preceding word can be repeated to create an expression of one or more words, with each word pair separated by one comma and one or more spaces.

For example,

`field_def-commalist`

resolves to:

field_def[,field_def[,field_def]...]

italics

Indicates a syntax variable:

`create_blob` *`blob-variable`* **`in`** *`dbfield-expression`*

|

Separates items in a list of choices. Only one such item in a list may be chosen at a time.

⇓

Indicates that parts of a program or statement have been omitted.

InterBase Documentation

The InterBase Version 3.0 documentation set contains the following books:

Getting Started with InterBase (INT0032WW2179A) provides an overview of InterBase components and interfaces.

Database Operations (INT0032WW2178D) describes how to use InterBase utilities to maintain databases.

Data Definition Guide (INT0032WW2178F) describes how to create and modify InterBase databases.

DDL Reference (INT0032WW2178E) describes the function and syntax for each of the data definition language clauses and statements. It also lists the standard error messages for **gdef**.

DSQL Programmer's Guide (INT0032WW2179C) describes how to program with DSQL, a capability for accepting or generating SQL statements at runtime.

Forms Guide (INT0032WW2178A) describes how to create forms using the InterBase forms editor, **fred,** and how to use forms in **qli** and GDML applications.

Programmer's Guide (INT0032WW2178I) describes how to program with GDML, a relational data manipulation language, and SQL, an industry standard language.

Programmer's Reference (INT0032WW2178H) describes the function and syntax for each of the GDML and InterBase supported SQL clauses and statements. It also lists the standard error messages for **gpre**.

Qli Guide (INT0032WW2178C) describes the use of **qli**, the InterBase query language interpreter that allows you to read to and write from the database using interactive GDML or SQL statements.

Qli Reference (INT0032WW2178B) describes the function and syntax for each of the data definition, GDML, and SQL clauses and statements that you can use in **qli.**

Sample Programs (INT0032WW2178G) contains sample programs that show the use of InterBase features.

Master Index (INT0032WW2179B) contains index entries for the entire InterBase Version 3.0 documentation set.

Chapter 1 Introduction to Dynamic SQL

This chapter presents an overview of dynamic SQL, describes dynamic SQL statements, and describes the SQL descriptor area.

Overview

Dynamic SQL (*DSQL*) is a facility that lets programs accept or generate SQL statements at runtime. You need to use DSQL when you don't know the form or type of SQL statements and commands your program processes. For example, you would use DSQL for an interactive application program that takes user input and translates it into SQL statements. You would also use DSQL for a program that reads a file that contains SQL statements and parameters.

Because DSQL is more complex to code than embedded SQL, you should use embedded SQL whenever possible.

DSQL Statements

A *DSQL statement* is a string that includes:

- One of the following SQL data manipulation statements:
  - **delete** (both single and cursor deletions)
  - **insert**
  - **select** (excluding **select into**)
  - **update** (both single and cursor updates)
- Or one of the following SQL metadata commands:
  - **alter table**
  - **create index**
  - **create table**
  - **create view**
  - **drop index**
  - **drop table**
  - **drop view**
  - **grant**
  - **revoke**
- Or one of the following SQL transaction control commands:
  - **commit**
  - **rollback**

DSQL statements can't include:

- Any SQL statement not in the preceding list
- The **exec sql** flag
- SQL statement terminators (semicolons)
- References to host language variables

Table 1-1 compares two DSQL statements with the corresponding embedded SQL statements.

Table 1-1. Comparison of DSQL and Embedded SQL

DSQL Statement	Embedded SQL Statement
delete cities where population < 100	exec sql delete cities where population < 100;
alter table ski_areas drop state, add state char(2)	exec sql alter table ski_areas drop state, add state char(2);

Instructions for processing DSQL statements are presented in Chapter 2.

The SQL Descriptor Area

When you write a DSQL program, you use the SQL descriptor area (*SQLDA*), to communicate information between the program and the InterBase access method.

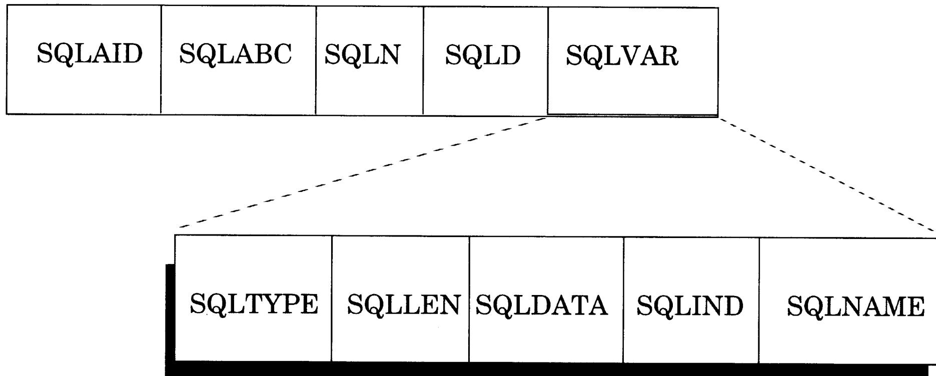
The SQLDA has two primary functions. To pass:

- Input parameters from the host program to SQL
- The output of **select** requests from SQL to the host program

A single SQLDA contains information about either input parameters or select list items at any given time. It never contains information about both simultaneously.

A pictoral representation of the SQLDA is shown in Figure 1-1. There is one occurrence of SQLVAR for each input parameter or select list item.

Figure 1-1. The SQLDA



Tables 1-2, 1-3, and 1-4 describe the fields, subfields, and valid datatypes for the SQLDA.

Table 1-2. SQLDA Fields

Field Name	**What It Does**
SQLAID	Acts as an internal identifier for the descriptor area. This string has the value "SQLDA" after a **describe** or **prepare** statement has been executed.
SQLABC	Indicates the length of the descriptor area. This length is calculated automatically according to the following formula: SQLABC = 16 + (43 * SQLN).
SQLN	Indicates the total number of select list items or input parameters represented in the descriptor area. This value specifies the number of expected occurrences of SQLVAR. The program sets the SQLN field. The value of this field must equal or exceed the value in SQLD.
SQLD	Indicates either the number of fields returned by a DSQL **select** statement or the number of parameter markers associated with the program. The program can use the SQLD field to: • Determine how many select fields to allocate storage for • Determine whether a prepared statement is a **select** statement • Tell SQL how many input parameters there are •The **select** field information is filled in automatically when you use a **prepare** or **describe** statement.
SQLVAR	Describes either of the following: • Each select list item in a DSQL **select** statement • Each parameter associated with a DSQL statement There is one SQLVAR for each select list item or parameter marker.

Table 1-3 describes the subfields that comprise the SQLVAR.

Table 1-3. SQLVAR Subfields

Field Name	**What It Does**
SQLTYPE	Indicates the data type of the select list item or input parameter. This information is filled in automatically when you use a **prepare** or **describe** statement.
SQLLEN	Indicates the length in bytes of the select list item or input parameter. This information is filled in automatically when you use a prepare or **describe** statement.
SQLDATA	Points to the address of the storage area allocated for the select list item or input parameter.
SQLIND	Points to the address of the indicator variable. This variable allows you to check for null or missing values.
SQLNAME	Names the field that the SQLVAR describes. This is the name of the select list item. This information is filled in automatically when you use a **prepare** or **describe** statement and your select item is a database field.

Table 1-4 lists the valid datatypes for the SQLTYPE field.

Table 1-4. Valid Datatypes

Value	Datatype	Nulls	Length	Associated Macro
448	Varying	No	Declared length	SQL_VARYING
449	Varying	Yes	Declared length	SQL_VARYING + 1
452	Character	No	Declared length	SQL_TEXT
453	Character	Yes	Declared length	SQL_TEXT + 1
480	Double	No	8 bytes	SQL_DOUBLE
481	Double	Yes	8 bytes	SQL_DOUBLE + 1
482	Float	No	4 bytes	SQL_FLOAT
483	Float	Yes	4 bytes	SQL_FLOAT + 1
496	Long	No	4 bytes	SQL_LONG
497	Long	No	4 bytes	SQL_LONG + 1
500	Short	No	2 bytes	SQL_SHORT
501	Short	Yes	2 bytes	SQL_SHORT + 1
510	Date	No	8 bytes	SQL_DATE
511	Date	Yes	8 bytes	SQL_DATE + 1
520	Blob	No	8 bytes	SQL_BLOB
521	Blob	Yes	8 bytes	SQL_BLOB + 1

DSQL Considerations

The following considerations apply to programming with DSQL:

- You can't access multiple databases through DSQL. If you declare more than one database, **gpre** uses the last one as its target. **gpre** also puts out a warning message indicating which database it selected.
- DSQL must use the default transaction.
- If your DSQL program opens, closes, and reopens the same database, you must include the **gds_$dsql_finish** routine prior to the **commit release** command.
- If you declare a database handle in your program through the GDML **database** declaration, name that database handle in the **gds_$dsql_finish** routine:

```
gds_$dsql_finish(database-handle);
EXEC SQL
      COMMIT RELEASE;
```

  If you don't declare a database handle in your program, name the default database handle in the **gds_$dsql_finish** routine:

```
gds_$dsql_finish(gds_$database);
EXEC SQL
      COMMIT RELEASE;
```

  In C programs, the database handle must be prefixed with an ampersand (&). For example:

```
gds_$dsql_finish(&database-handle);
EXEC SQL
      COMMIT RELEASE;

gds_$dsql_finish(&gds_$database);
EXEC SQL
      COMMIT RELEASE;
```

- If you plan to use DSQL in COBOL and BASIC programs, you must allocate all variables in advance. This is because you can't use pointers in these languages.

  For more information on using DSQL in COBOL and BASIC programs, refer to Chapter 4, *Setting Up an SQLDA*.
- If you plan to use DSQL in FORTRAN programs, note the following limitations. You:
  - Must allocate all variables in advance, since you can't use pointers in FORTRAN.
  - Can't easily declare complex structures in FORTRAN. Since the SQLDA is a complex structure, you must take special care to declare it.

For more information on using DSQL in FORTRAN programs, refer to Chapter 4, *Setting Up an SQLDA*.

For More Information

- For more information on DSQL, refer to:
- Chapter 2, *Processing DSQL Statements*, for information on processing DSQL statements
- Chapter 3, *Accessing Special Field Types*, for information on accessing blob and date fields with DSQL
- Chapter 4, *Setting up an SQLDA*, for information on setting up an SQLDA and on the structures that InterBase provides
- The appendix on reporting and handling errors in the *Programmer's Reference*, for information on DSQL error handling

Chapter 2
Processing DSQL Statements

This chapter describes how to process the following types of DSQL statements:

- A statement other than **select** that doesn't have parameter markers
- A statement other than **select** that has parameter markers
- A **select** statement that doesn't have parameter markers
- A **select** statement that has parameter markers
- An unknown statement

It also describes how to check for null values.

Overview

You process DSQL statements by using the nondynamic statements listed in Table 2-1.

Table 2-1. Statements for Processing DSQL Statements

Statement	What It Does
prepare	Checks the DSQL statement for errors. For **select** statements, fills in the SQLDA with information about the datatype, length, and name of select list fields.
describe	Fills in the SQLDA with a description of the data returned by a prepared DSQL statement.
execute	Executes a prepared statement.
execute immediate	Prepares and executes an SQL statement in one step. You can only use this statement for non-**select** statements that don't have parameter markers.
declare cursor	Declares a cursor for a prepared **select** statement.
open cursor	Opens a cursor declared for a prepared **select** statement.
fetch	Retrieves values from a cursor declared for a prepared **select** statement.

As you look at the examples in this chapter, note the following:

- Many of the examples use a constant literal string in the DSQL command. This is done so that the examples emphasize how to use the string, rather than how to build it.
- Languages that don't support a varying string datatype must treat varying data as simple text.

Processing Non-Select Statements with No Associated Parameters

The easiest type of DSQL statement to process is a statement that's not a **select** statement and that doesn't have associated parameters. This is because a non-**select** statement with no associated parameters doesn't have to access the SQLDA. Instead, it can generate a statement string and pass it to an **execute immediate** statement in the program.

The steps required to process this type of statement in C and Pascal programs are presented below. Each group of steps is followed by an example.

Steps for C Programs

To process a non-**select** statement that doesn't have parameters in a C program, follow these steps:

1. Include the SQL communications area. This defines the relevant type definitions:

```
EXEC SQL
      INCLUDE SQLCA;
```

2. Construct the DSQL string:

```
char *statement = "DELETE FROM CITIES WHERE STATE = 'MA' \
      AND POPULATION > 1000";
```

3. Prepare and execute the DSQL string:

```
EXEC SQL
     EXECUTE IMMEDIATE :statement;
```

C Program Example

The following program executes a DSQL string built by the program.

```
EXEC SQL
    INCLUDE SQLCA;


EXEC SQL
    WHENEVER SQLERROR GO TO ERR;

char *statement = "DELETE FROM CITIES WHERE STATE = 'MA'\
      AND POPULATION > 1000";
```

```
main() {

  EXEC SQL
       EXECUTE IMMEDIATE :statement;

  EXEC SQL
       COMMIT RELEASE;

  exit();

ERR:

  printf("Database error, SQLCODE = %d\n", SQLCODE);
  gds_$print_status (gds_$status);

  EXEC SQL
       ROLLBACK RELEASE;
}
```

Steps for Pascal Programs

To process a non-**select** statement that doesn't have parameters in a Pascal program, follow these steps:

1. Include an SQL communications area. This defines the relevant type definitions:

   ```
   EXEC SQL
         INCLUDE SQLCA;
   ```

2. Construct the DSQL string, keeping the query on a single line when you code the program :

   ```
   statement: array[1..61] of CHAR :=
    'DELETE FROM CITIES
          WHERE STATE = "MA" AND POPULATION > 1000';
   ```

3. Prepare and execute the DSQL string:

   ```
   EXEC SQL
        EXECUTE IMMEDIATE :statement;
   ```

Pascal Program Example

The program below executes a DSQL string built by the program. When you code the program, be sure to code the query on a single line.

```
program dsql (INPUT, OUTPUT);

    EXEC SQL
        INCLUDE SQLCA
var

    statement: array[1..61] of CHAR :=
       'DELETE FROM CITIES WHERE STATE = "MA" AND
             POPULATION > 1000';

label
    999;

begin

    EXEC SQL
         WHENEVER SQLERROR GO TO 999

    EXEC SQL
        EXECUTE IMMEDIATE :statement;

    EXEC SQL
         COMMIT RELEASE;

     if (sqlcode <> 0) begin
999: writeln ('Database error, SQLCODE = ', SQLCODE);
     gds_$print_status (gds_$status);

    EXEC SQL
         ROLLBACK RELEASE;
     end;
end.
```

Processing Non-Select Statements with Associated Parameters

You can associate input parameters with non-**select** statements in order to make them more flexible. You do this by coding parameter markers (question marks) when constructing the statement string for the **prepare** statement. At runtime, these markers are replaced by values entered dynamically.

To process a non-**select** statement that has associated parameters, you must always allocate an input SQLDA.

The steps required to process this type of statement in C and Pascal programs are presented below. Each group of steps is followed by an example.

Steps for C Programs

To process non-**select** statements that have associated parameters in C programs:

1. Include the SQL communications area. This defines the relevant type definitions:

```
EXEC SQL
     INCLUDE SQLCA;
```

2. Construct the DSQL string:

```
char *query =
      "UPDATE STATES SET AREA = ? WHERE STATE = ?"
```

3. First, allocate the structure of the SQLDA by using the SQLDA_LENGTH macro. This macro evaluates the amount of memory required to hold the specified number of input variables. Then, set SQLN and SQLD to the number of input variables:

```
sqlda = (SQLDA*) malloc (SQLDA_LENGTH (2));
sqlda->sqln = 2;
sqlda->sqld = 2;
```

4. Prepare the DSQL string:

```
EXEC SQL
   PREPARE Q FROM :query;
```

5. Fill in the SQLDA:

```
sqlda->sqlvar[1].sqldata = new_state;
sqlda->sqlvar[1].sqltype = SQL_TEXT;
sqlda->sqlvar[1].sqllen = 4;
```

```
sqlda->sqlvar[0].sqldata = (char*) &area;
sqlda->sqlvar[0].sqltype = SQL_LONG;
sqlda->sqlvar[0].sqllen = 4;
sqlda->sqlvar[0].sqld = 2;
```

Note

Languages, such as C, that do not support a varying string datatype must treat varying data as simple text.

6. Input the parameters:

```
while (1) {
   printf("Enter State to update: ");
   scanf("%s",new_state);
   if (strlen(new_state) == 2)
      strcat(new_state,"   ")
         printf("Enter new area: ");

   scanf("%d",&area);
```

7. Execute the DSQL string:

```
EXEC SQL
     EXECUTE Q USING DESCRIPTOR sqlda;
```

C Program Example

The following program shows how to use DSQL to update multiple parameters through a loop without having to issue a **prepare** statement each time through:

```
#include <stdio.h>
#include <string.h>

DATABASE DB = 'atlas.gdb';

EXEC SQL
    INCLUDE SQLCA;

EXEC SQL
    WHENEVER SQLERROR GO TO ERR;

char *query =
      "UPDATE STATES SET AREA = ? WHERE STATE = ?";
char new_state[5];
int area;
```

```
SQLDA *sqlda;

main ()
{
    READY;
    START_TRANSACTION;

    sqlda = (SQLDA*) malloc (SQLDA_LENGTH (2));
    sqlda->sqln = 2;
    sqlda->sqld = 2;

    EXEC SQL
        PREPARE Q FROM :query;

    sqlda->sqlvar[1].sqldata = new_state;
    sqlda->sqlvar[1].sqltype = SQL_TEXT;
    sqlda->sqlvar[1].sqllen = 4;

    sqlda->sqlvar[0].sqldata = (char*) &area;
    sqlda->sqlvar[0].sqltype = SQL_LONG;
    sqlda->sqlvar[0].sqllen = 4;


    while (1) {
        printf("Enter State to update: ");
        scanf("%s",new_state);
        if (strlen(new_state) == 2)
           strcat(new_state,"  ");
                printf("Enter new area: ");

        scanf("%d",&area);
                if (area == 0)
            break;

        EXEC SQL
            EXECUTE Q USING DESCRIPTOR sqlda;

            FOR X IN STATES WITH X.STATE = new_state
            printf ("%s %d\n", X.STATE, X.AREA);
        END_FOR;
    }
        EXEC SQL
            COMMIT RELEASE;
```

```
    exit ();

ERR: printf ("Data base error, SQLCODE = %d\n", SQLCODE);
     gds_$print_status (gds_$status);

     EXEC SQL
         ROLLBACK RELEASE;
}
```

Steps for Pascal Programs

To process non-**select** statements that have associated parameters in Pascal programs:

1. Include the SQL communications area. This defines the relevant type definitions:

```
EXEC SQL
     INCLUDE SQLCA;
```

2. Construct the DSQL string:

```
query : array [1..61] of char :=
    'UPDATE STATES SET AREA = ? WHERE STATE = ?';
```

3. Declare an SQLDA:

```
var
    sqlda1 : sqlda;
```

4. Set SQLN and SQLD to the number of input variables:

```
sqlda1.sqln :=2;
sqlda1.sqld := 2;
```

5. Prepare the DSQL string:

```
EXEC SQL
     PREPARE Q FROM :query;
```

6. Fill in the SQLDA:

```
sqlda1.salvars[2].sqldata := addr(new_state);
sqlda1.sqlvars[2].sqlind := addr(flag2);
sqlda1.sqlvars[2].sqltype := SQL_TEXT;
sqlda1.sqlvars[2].sqllen := 4;

sqlda1.salvars[2].sqldata := addr(area);
sqlda1.sqlvars[2].sqlind := addr(flag1);
```

```
sqlda1.sqlvars[2].sqltype := SQL_LONG;
sqlda1.sqlvars[2].sqllen := 4;
```

7. **Input the parameters:**

```
while (continue = 1) do
 begin
   write('Enter State to update: ');
   readln(new_state);

   write('Enter new area: ');
   readln(area);
```

8. **Execute the DSQL string:**

```
EXEC SQL
     EXECUTE Q USING DESCRIPTOR sqlda;
```

Pascal Progam Example

The program below shows how to use DSQL to update multiple parameters through a loop without having to issue a **prepare** statement each time through:

```
program dsql_update_param (INPUT, OUTPUT);

DATABASE DB = 'atlas.gdb';

EXEC SQL
    INCLUDE SQLCA;

EXEC SQL
    WHENEVER SQLERROR GO TO 999

var
    query : array [1..61] of char :=
       'UPDATE STATES SET AREA = ? WHERE STATE = ?';
    new_state : array [1..4] of char;
    area : integer32;
    flag1, flag2 : integer16;
    continue : integer16;
    loop : integer16;
    sqlda1 : sqlda;

label
   999;
```

```
begin

    READY;
    START_TRANSACTION;

    sqlda1.sqln := 2;
    sqlda1.sqld := 2;
    continue := 1;

    EXEC SQL
        PREPARE Q FROM :query;

    sqlda1.sqlvars[2].sqldata := addr(new_state);
    sqlda1.sqlvars[2].sqlind  := addr(flag2);
    sqlda1.sqlvars[2].sqltype := SQL_TEXT;
    sqlda1.sqlvars[2].sqllen := 4;

    sqlda1.sqlvars[1].sqldata := addr(area);
    sqlda1.sqlvars[1].sqlind  := addr(flag1);
    sqlda1.sqlvars[1].sqltype := SQL_LONG;
    sqlda1.sqlvars[1].sqllen := 4;

    while (continue = 1) do
    begin
        write('Enter State to update: ');
        readln(new_state);

        write('Enter new area: ');
        readln(area);

        if area = 0 then continue := 0
        else
         begin
            EXEC SQL
                EXECUTE Q USING DESCRIPTOR sqlda1;

            FOR X IN STATES WITH X.STATE = new_state
                writeln (X.STATE,' ',X.AREA);
            END_FOR;
         end;
    end;
```

```
      EXEC SQL
          COMMIT RELEASE;

      if (sqlcode <> 0) begin
999:  writeln ('Data base error, SQLCODE = ', SQLCODE);
      gds_$print_status (gds_$status);

      EXEC SQL
           ROLLBACK RELEASE;

      end;
end.
```

Processing Select Statements with No Associated Parameters

You must always use the SQLDA t o process **select** statements. This is because the SQLDA describes the output that is returned to the program. You must also open a cursor and use **fetch** statements to retrieve these values.

The steps required to process this type of statement in C and Pascal programs are presented below. Each group of steps is followed by an example.

Steps for C Programs

To process **select** statements with no associated parameters in C programs:

1. Include an SQL communications area. This defines the relevant type definitions:

```
EXEC SQL
   INCLUDE SQLCA;
```

2. Construct the DSQL string:

```
char *string = "SELECT CITY, STATE, POPULATION, ALTITUDE   \
       FROM CITIES WHERE STATE = 'CA' ORDER BY CITY";
```

3. First, allocate the structure of the SQLDA by using the SQLDA_LENGTH macro. Then, set SQLN to the number of select list items:

```
output_sqlda = (SQLDA*) malloc (SQLDA_LENGTH (4));
output_sqlda->sqln = 4;
```

4. Prepare the DSQL string:

```
EXEC SQL
     PREPARE QUERY INTO output_sqlda FROM :string;
```

5. Fill in the SQLDA:

```
output_sqlda->sqlvar[0].sqldata = new_city;
new_city[25] = 0;
output_sqlda->sqlvar[0].sqlind = &flag0;
output_sqlda->sqlvar[0].sqltype = SQL_TEXT + 1;

output_sqlda->sqlvar[1].sqldata = new_state;
new_state[4] = 0;
output_sqlda->sqlvar[1].sqlind = &flag1;
output_sqlda->sqlvar[1].sqltype = SQL_TEXT + 1;
```

```
output_sqlda->sqlvar[2].sqldata = (char*) &new_pop;
output_sqlda->sqlvar[2].sqlind = &flag2;
output_sqlda->sqlvar[2].sqltype = SQL_LONG + 1;

output_sqlda->sqlvar[3].sqldata = (char*) &new_altitude;
output_sqlda->sqlvar[3].sqlind = &flag3;
output_sqlda->sqlvar[3].sqltype = SQL_LONG + 1;
```

6. Declare the cursor:

```
EXEC SQL
     DECLARE C CURSOR FOR QUERY;
```

7. Open the cursor:

```
EXEC SQL
     OPEN C;
```

8. Retrieve information using the cursor:

```
EXEC SQL
     FETCH C USING DESCRIPTOR output_sqlda;
```

C Program Example

The following program shows how to process a **select** statement that has no associated parameters.

```
#include <stdio.h>
#include <strings.h>

#define MISSING_STRING "*"


EXEC SQL
    INCLUDE SQLCA;

EXEC SQL
    WHENEVER SQLERROR GO TO ERR;

char *string = "SELECT CITY, STATE, POPULATION, ALTITUDE      \
       FROM CITIES WHERE STATE = 'CA' ORDER BY CITY";
char new_city[26], new_state[5];
int new_pop, new_altitude;
short flag0, flag1, flag2, flag3;
```

```
SQLDA *output_sqlda;

main ()
{
    READY;
    START_TRANSACTION;

    output_sqlda = (SQLDA*) malloc (SQLDA_LENGTH (4));
    output_sqlda->sqln = 4;

    EXEC SQL
        PREPARE QUERY INTO output_sqlda FROM :string;

    output_sqlda->sqlvar[0].sqldata = new_city;
    new_city[25] = 0;
    output_sqlda->sqlvar[0].sqlind = &flag0;
    output_sqlda->sqlvar[0].sqltype = SQL_TEXT + 1;

    output_sqlda->sqlvar[1].sqldata = new_state;
    new_state[4] = 0;
    output_sqlda->sqlvar[1].sqlind = &flag1;
    output_sqlda->sqlvar[1].sqltype = SQL_TEXT + 1;

    output_sqlda->sqlvar[2].sqldata = (char*) &new_pop;
    output_sqlda->sqlvar[2].sqlind = &flag2;
    output_sqlda->sqlvar[2].sqltype = SQL_LONG + 1;

    output_sqlda->sqlvar[3].sqldata = (char*) &new_altitude;
    output_sqlda->sqlvar[3].sqlind = &flag3;
    output_sqlda->sqlvar[3].sqltype = SQL_LONG + 1;


    EXEC SQL
        DECLARE C CURSOR FOR QUERY;

    EXEC SQL
        OPEN C;

    EXEC SQL
        FETCH C USING DESCRIPTOR output_sqlda;

    while (SQLCODE == 0) {
        if (flag0 >= 0)
```

```
                printf ("%s",new_city);
            else
                printf("%s",MISSING_STRING);

            if (flag1 >= 0)
                printf ("\t%s",new_state);
            else
                printf("\t%s",MISSING_STRING);

            if (flag2 >=0)
                printf ("\t%d",new_pop);
            else
                printf("\t%s",MISSING_STRING);

            if (flag3 >= 0)
                printf("\t%d\n", new_altitude);
            else
                printf("\t%s\n",MISSING_STRING);

            EXEC SQL
                FETCH C USING DESCRIPTOR output_sqlda;
        }

        EXEC SQL
            ROLLBACK RELEASE;
        exit();

ERR: printf ("Data base error, SQLCODE = %d\n", SQLCODE);
     gds_$print_status (gds_$status);

     EXEC SQL
        ROLLBACK RELEASE;
}
```

Steps for Pascal Programs

To process **select** statements with no associated parameters in Pascal programs:

1. Include an SQL communications area. This defines the relevant type definitions:

```
EXEC SQL
   INCLUDE SQLCA;
```

2. Construct the DSQL string, keeping the query on a single line when you code the program:

```
query : array [1..91] of char :=
     'SELECT CITY, STATE, POPULATION, ALTITUDE
         FROM CITIES WHERE STATE = "CA" ORDER BY CITY';
```

3. Declare an SQLDA:

```
var
       output_sqlda : sqlda;
```

4. Set SQLN to the number of select fields:

```
output_sqlda.sqln := 4;
```

5. Prepare the DSQL string:

```
EXEC SQL
   PREPARE QUERY INTO output_sqlda FROM :query;
```

6. Fill in the SQLDA:

```
output_sqlda.sqlvars[1].sqldata := addr(new_city);
output_sqlda.sqlvars[1].sqlind  := addr(flag0);
output_sqlda.sqlvars[1].sqltype := SQL_TEXT + 1;

output_sqlda.sqlvars[2].sqldata := addr(new_state);
output_sqlda.sqlvars[2].sqlind  := addr(flag1);
output_sqlda.sqlvars[2].sqltype := SQL_TEXT + 1;

output_sqlda.sqlvars[3].sqldata := addr(new_pop);
output_sqlda.sqlvars[3].sqlind  := addr(flag2);
output_sqlda.sqlvars[3].sqltype := SQL_LONG + 1;

output_sqlda.sqlvars[4].sqldata := addr(new_altitude);
output_sqlda.sqlvars[4].sqlind  := addr(flag3);
output_sqlda.sqlvars[4].sqltype := SQL_LONG + 1;
```

7. Declare the cursor:

```
EXEC SQL
   DECLARE C CURSOR FOR QUERY;
```

8. Open the cursor:

```
EXEC SQL
   OPEN C;
```

9. Retrieve information using the cursor:

```
EXEC SQL
   FETCH C USING DESCRIPTOR output_sqlda;
```

Pascal Program Example

The following program shows how to process a select statement that has no associated parameters. When you code the program, be sure to code the DSQL query on a single line.

```
program dsql_sel_with (INPUT, OUTPUT);

EXEC SQL
    INCLUDE SQLCA;


EXEC SQL
    WHENEVER SQLERROR GO TO 999

const
    MISSING_STRING = '*';
    TAB_STRING = '        ';
var
    query : array [1..91] of char :=
         'SELECT CITY, STATE, POPULATION, ALTITUDE
             FROM CITIES WHERE STATE = "CA" ORDER BY CITY';
    new_state : array [1..4] of char;
    new_city : array [1..25] of char;
    new_pop : integer32;
    new_altitude : integer32;
    flag0, flag1, flag2, flag3 : integer16;
    loop : integer16;

    output_sqlda : sqlda;

label
   999;

begin

    READY;
    START_TRANSACTION;
```

```
output_sqlda.sqln := 4;

EXEC SQL
    PREPARE QUERY INTO output_sqlda FROM :query;

output_sqlda.sqlvars[1].sqldata := addr(new_city);
output_sqlda.sqlvars[1].sqlind  := addr(flag0);
output_sqlda.sqlvars[1].sqltype := SQL_TEXT + 1;

output_sqlda.sqlvars[2].sqldata := addr(new_state);
output_sqlda.sqlvars[2].sqlind  := addr(flag1);
output_sqlda.sqlvars[2].sqltype := SQL_TEXT + 1;

output_sqlda.sqlvars[3].sqldata := addr(new_pop);
output_sqlda.sqlvars[3].sqlind  := addr(flag2);
output_sqlda.sqlvars[3].sqltype := SQL_LONG + 1;

output_sqlda.sqlvars[4].sqldata := addr(new_altitude);
output_sqlda.sqlvars[4].sqlind  := addr(flag3);
output_sqlda.sqlvars[4].sqltype := SQL_LONG + 1;

EXEC SQL
    DECLARE C CURSOR FOR QUERY;

EXEC SQL
    OPEN C;

EXEC SQL
    FETCH C USING DESCRIPTOR output_sqlda;


while (SQLCODE = 0) do
begin
    if flag0 >= 0 then
        write(new_city)
    else
        write(MISSING_STRING);

    if flag1 >= 0 then
        write(TAB_STRING, new_state)
    else
        write(TAB_STRING, MISSING_STRING);
```

```
        if flag2 >= 0 then
            write(TAB_STRING, new_pop)
        else
            write(TAB_STRING, MISSING_STRING);

        if flag3 >= 0 then
            writeln(TAB_STRING, new_altitude)
        else
            writeln(TAB_STRING, MISSING_STRING);

        EXEC SQL
            FETCH C USING DESCRIPTOR output_sqlda;
    end;

    EXEC SQL
        ROLLBACK RELEASE;

    if (sqlcode <> 0 and SQLCODE <> 100) begin
999: writeln ('Data base error, SQLCODE = ', SQLCODE);
     gds_$print_status (gds_$status);

     EXEC SQL
         ROLLBACK RELEASE;

    end;
end.
```

Processing Select Statements with Associated Parameters

Processing **select** statements that have associated parameters involves setting up input variables as well as setting up an output SQLDA.

The steps required to process this type of statement in C and Pascal programs are presented below. Each group of steps is followed by an example.

Steps for C Programs

To process select statements that have associated parameters in C programs:

1. Include an SQL communications area. This defines the relevant type definitions:

```
EXEC SQL
      INCLUDE SQLCA;
```

2. Construct the DSQL string:

```
char *string = "SELECT CITY, STATE, POPULATION, ALTITUDE  \
      FROM CITIES WHERE STATE = ? ORDER BY CITY";
```

3. Declare the input and output SQLDAs:

```
SQLDA *input_sqlda;
SQLDA *output_sqlda;
```

4. First, allocate the structure of the input SQLDA by using the SQLDA_LENGTH macro. Then, set SQLN and SQLD to the number of input variables:

```
input_sqlda = (SQLDA*) malloc (SQLDA_LENGTH (1));
input_sqlda->sqln = 1;
input_sqlda->sqld = 1;
```

5. First, allocate the structure of the output SQLDA by using the SQLDA_LENGTH macro. Then, set SQLN to the number of select list items:

```
output_sqlda = (SQLDA*) malloc (SQLDA_LENGTH (4));
output_sqlda->sqln = 4;
```

6. Prepare the DSQL string into the output SQLDA:

```
EXEC SQL
      PREPARE QUERY INTO output_sqlda FROM :string;
```

7. **Fill in the input SQLDA:**

```
input_sqlda->sqlvar[0].sqldata = new_state;
input_sqlda->sqlvar[0].sqltype = SQL_TEXT;
input_sqlda->sqlvar[0].sqllen = 4;
```

8. **Fill in the output SQLDA:**

```
output_sqlda->sqlvar[0].sqldata = new_city;
new_city[25] = 0;
output_sqlda->sqlvar[0].sqlind = &flag0;
output_sqlda->sqlvar[0].sqltype = SQL_TEXT + 1;

output_sqlda->sqlvar[1].sqldata = new_state;
new_state[4] = 0;
output_sqlda->sqlvar[1].sqlind = &flag1;
output_sqlda->sqlvar[1].sqltype = SQL_TEXT + 1;

output_sqlda->sqlvar[2].sqldata = (char*) &new_pop;
output_sqlda->sqlvar[2].sqlind = &flag2;
output_sqlda->sqlvar[2].sqltype = SQL_LONG + 1;

output_sqlda->sqlvar[3].sqldata = (char*) &new_altitude;
output_sqlda->sqlvar[3].sqlind = &flag3;
output_sqlda->sqlvar[3].sqltype = SQL_LONG + 1;
```

9. **Input the parameters:**

```
printf("Enter 2 character state code: ");
   scanf("%s",new_state);
   if (strlen(new_state) == 2)
      strcat(new_state,"  ");
```

10. **Declare the cursor:**

```
EXEC SQL
     DECLARE C CURSOR FOR QUERY;
```

11. **Open the cursor using the input SQLDA:**

```
EXEC SQL
     OPEN C USING DESCRIPTOR input_sqlda;
```

12. **Fetch the cursor using the output SQLDA:**

```
EXEC SQL
     FETCH C USING DESCRIPTOR output_sqlda;
```

C Program Example

The following program shows how to process a **select** statement that has associated parameters.

```
#include <stdio.h>
#include <strings.h>

#define MISSING_STRING "*"

EXEC SQL
    INCLUDE SQLCA;

EXEC SQL
    WHENEVER SQLERROR GO TO ERR;

char *string = "SELECT CITY, STATE, POPULATION, ALTITUDE      \
      FROM CITIES WHERE STATE = ? ORDER BY CITY";
char new_city[26], new_state[5];
int new_pop, new_altitude;
short flag0, flag1, flag2, flag3;


SQLDA *input_sqlda;
SQLDA *output_sqlda;

main ()
{
    READY;
    START_TRANSACTION;

    input_sqlda = (SQLDA*) malloc (SQLDA_LENGTH (1));
    input_sqlda->sqln = 1;
    input_sqlda->sqld = 1;

    output_sqlda = (SQLDA*) malloc (SQLDA_LENGTH (4));
    output_sqlda->sqln = 4;

    input_sqlda->sqlvar[0].sqldata = new_state;
    input_sqlda->sqlvar[0].sqltype = SQL_TEXT;
    input_sqlda->sqlvar[0].sqllen = 4;

    EXEC SQL
        PREPARE QUERY INTO output_sqlda FROM :string;
```

```
output_sqlda->sqlvar[0].sqldata = new_city;
new_city[25] = 0;
output_sqlda->sqlvar[0].sqlind = &flag0;
output_sqlda->sqlvar[0].sqltype = SQL_TEXT + 1;

output_sqlda->sqlvar[1].sqldata = new_state;
new_state[4] = 0;
output_sqlda->sqlvar[1].sqlind = &flag1;
output_sqlda->sqlvar[1].sqltype = SQL_TEXT + 1;

output_sqlda->sqlvar[2].sqldata = (char*) &new_pop;
output_sqlda->sqlvar[2].sqlind = &flag2;
output_sqlda->sqlvar[2].sqltype = SQL_LONG + 1;

output_sqlda->sqlvar[3].sqldata = (char*) &new_altitude;
output_sqlda->sqlvar[3].sqlind = &flag3;
output_sqlda->sqlvar[3].sqltype = SQL_LONG + 1;


printf("Enter 2 character state code: ");
scanf("%s",new_state);
if (strlen(new_state) == 2)
    strcat(new_state,"  ");

EXEC SQL
    DECLARE C CURSOR FOR QUERY;

EXEC SQL
    OPEN C USING DESCRIPTOR input_sqlda;

EXEC SQL
    FETCH C USING DESCRIPTOR output_sqlda;


while (SQLCODE == 0) {
    if (flag0 >=0)
         printf ("%s",new_city);
    else
         printf("%s",MISSING_STRING);

    if (flag1 >=0)
         printf ("\t%s",new_state);
```

```
        else
             printf("\t%s",MISSING_STRING);

        if (flag2 >=0)
             printf ("\t%d",new_pop);
        else
             printf("\t%s",MISSING_STRING);

        if (flag3 >=0)
             printf("\t%d\n", new_altitude);
        else
             printf("\t%s\n",MISSING_STRING);

        EXEC SQL
            FETCH C USING DESCRIPTOR output_sqlda;
    }

    EXEC SQL
        ROLLBACK RELEASE;
    exit();

ERR: printf ("Data base error, SQLCODE = %d\n", SQLCODE);
     gds_$print_status (gds_$status);

     EXEC SQL
        ROLLBACK RELEASE;
}
```

Steps for Pascal Programs

To process **select** statements that have associated parameters in Pascal programs:

1. Include an SQL communications area. This defines the relevant type definitions:

```
EXEC SQL
      INCLUDE SQLCA;
```

2. Construct the DSQL string, keeping the query on a single line when you code the program:

```
query : array [1..91] of char :=
    'SELECT CITY, STATE, POPULATION, ALTITUDE
         FROM CITIES WHERE STATE = ? ORDER BY CITY';
```

3. Declare the input and output SQLDAs:

```
var
    input_sqlda : sqlda;
    output_sqlda : sqlda;
```

4. Set the SQLN and SQLD fields in the input SQLDA to the number of input variables:

```
input_sqlda.sqln := 1;
input_sqlda.sqld := 1;
```

5. Set the SQLN field in the output SQLDA to the number of select fields:

```
output_sqlda.sqln := 4;
```

6. Prepare the DSQL string into the output SQLDA:

```
EXEC SQL
   PREPARE QUERY INTO output_sqlda FROM :query;
```

7. Fill in the input SQLDA:

```
input_sqlda.sqlvars[1].sqldata := addr(new_state);
input_sqlda.sqlvars[1].sqltype := SQL_TEXT;
input_sqlda.sqlvars[1].sqllen := 4;
```

8. Fill in the output SQLDA:

```
output_sqlda.sqlvars[1].sqldata := addr(new_city);
output_sqlda.sqlvars[1].sqlind  := addr(flag0);
output_sqlda.sqlvars[1].sqltype := SQL_TEXT + 1;

output_sqlda.sqlvars[2].sqldata := addr(new_state);
output_sqlda.sqlvars[2].sqlind  := addr(flag1);
output_sqlda.sqlvars[2].sqltype := SQL_TEXT + 1;

output_sqlda.sqlvars[3].sqldata := addr(new_pop);
output_sqlda.sqlvars[3].sqlind  := addr(flag2);
output_sqlda.sqlvars[3].sqltype := SQL_LONG + 1;

output_sqlda.sqlvars[4].sqldata := addr(new_altitude);
output_sqlda.sqlvars[4].sqlind  := addr(flag3);
output_sqlda.sqlvars[4].sqltype := SQL_LONG + 1;
```

9. Input the parameters:

```
write('Enter 2 character State code: ');
   readln(new_state);
```

10. Declare the cursor:

```
EXEC SQL
     DECLARE C CURSOR FOR QUERY;
```

11. Open the cursor using the input SQLDA:

```
EXEC SQL
     OPEN C USING DESCRIPTOR input_sqlda;
```

12. Fetch the cursor using the output SQLDA:

```
EXEC SQL
     FETCH C USING DESCRIPTOR output_sqlda;
```

Pascal Program Example

The following program shows how to process a **select** statement that has associated parameters. When you code the program, be sure to code the DSQL query on a single line.

```
program dsql_sel_with (INPUT, OUTPUT);

EXEC SQL
    INCLUDE SQLCA;

EXEC SQL
    WHENEVER SQLERROR GO TO 999

const
    MISSING_STRING = '*';
    TAB_STRING = '        ';
var
    query : array [1..91] of char :=
       'SELECT CITY, STATE, POPULATION, ALTITUDE
              FROM CITIES WHERE STATE = ? ORDER BY CITY';
    new_state : array [1..4] of char;
    new_city : array [1..25] of char;
    new_pop : integer32;
    new_altitude : integer32;
    flag0, flag1, flag2, flag3 : integer16;
    loop : integer16;

    input_sqlda : sqlda;
    output_sqlda : sqlda;
label
```

```
    999;

begin

    READY;
    START_TRANSACTION;

    input_sqlda.sqln := 1;
    input_sqlda.sqld := 1;
    output_sqlda.sqln := 4;

    input_sqlda.sqlvars[1].sqldata := addr(new_state);
    input_sqlda.sqlvars[1].sqltype := SQL_TEXT;
    input_sqlda.sqlvars[1].sqllen := 4;

    EXEC SQL
        PREPARE QUERY INTO output_sqlda FROM :query;

    output_sqlda.sqlvars[1].sqldata := addr(new_city);
    output_sqlda.sqlvars[1].sqlind  := addr(flag0);
    output_sqlda.sqlvars[1].sqltype := SQL_TEXT + 1;

    output_sqlda.sqlvars[2].sqldata := addr(new_state);
    output_sqlda.sqlvars[2].sqlind  := addr(flag1);
    output_sqlda.sqlvars[2].sqltype := SQL_TEXT + 1;

    output_sqlda.sqlvars[3].sqldata := addr(new_pop);
    output_sqlda.sqlvars[3].sqlind  := addr(flag2);
    output_sqlda.sqlvars[3].sqltype := SQL_LONG + 1;

    output_sqlda.sqlvars[4].sqldata := addr(new_altitude);
    output_sqlda.sqlvars[4].sqlind  := addr(flag3);
    output_sqlda.sqlvars[4].sqltype := SQL_LONG + 1;

    write('Enter 2 character State code: ');
    readln(new_state);

    EXEC SQL
        DECLARE C CURSOR FOR QUERY;

    EXEC SQL
        OPEN C USING DESCRIPTOR input_sqlda;
```

```
    EXEC SQL
        FETCH C USING DESCRIPTOR output_sqlda;

    while (SQLCODE = 0) do
    begin
        if flag0 >= 0 then
            write(new_city)

else
            write(MISSING_STRING);

        if flag1 >= 0 then
            write(TAB_STRING, new_state)
        else
            write(TAB_STRING, MISSING_STRING);

        if flag2 >= 0 then
            write(TAB_STRING, new_pop)
        else
            write(TAB_STRING, MISSING_STRING);

        if flag3 >= 0 then
            writeln(TAB_STRING, new_altitude)
        else
            writeln(TAB_STRING, MISSING_STRING);

        EXEC SQL
            FETCH C USING DESCRIPTOR output_sqlda;
    end;

    EXEC SQL
        ROLLBACK RELEASE;

    if (sqlcode <> 0 and sqlcode <> 100) begin
999: writeln ('Data base error, SQLCODE = ', SQLCODE);
     gds_$print_status (gds_$status);


     EXEC SQL
         ROLLBACK RELEASE;
    end;
end.
```

Processing Unknown Statements

The most complicated statements for programs to handle in DSQL are those about which the program has no prior information. These statements may or may not have select list items or parameter markers.

There are a number of ways to process unknown statements. The example presented below shows one way to process these statements. It assumes that the statements do not have input parameters. It also uses the non-SQL **database** declaration, **ready** command, and **start_transaction** command, in order to work with an unknown database.

Steps for C Programs

To process unknown statements in C programs:

1. First, allocate the structure of the output SQLDA by using the SQLDA_LENGTH macro. Then, set SQLN to a value that you think is large enough to accommodate the unknown request. In this example, the value is set to 20:

   ```
   sqlda = (SQLDA*) malloc (SQLDA_LENGTH (20));
   sqlda->sqln = 20;
   ```

2. Construct the DSQL string by prompting the user for input and reading one character at a time:

   ```
   /* Prompt for SQL statements and process them */

   while (1)
       {
       /* get command */
       get_statement (statement);

       /* Check for command to leave */
       if ((UPPER (statement[0]) == 'E') &&
            (UPPER (statement[1]) == 'X') &&
            (UPPER (statement[2]) == 'I') &&
            (UPPER (statement[3]) == 'T'))
             {
       COMMIT;
       FINISH;
       return;
             }
       if ((UPPER (statement[0]) == 'Q') &&
            (UPPER (statement[1]) == 'U') &&
   ```

```
        (UPPER (statement[2]) == 'I') &&
        (UPPER (statement[3]) == 'T'))
         {
    ROLLBACK;
    FINISH;
    return;
         }

    process_statement (statement, sqlda);
    }
}


static get_statement (statement)
    char *statement;
{
/**************************************
 *
 * g e t _ s t a t e m e n t
 *
 **************************************
 *
 * Functional description
 * Get an SQL statment, or QUIT/EXIT command to process
 *
 **************************************/
char *p;
short  c, blank, done;

blank = 1;
while (blank)
    {
    printf ("SQL> ");
    blank = 1;
    p = statement;
    done = 0;
    while (!done)
        switch (c = getchar ())
            {
            case EOF:
                strcpy (statement, "EXIT");
                return;
```

```
                    case '\n':
                        if (p == statement || p [-1] != '-')
                            done = 1;
                        else
                            {
                            p [-1] = ' ';
                            printf ("CON> ");
                            }
                        break;

                    case '\t':
                    case ' ':
                        *p++ = ' ';
                        break;

                    default:
                        *p++ = c;
                        blank = 0;
                        break;
                    }
            *p = 0;
            }
        }
```

3. Prepare the DSQL string into the output SQLDA:

```
EXEC SQL PREPARE Q1 INTO sqlda FROM :string;
```

4. Check to see if the SQLD field is 0. This indicates whether or not the statement is a **select** statement. If the SQLD field is 0, the statement *is not* a **select** statement and can be processed accordingly:

```
if (!sqlda->sqld)
    {
    EXEC SQL EXECUTE Q1;
    return;
    }
```

If the SQLD field is not 0, the statement *is* a **select** statement. In this case, you need to follow Steps 5 through 10.

5. Check to see if the output SQLD is greater than the output SQLN. If the output SQLD *is* greater than the output SQLN, the number of select items returned is too large for the space allocated. In this case, return an error message:

```
else if (sqlda->sqld > sqlda->sqln)
    {
```

```
printf ("Statment not executed, can only select %d
       items\n", sqlda->sqln);
return;
}
```

6. Declare the cursor:

```
EXEC SQL DECLARE C CURSOR FOR Q1;
```

7. Open the cursor:

```
EXEC SQL OPEN C;
```

8. Fetch the cursor using the output SQLDA, continuing until SQLCODE becomes non-zero:

```
EXEC SQL FETCH C USING DESCRIPTOR sqlda;
```

9. When you finish processing the statement, close the cursor and process the next statement:

```
EXEC SQL CLOSE C;
```

C Program Example

The following program shows how to use DSQL to process unknown statements:

```
#include <stdio.h>
#ifdef VMS
#include time.h
#else
#include <sys/time.h>
#endif

extern char*malloc();

DATABASE DB = COMPILETIME "atlas.gdb"
           RUNTIME db_name;

chardb_name[128];
/***************************************
 *
 * Macro for aligning buffer space to boundaries
 *   needed by a datatype on a given machine
 *
 ***************************************/
#ifdef VMS
```

```
#define ALIGN(n,b)(n)
#endif

#ifdef sun
#ifdef sparc
#define ALIGN(n,b)((n + b - 1) & ~(b - 1))
#endif
#endif

#ifdef hpux
#define ALIGN(n,b)((n + b - 1) & ~(b - 1))
#endif

#ifdef ultrix
#define ALIGN(n,b)((n + b - 1) & ~(b - 1))
#endif

#ifdef AIX
#define ALIGN(n,b)((n + b - 1) & ~(b - 1))
#endif

#ifndef ALIGN
#define ALIGN(n,b)((n+1) & ~1)
#endif

#define UPPER(c) ((c >= 'a' && c<= 'z') ? c + 'A' - 'a': c )

static char*alpha_months [] =
          {
          "January",
          "February",
          "March",
          "April",
          "May",
          "June",
          "July",
          "August",
          "September",
          "October",
          "November",
          "December"
          };
```

```
typedef struct vary {
    shortvary_length;
    charvary_string [1];
} VARY;


main (argc, argv)
    intargc;
    char*argv[];
{
/*************************************
 *
 *m a i n
 *
 *************************************
 *
 * Functional description
 *Process sql statements until utterly bored.
 *
 *************************************/
char  statement [512];
SQLDA  *sqlda;

/* Get database name */

if (argc < 2)
    {
    fprintf (stderr, "No database specified on command
          line\n");
    exit (1);
    }
++argv;
strcpy (db_name, *argv);

/* Open database and start tansaction */

READY;
START_TRANSACTION;

/*Set up SQLDA for SELECTs, allow up to 20 fields to be
  selected */

sqlda = (SQLDA*) malloc (SQLDA_LENGTH (20));
```

```
sqlda->sqln = 20;

/* Prompt for SQL statements and process them */

while (1)
    {
    /* get command */
    get_statement (statement);

    /* Check for command to leave */
    if ((UPPER (statement[0]) == 'E') &&
        (UPPER (statement[1]) == 'X') &&
        (UPPER (statement[2]) == 'I') &&
        (UPPER (statement[3]) == 'T'))
         {
    COMMIT;
    FINISH;
    return;
         }
    if ((UPPER (statement[0]) == 'Q') &&
        (UPPER (statement[1]) == 'U') &&
        (UPPER (statement[2]) == 'I') &&
        (UPPER (statement[3]) == 'T'))
         {
    ROLLBACK;
    FINISH;
    return;
         }

    process_statement (statement, sqlda);
    }
}


static get_statement (statement)
    char *statement;
{
/*************************************
 *
 *g e t _ s t a t e m e n t
 *
 *************************************
 *
```

```
 * Functional description
 * Get an SQL statment, or QUIT/EXIT command to process
 *
 *************************************/
char *p;
short  c, blank, done;

blank = 1;
while (blank)
    {
    printf ("SQL> ");
    blank = 1;
    p = statement;
    done = 0;
    while (!done)
        switch (c = getchar ())
            {
            case EOF:
                strcpy (statement, "EXIT");
                return;

            case '\n':
                if (p == statement || p [-1] != '-')
                    done = 1;
                else
                    {
                    p [-1] = ' ';
                    printf ("CON> ");
                    }
                break;

            case '\t':
            case ' ':
                *p++ = ' ';
                break;

            default:
                *p++ = c;
                blank = 0;
                break;
            }
    *p = 0;
    }
```

```
}




static print_item (s, var)
    char    **s;
    SQLVAR  *var;
{
/*************************************
 *
 *p r i n t _ i t e m
 *
 *************************************
 *
 * Functional description
 * Print a SQL data item as described.
 *
 *************************************/
char     *p, *string;
char      d[20];
VARY     *vary;
short     dtype;
struct tm  times;

p = *s;

dtype = var->sqltype & ~1;
if ((var->sqltype & 1) && (*var->sqlind < 0))
    /* If field was missing print <null> */
    {
    if ((dtype == SQL_TEXT) || (dtype == SQL_VARYING))
        sprintf (p, "%-*s ", var->sqllen, "<null>");
    else if (dtype == SQL_DATE)
        sprintf (p, "%-20s ", "<null>");
    else
        sprintf (p, "%*s ", var->sqllen, "<null>");
    }
else
```

```
    {
    /* Print field according to its data type */
    if (dtype == SQL_SHORT)
  sprintf (p, "%6d ", *(short*) (var->sqldata));
    else if (dtype == SQL_LONG)
  sprintf (p, "%11ld ", *(long*) (var->sqldata));
    else if (dtype == SQL_FLOAT)
  sprintf (p, "%f ", *(float*) (var->sqldata));
    else if (dtype == SQL_DOUBLE)
  sprintf (p, "%f ", *(double*) (var->sqldata));
    else if (dtype == SQL_TEXT)
  sprintf (p, "%.*s ", var->sqllen, var->sqldata);
    else if (dtype == SQL_DATE)
        {
        gds_$decode_date (var->sqldata, &times);
  sprintf (d, "%2d-%s-%4d ", times.tm_mday, alpha_months
[times.tm_mon],
       times.tm_year + 1900);
        sprintf (p, "%-20s ", d);
        }
    else if (dtype == SQL_VARYING)
  {
  vary = (VARY*) var->sqldata;
        string = vary->vary_string;
        string[vary->vary_length] = 0;
  sprintf (p, "%-*s ",
      var->sqllen, vary->vary_string);
  }
    }

while (*p)
    ++p;

*s = p;
}


static print_line (sqlda)
    SQLDA  *sqlda;
{
/*************************************
 *
 *p r i n t _ l i n e
```

```
 *
 *************************************
 *
 * Functional description
 * Print a line of SQL variables.
 *
 *************************************/
char    *p, line [1024];
SQLVAR   *var, *end;

p = line;

for (var = sqlda->sqlvar, end = var + sqlda->sqld; var < end;
var++)
    print_item (&p, var);

*p++ = '\n';
*p = 0;
printf (line);
}



static process_statement (string, sqlda)
    char  *string;
    SQLDA  *sqlda;
{
/*************************************
 *
 *p r o c e s s _ s t a t e m e n t
 *
 *************************************
 *
 * Functional description
 * Prepare and execute a dynamic SQL statement.
 *
 *************************************/
long   buffer [512];
short   length, lines, alignment, type, offset;
SQLVAR  *var, *end;

EXEC SQL WHENEVER SQLERROR GO TO punt;
EXEC SQL PREPARE Q1 INTO sqlda FROM :string;
```

```
/* If the statement isn't a select, execute it and be done */

if (!sqlda->sqld)
    {
    EXEC SQL EXECUTE Q1;
    return;
    }
else if (sqlda->sqld > sqlda->sqln)
    {
    printf ("Statment not executed, can only select %d
          items\n", sqlda->sqln);
    return;
    }

/* Otherwise, open the cursor to start things up */

EXEC SQL DECLARE C CURSOR FOR Q1;
EXEC SQL OPEN C;

/* Set up SQLDA to receive data */

offset = 0;

for (var = sqlda->sqlvar, end = var + sqlda->sqld; var < end;
var++)
    {
    alignment = var->sqllen;
    type = var->sqltype & ~1;
    if (type == SQL_BLOB)
        {
        printf ("Statment not executed, cannot select blob
fields\n");
        return;
        }
    if (type == SQL_TEXT)
  alignment = 1;
    else if (type == SQL_VARYING)
  alignment = sizeof (short);
    offset = ALIGN (offset, alignment);
    var->sqldata = (char*) buffer + offset;
    offset += var->sqllen;
    offset = ALIGN (offset, sizeof (short));
```

```
        var->sqlind = (short*) ((char*) buffer + offset);
        offset += sizeof  (short);
        }

    /* Fetch and print records until EOF */

    for (lines = 0;; ++lines)
        {
        EXEC SQL FETCH C USING DESCRIPTOR sqlda;
        if (SQLCODE)
      break;
        if (!lines)
      printf ("\n");
        print_line (sqlda);
        }

    if (lines)
        printf ("\n");

    EXEC SQL CLOSE C;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    return;

    punt:
       /* printf ("Statement failed, SQLCODE = %d\n\n", SQLCODE); */
        gds_$print_status (gds_$status);
    }
```

Steps for Pascal Programs

To process unknown statements in Pascal programs:

1. Declare the output SQLDA:

```
sqlda1 : sqlda;
```

2. Construct the DSQL string by prompting the user for input and reading one character at a time:

```
(* Prompt for SQL statements and process them *)
done := FALSE;
while NOT done do
    begin
    get_statement (statement);
```

```
    if (((statement[1] = 'E') OR (statement[1] = 'e')) AND
        ((statement[2] = 'X') OR (statement[2] = 'x')) AND
        ((statement[3] = 'I') OR (statement[3] = 'i')) AND
        ((statement[4] = 'T') OR (statement[4] = 't'))) then
         begin
    COMMIT;
    FINISH;
    done := TRUE;
         end
   else if (((statement[1] = 'Q') OR (statement[1] = 'q')) AND
        ((statement[2] = 'U') OR (statement[2] = 'u')) AND
        ((statement[3] = 'I') OR (statement[3] = 'i')) AND
        ((statement[4] = 'T') OR (statement[4] = 't'))) then
         begin
    ROLLBACK;
    FINISH;
    done := TRUE;
         end
    else
        process_statement (statement, sqlda1);
    end;
end.
```

3. Prepare the DSQL string into the output SQLDA:

```
EXEC SQL PREPARE Q1 INTO sqlda1 FROM :statement;
```

4. Check to see if the SQLD field is 0. This indicates whether or not the statement is a **select** statement. If the SQLD field is 0, the statement *is not* a **select** statement and can be processed accordingly:

```
if sqlda1.sqld = 0 then
    begin
    (* If the statement isn't a select, execute it and be
        done *)
    EXEC SQL EXECUTE Q1;
    goto 999;
    end
```

If the SQLD field is not 0, the statement *is* a **select** statement. In this case, you need to follow Steps 5 through 10.

5. Check to see if the output SQLD is greater than the output SQLN. If the output SQLD *is* greater than the output SQLN, the number of select items returned is too large for the space allocated. In this case, return an error message:

```
else if sqlda1.sqld > sqlda1.sqln then
    begin
    write ('Statement not executed, can only select ');
    write (sqlda1.sqln);
    writeln (' items');
    goto 999;
    end;
```

6. Declare the cursor:

```
EXEC SQL DECLARE C CURSOR FOR Q1;
```

7. Open the cursor:

```
EXEC SQL OPEN C;
```

8. Fetch the cursor using the output SQLDA:

```
EXEC SQL FETCH C USING DESCRIPTOR sqlda1;
```

9. When you finish processing the statement, close the cursor and process the next statement:

```
EXEC SQL CLOSE C;
```

Pascal Program Example

The following example shows how to use DSQL to process unknown statements.

```
(**************************************************************
 *
 * A simple minded, generalized program to execute Dynamic SQLdq.
 *
 **************************************************************)


program sql (INPUT, OUTPUT);


DATABASE DB = COMPILETIME "atlas.gdb" RUNTIME db_name;

type
  statement_type = array [1..2000] of char;
  string_type = packed array [1..200] of char;

var
```

```
   done : boolean;
   sqlda1 : sqlda;
   statement : statement_type;
   db_name : packed array [1..128] of char;

procedure get_statement (VAR statement : statement_type);

(*************************************
 *
 *   g e t _ s t a t e m e n t
 *
 *************************************
 *
 * Functional description
 *
 *************************************)
var
   done, more : boolean;
   i, j, k : integer;
   buffer : packed array [1..200] of char;

begin

write ('SQL> ');
readln (buffer);
done := FALSE;
j := 0;
while NOT done do
   begin
         i := 200;
   more := FALSE;
         while (i > 0) AND (buffer[i] = ' ') do
      i := i - 1;
         if i <> 0 then
      begin
      if buffer[i] = '-' then
         begin
         more := TRUE;
         i := i - 1;
         end;
      for k := 1 to i do
         statement [j + k] := buffer [k];
      j := j + i;
```

```
            if NOT more then
                begin
                for k := j + 1 to size (statement) do
                    statement [k] := ' ';
                done := TRUE;
                end;
            end;
        if NOT done then
            begin
            write ('CON> ');
            readln (buffer);
            end;
        end;
    end;

    procedure print_line (VAR sqlda1: sqlda);

    (*************************************
     *
     *   p r i n t _ l i n e
     *
     *************************************
     *
     * Functional description
     *
     *************************************)
    var
      len, i, typ : integer;
      string_ptr : ^string_type;
      short_ptr : ^integer16;
      long_ptr : ^integer32;
      double_ptr : ^double;
      float_ptr : ^real;

    begin

    for i := 1 to sqlda1.sqld do begin
        short_ptr := sqlda1.sqlvars[i].sqlind;
        typ := sqlda1.sqlvars[i].sqltype - 1;
        if (short_ptr^ < 0) then
            begin
            (* If field was missing print <null> *)
            case typ of
```

```
        SQL_SHORT    : len := 8;
        SQL_LONG     : len := 12;
          SQL_FLOAT    : len := 12;
        SQL_DOUBLE   : len := 12;
        SQL_TEXT     : len := sqlda1.sqlvars[i].sqllen;
        end;

    write ('<null>');
    len := len - 6;
    if len > 0 then
        write (' ' : len);
    end
else begin
    (* Print field according to its data type *)
    case typ of
        SQL_SHORT    : begin;
                       short_ptr :=
                           sqlda1.sqlvars[i].sqldata;
                       write (short_ptr^);
                       end;
        SQL_LONG     : begin;
                       long_ptr :=
                           sqlda1.sqlvars[i].sqldata;
                       write (long_ptr^);
                       end;
          SQL_FLOAT    : begin;
                       float_ptr :=
                           sqlda1.sqlvars[i].sqldata;
                       write (float_ptr^);
                       end;
        SQL_DOUBLE   : begin;
                       double_ptr :=
                           sqlda1.sqlvars[i].sqldata;
                       write (double_ptr^);
                       end;
        SQL_TEXT     : begin;
                       len := sqlda1.sqlvars[i].sqllen;
                       string_ptr :=
                           sqlda1.sqlvars[i].sqldata;
                       write (string_ptr^ : len);
                       end;
        end;
    end;
```

```
        if i <> sqlda1.sqld then
            write (' ');
        end;
    writeln (' ');
    end;

    procedure process_statement (VAR statement : statement_type;
                    VAR sqlda1: sqlda);

    (*************************************
     *
     *   p r o c e s s _ s t a t e m e n t
     *
     *************************************
     *
     * Functional description
     *
     *************************************)
    var
      done, first : boolean;
      i, typ : integer;
      string_ptr : ^string_type;
      short_ptr :  ^integer16;
      long_ptr : ^integer32;
      double_ptr :  ^double;
      float_ptr :  ^real;
    label
        999;

    begin

    EXEC SQL WHENEVER SQLERROR GO TO 999;
    EXEC SQL PREPARE Q1 INTO sqlda1 FROM :statement;

    if sqlda1.sqld = 0 then
        begin
        (* If the statement isn't a select, execute it and be done *)
        EXEC SQL EXECUTE Q1;
        goto 999;
        end
    else if sqlda1.sqld > sqlda1.sqln then
        begin
        write ('Statment not executed, can only select ');
```

```
    write (sqlda1.sqln);
    writeln (' items');
    goto 999;
    end;

(* Otherwise, open the cursor to start things up *)
EXEC SQL DECLARE C CURSOR FOR Q1;
EXEC SQL OPEN C;

(* Set up SQLDA to receive data *)
for i := 1 to sqlda1.sqld do begin
    typ := sqlda1.sqlvars[i].sqltype - 1;
    case typ of
        SQL_SHORT   : begin;
                      new (short_ptr);
                      sqlda1.sqlvars[i].sqldata := short_ptr;
                      end;
        SQL_LONG    : begin;
                      new (long_ptr);
                      sqlda1.sqlvars[i].sqldata := long_ptr;
                      end;
        SQL_FLOAT   : begin;
                      new (float_ptr);
                      sqlda1.sqlvars[i].sqldata := float_ptr;
                      end;
        SQL_DOUBLE  : begin;
                      new (double_ptr);
                      sqlda1.sqlvars[i].sqldata := double_ptr;
                      end;
        SQL_TEXT    : begin;
                      new (string_ptr);
                      sqlda1.sqlvars[i].sqldata := string_ptr;
                      end;
        SQL_VARYING : begin;
                      sqlda1.sqlvars[i].sqltype :=
                          SQL_TEXT + 1;
                      sqlda1.sqlvars[i].sqllen :=
                          sqlda1.sqlvars[i].sqllen - 2;
                      new (string_ptr);
                      sqlda1.sqlvars[i].sqldata := string_ptr;
                      end;
        SQL_DATE    : begin;
                      sqlda1.sqlvars[i].sqltype :=
```

```
                                  SQL_TEXT + 1;
                              sqlda1.sqlvars[i].sqllen := 12;
                              new (string_ptr);
                              sqlda1.sqlvars[i].sqldata := string_ptr;
                              end;
                SQL_BLOB    : begin;
                              writeln
               ('Statment not executed, cannot select blob fields');
                              goto 999;
                              end;
                end;
            new (short_ptr);
            sqlda1.sqlvars[i].sqlind := short_ptr;
            end;

        (* Fetch and print records until EOF *)
        first := TRUE;
        done := FALSE;
        while NOT done do
            begin
            EXEC SQL FETCH C USING DESCRIPTOR sqlda1;
            if SQLCODE <> 0 then
          done := TRUE
            else
                print_line (sqlda1);
            end;

        if NOT first then
            writeln (' ');

        EXEC SQL CLOSE C;
        EXEC SQL WHENEVER SQLERROR CONTINUE;

        (* Now free the allocated variables *)
        for i := 1 to sqlda1.sqld do begin
            typ := sqlda1.sqlvars[i].sqltype - 1;
            case typ of
                SQL_SHORT   : begin;
                              short_ptr := sqlda1.sqlvars[i].sqldata;
                              dispose (short_ptr);
                              end;
                SQL_LONG    : begin;
                              long_ptr := sqlda1.sqlvars[i].sqldata;
```

```
                            dispose (long_ptr);
                            end;
            SQL_FLOAT     : begin;
                            float_ptr := sqlda1.sqlvars[i].sqldata;
                            dispose (float_ptr);
                            end;
              SQL_DOUBLE  : begin;
                            double_ptr := sqlda1.sqlvars[i].sqldata;
                            dispose (double_ptr);
                            end;
            SQL_TEXT      : begin;
                            string_ptr := sqlda1.sqlvars[i].sqldata;
                            dispose (string_ptr);
                            end;
            end;
        short_ptr := sqlda1.sqlvars[i].sqlind;
        dispose (short_ptr);
        end;

999: if (SQLCODE <> 0) then
        begin
        writeln ('Data base error, SQLCODE = ', SQLCODE);
        gds_$print_status (gds_$status);
        end;
end;

begin  (* main *)

sqlda1.sqln := 100;

(* Get the runtime database name. This is the db we'll process
   queries against *)
write('Database to query?  ');
readln(db_name);

READY;
START_TRANSACTION;

(* Prompt for SQL statements and process them *)
done := FALSE;
while NOT done do
    begin
    get_statement (statement);
```

```
 if (((statement[1] = 'E') OR (statement[1] = 'e')) AND
     ((statement[2] = 'X') OR (statement[2] = 'x')) AND
     ((statement[3] = 'I') OR (statement[3] = 'i')) AND
     ((statement[4] = 'T') OR (statement[4] = 't'))) then
      begin
COMMIT;
FINISH;
done := TRUE;
      end
 else if (((statement[1] = 'Q') OR (statement[1] = 'q')) AND
     ((statement[2] = 'U') OR (statement[2] = 'u')) AND
     ((statement[3] = 'I') OR (statement[3] = 'i')) AND
     ((statement[4] = 'T') OR (statement[4] = 't'))) then
      begin
ROLLBACK;
FINISH;
done := TRUE;
      end
 else
     process_statement (statement, sqlda1);
 end;
end.
```

Checking for Null Values

You can use DSQL to check whether a selected field contains null values.

The steps required to check for null values in C and Pascal programs are presented below. Each group of steps is followed by an example.

Steps for C Programs

To use DSQL in a C program to check whether a selected field contains null values:

1. Set up a local storage variable to hold the null indicator. This variable must be set up as a short datatype:

   ```
   short flag0, flag1, flag2, flag3;
   ```

2. Set up SQLIND to point to the local storage variable:

   ```
   output_sqlda->sqlvar[0].sqlind = &flag0;

   output_sqlda->sqlvar[1].sqlind = &flag1;

   output_sqlda->sqlvar[2].sqlind = &flag2;

   output_sqlda->sqlvar[3].sqlind = &flag3;
   ```

3. InterBase automatically sets the SQLTYPE field to the + 1 version of the field's datatype. This allows you to check for nulls. If you want to change the datatype, explicitly set SQLTYPE to the + 1 version of the changed datatype:

   ```
   output_sqlda->sqlvar[0].sqltype = SQL_TEXT + 1;

   output_sqlda->sqlvar[1].sqltype = SQL_TEXT + 1;

   output_sqlda->sqlvar[2].sqltype = SQL_LONG + 1;

   output_sqlda->sqlvar[3].sqltype = SQL_LONG + 1;
   ```

4. Check the local storage variable to see if the field value is missing:
   - If the local storage variable is greater than or equal to zero, the field value exists.
   - If the local storage variable is less than zero, the field value is missing.

```
while (SQLCODE == 0) {
        if (flag0 >= 0)
             printf ("%s",new_city);
```

```
        else
            printf("<missing>");

        if (flag1 >= 0)
            printf ("\t%s",new_state);
        else
            printf("\t<missing>");

        if (flag2 >= 0)
            printf ("\t%d",new_pop);
        else
            printf("\t<missing>");

        if (flag3 >= 0)
            printf("\t%d\n", new_altitude);
        else
            printf("\t<missing>\n");
```

C Program Example

The following program checks four fields for null values.

```
#include <stdio.h>
#include <string.h>

DATABASE atlas = FILENAME "atlas.gdb";

EXEC SQL
    INCLUDE SQLCA;


EXEC SQL
    WHENEVER SQLERROR GO TO ERR;

char *string = "SELECT CITY, STATE, POPULATION, ALTITUDE      \
      FROM CITIES WHERE STATE = ? ORDER BY CITY";
char new_city[26], new_state[5];
int new_pop, new_altitude;
short flag0, flag1, flag2, flag3;

SQLDA *input_sqlda;
SQLDA *output_sqlda;

main ()
```

```
{
    READY;
    START_TRANSACTION;

    input_sqlda = (SQLDA*) malloc (SQLDA_LENGTH (1));
    input_sqlda->sqln = 1;
    input_sqlda->sqld = 1;

    output_sqlda = (SQLDA*) malloc (SQLDA_LENGTH (4));
    output_sqlda->sqln = 4;

    input_sqlda->sqlvar[0].sqldata = new_state;
    input_sqlda->sqlvar[0].sqltype = SQL_TEXT;
    input_sqlda->sqlvar[0].sqllen = 4;

    EXEC SQL
        PREPARE QUERY INTO output_sqlda FROM :string;

    output_sqlda->sqlvar[0].sqldata = new_city;
    new_city[25] = 0;
    output_sqlda->sqlvar[0].sqlind = &flag0;
    output_sqlda->sqlvar[0].sqltype = SQL_TEXT + 1;

    output_sqlda->sqlvar[1].sqldata = new_state;
    new_state[4] = 0;
    output_sqlda->sqlvar[1].sqlind = &flag1;
    output_sqlda->sqlvar[1].sqltype = SQL_TEXT + 1;

    output_sqlda->sqlvar[2].sqldata = (char*) &new_pop;
    output_sqlda->sqlvar[2].sqlind = &flag2;
    output_sqlda->sqlvar[2].sqltype = SQL_LONG + 1;

    output_sqlda->sqlvar[3].sqldata = (char*) &new_altitude;
    output_sqlda->sqlvar[3].sqlind = &flag3;
    output_sqlda->sqlvar[3].sqltype = SQL_LONG + 1;


    printf("Enter 2 character state code: ");
    scanf("%s",new_state);
    if (strlen(new_state) == 2)
        strcat(new_state,"  ");

    EXEC SQL
```

```
        DECLARE C CURSOR FOR QUERY;

    EXEC SQL
        OPEN C USING DESCRIPTOR input_sqlda;

    EXEC SQL
        FETCH C USING DESCRIPTOR output_sqlda;

    while (SQLCODE == 0) {
        if (flag0 >= 0)
             printf ("%s",new_city);
        else
             printf("<missing>");

        if (flag1 >= 0)
             printf ("\t%s",new_state);
        else
             printf("\t<missing>");

        if (flag2 >= 0)
             printf ("\t%d",new_pop);
        else
             printf("\t<missing>");

        if (flag3 >= 0)
             printf("\t%d\n", new_altitude);
        else
             printf("\t<missing>\n");

        EXEC SQL
            FETCH C USING DESCRIPTOR output_sqlda;
    }

    EXEC SQL
        ROLLBACK RELEASE;
    exit();

ERR: printf ("Data base error, SQLCODE = %d\n", SQLCODE);
     gds_$print_status (gds_$status);

     EXEC SQL
        ROLLBACK RELEASE;
}
```

Steps for Pascal programs

To use DSQL in a Pascal program to check whether a selected field contains null values:

1. Set up a local storage variable to hold the null indicator. This variable must be set up as a 2-byte integer datatype:

   ```
   flag0, flag1, flag2, flag3 : integer16;
   ```

2. Set up SQLIND to point to the local storage variable:

   ```
   output_sqlda.sqlvars[1].sqlind  := addr(flag0);

   output_sqlda.sqlvars[2].sqlind  := addr(flag1);

   output_sqlda.sqlvars[3].sqlind  := addr(flag2);

   output_sqlda.sqlvars[4].sqlind  := addr(flag3);
   ```

3. InterBase automatically sets the SQLTYPE field to the + 1 version of the field's datatype. This allows you to check for nulls. If you want to change the datatype, explicitly set SQLTYPE to the + 1 version of the changed datatype:

   ```
   output_sqlda.sqlvars[1].sqltype := SQL_TEXT + 1;

   output_sqlda.sqlvars[2].sqltype := SQL_TEXT + 1;

   output_sqlda.sqlvars[3].sqltype := SQL_LONG + 1;

   output_sqlda.sqlvars[4].sqltype := SQL_LONG + 1;
   ```

4. Check the local storage variable to see if the field value is missing:
   - If the local storage variable is greater than or equal to zero, the field value exists.
   - If the local storage variable is less than zero, the field value is missing.

   ```
   while (SQLCODE = 0) do
      begin
          if flag0 >= 0 then
              write(new_city)
          else
              write(MISSING_STRING);

          if flag1 >= 0 then
              write(TAB_STRING, new_state)
          else
   ```

```
                write(TAB_STRING, MISSING_STRING);

        if flag2 >= 0 then
            write(TAB_STRING, new_pop)
        else
            write(TAB_STRING, MISSING_STRING);

        if flag3 >= 0 then
            writeln(TAB_STRING, new_altitude)
        else
            writeln(TAB_STRING, MISSING_STRING);
```

Pascal Example

The following program checks four fields for null values:

```
program dsql_sel_with (INPUT, OUTPUT);

EXEC SQL
    INCLUDE SQLCA;

EXEC SQL
    WHENEVER SQLERROR GO TO 999

const
    MISSING_STRING = '*';
    TAB_STRING = '        ';
var
    query : array [1..91] of char :=
       'SELECT CITY, STATE, POPULATION, ALTITUDE
              FROM CITIES WHERE STATE = ? ORDER BY CITY';
    new_state : array [1..4] of char;
    new_city : array [1..25] of char;
    new_pop : integer32;
    new_altitude : integer32;
    flag0, flag1, flag2, flag3 : integer16;
    loop : integer16;

    input_sqlda : sqlda;
    output_sqlda : sqlda;

label
   999;
```

```
begin

    READY;
    START_TRANSACTION;

    input_sqlda.sqln := 1;
    input_sqlda.sqld := 1;
    output_sqlda.sqln := 4;

    input_sqlda.sqlvars[1].sqldata := addr(new_state);
    input_sqlda.sqlvars[1].sqltype := SQL_TEXT;
    input_sqlda.sqlvars[1].sqllen := 4;

    EXEC SQL
        PREPARE QUERY INTO output_sqlda FROM :query;

    output_sqlda.sqlvars[1].sqldata := addr(new_city);
    output_sqlda.sqlvars[1].sqlind  := addr(flag0);
    output_sqlda.sqlvars[1].sqltype := SQL_TEXT + 1;

    output_sqlda.sqlvars[2].sqldata := addr(new_state);
    output_sqlda.sqlvars[2].sqlind  := addr(flag1);
    output_sqlda.sqlvars[2].sqltype := SQL_TEXT + 1;

    output_sqlda.sqlvars[3].sqldata := addr(new_pop);
    output_sqlda.sqlvars[3].sqlind  := addr(flag2);
    output_sqlda.sqlvars[3].sqltype := SQL_LONG + 1;

    output_sqlda.sqlvars[4].sqldata := addr(new_altitude);
    output_sqlda.sqlvars[4].sqlind  := addr(flag3);
    output_sqlda.sqlvars[4].sqltype := SQL_LONG + 1;

    write('Enter 2 character State code: ');
    readln(new_state);

    EXEC SQL
        DECLARE C CURSOR FOR QUERY;

    EXEC SQL
        OPEN C USING DESCRIPTOR input_sqlda;

    EXEC SQL
```

```
          FETCH C USING DESCRIPTOR output_sqlda;

      while (SQLCODE = 0) do
      begin
          if flag0 >= 0 then
              write(new_city)
          else
              write(MISSING_STRING);

          if flag1 >= 0 then
              write(TAB_STRING, new_state)
          else
              write(TAB_STRING, MISSING_STRING);

          if flag2 >= 0 then
              write(TAB_STRING, new_pop)
          else
              write(TAB_STRING, MISSING_STRING);

          if flag3 >= 0 then
              writeln(TAB_STRING, new_altitude)
          else
              writeln(TAB_STRING, MISSING_STRING);

          EXEC SQL
              FETCH C USING DESCRIPTOR output_sqlda;
      end;

      EXEC SQL
          ROLLBACK RELEASE;
      return;

  999: writeln ('Data base error, SQLCODE = ', SQLCODE);
       gds_$print_status (gds_$status);

       EXEC SQL
           ROLLBACK RELEASE;
  end.
```

For More Information

For more information on DSQL, refer to:

- Chapter 1, *Introduction to DSQL*, for an overview of DSQL
- Chapter 3, *Accessing Special Field Types*, for information on accessing blob and date fields with DSQL
- Chapter 4, *Setting up an SQLDA*, for information on setting up an SQLDA and on the structures that InterBase provides
- The chapter on SQL statements in the *Programmer's Reference*, for information on specific DSQL-related statements

Chapter 3
Accessing Special Field Types

This chapter describes how to access blob fields and date fields when using DSQL.

Overview

You access blob fields from a DSQL program by declaring the local variable using **gds_$quad** and then making one or more blob calls.

You access date fields from a DSQL program by either casting the date as a character string or converting the date field to the UNIX time structure.

Accessing Blob Fields

You can access blob fields in DSQL by using any GDML statement or **gds** call that operates on a blob id. For example, you can use the **blob_$display** and the **gds_$get_-segment** calls to access a blob.

The following considerations apply to accessing blob fields. You can't:

- Cast a blob field as a string.
- Check a blob field for missing values or set a blob field to missing values.

The steps required to access blob fields for C and Pascal programs are presented below. Each group of steps is followed by an example.

Steps for C Programs

To access blob fields in C programs:

1. Declare the local variable using **GDS_$QUAD**:

   ```
   GDS_$QUAD blob;
   ```

2. Use one or more of the permitted blob calls:

   ```
   blob_display (&blob, DB, gds_$trans, "");
   ```

C Program Example

The following program shows how to display a blob field through DSQL:

```
#include <stdio.h>

DATABASE DB = 'atlas.gdb';

EXEC SQL
    INCLUDE SQLCA;


EXEC SQL
    WHENEVER SQLERROR GO TO ERR;

SQLDA *sqlda;
GDS_$QUAD blob;
short flag;
char *query = "SELECT GUIDEBOOK FROM TOURISM                    \
   WHERE STATE = 'NY'";
```

```
main ()
{
    READY;
    START_TRANSACTION;

    sqlda = (SQLDA*) malloc (SQLDA_LENGTH (1));
    sqlda->sqln = 1;

    EXEC SQL
        PREPARE Q INTO sqlda FROM :query;

    sqlda->sqlvar[0].sqldata = (char*) &blob;
    sqlda->sqlvar[0].sqlind  = &flag;

    EXEC SQL
        DECLARE C CURSOR FOR Q;

    EXEC SQL
        OPEN C;


    EXEC SQL
        FETCH C USING DESCRIPTOR sqlda;

    if (SQLCODE)
        gds_$print_status (gds_$status);
    else
        BLOB_display (&blob, DB, gds_$trans, "");

    EXEC SQL
        COMMIT RELEASE;
    exit();
    ERR: printf ("Data base error, SQLCODE = %d\n", SQLCODE);
     gds_$print_status (gds_$status);

     EXEC SQL
        ROLLBACK RELEASE;
}
```

Steps for Pascal Programs

To access blob fields in Pascal programs:

1. Declare the local variable using **gds_$quad**:

```
blob : gds_$quad;
```

2. Use one or more of the permitted blob calls:

```
blob_$display (blob, atlas, gds_$trans, 'guidebook',
     sizeof ('guidebook'));
```

Pascal Program Example

The following program shows how to display a blob field through DSQL:

```
program dsql_blob_display (INPUT, OUTPUT);

DATABASE atlas = 'atlas.gdb';

EXEC SQL
    INCLUDE SQLCA;

EXEC SQL
    WHENEVER SQLERROR GO TO 999

var
    blob : gds_$quad;
    sqlda1 : sqlda;
    flag : integer16;
    query: array[1..61] of CHAR :=
       'SELECT GUIDEBOOK FROM TOURISM WHERE STATE = "NY"';
label
    999;

begin

    READY;
    START_TRANSACTION;

    sqlda1.sqln := 1;

    EXEC SQL
        PREPARE Q INTO sqlda1 FROM :query;
```

```
    sqlda1.sqlvars[1].sqldata := addr(blob);
    sqlda1.sqlvars[1].sqlind := addr(flag);

    EXEC SQL
        DECLARE C CURSOR FOR Q;

    EXEC SQL
        OPEN C;

    EXEC SQL
        FETCH C USING DESCRIPTOR sqlda1;

    if SQLCODE <> 0 then goto 999
    else
        blob_$display (blob, atlas, gds_$trans, 'guidebook',
              sizeof ('guidebook'));

    EXEC SQL
        COMMIT RELEASE;

    if (sqlcode <> 0 or sqlcode <> 100) begin
999: writeln ('Data base error, SQLCODE = ', SQLCODE);
     gds_$print_status (gds_$status);

     EXEC SQL
         ROLLBACK RELEASE;
    end;
end.
```

Accessing Date Fields

There are two ways to access a date field from a DSQL program. You can:

- Cast the date as a character string.
- Convert the date field to the UNIX time structure and then use **gds** routines to access the date.

These methods are described in this section.

Casting a Date Field

Instructions for casting a date field in C and Pascal are presented below.

Steps for C Programs

To cast a date field as a character string in C programs:

1. Set the SQLTYPE field to SQL_TEXT or SQL_TEXT + 1:

```
sqlda->sqlvar[1].sqltype = SQL_TEXT + 1;
```

2. Change the default value of SQLLEN to be at least 11 characters. If you want the time as well as the date returned, set the default value of SQLLEN to be 25 characters:

```
sqlda->sqlvar[1].sqldata = state_date;
    state_date[12] = 0;
    sqlda->sqlvar[1].sqllen = sizeof(state_date);
```

C Program Example

The following program shows how to access a date field through DSQL by casting it as a character string:

```
#include <stdio.h>

DATABASE atlas = FILENAME "atlas.gdb";
EXEC SQL
    INCLUDE SQLCA;

EXEC SQL
    WHENEVER SQLERROR GO TO ERR;

char *string = "SELECT STATE_NAME, STATEHOOD FROM STATES      \
```

```
    ORDER BY STATEHOOD";
SQLDA *sqlda;
char state_date[12], state_name[26];
short flag1, flag2;

main ()
{
    READY;
    START_TRANSACTION;

    sqlda = (SQLDA*) malloc (SQLDA_LENGTH (2));
    sqlda->sqln = 2;

    EXEC SQL
        PREPARE Q INTO sqlda FROM :string;

    sqlda->sqlvar[0].sqldata = state_name;
    state_name[25] = 0;
    sqlda->sqlvar[0].sqlind = &flag1;
    sqlda->sqlvar[0].sqltype = SQL_TEXT + 1;

/* Manually change the date's default size of eight chars */

    sqlda->sqlvar[1].sqldata = state_date;
    state_date[12] = 0;
    sqlda->sqlvar[1].sqllen = sizeof(state_date);
    sqlda->sqlvar[1].sqlind = &flag2;
    sqlda->sqlvar[1].sqltype = SQL_TEXT + 1;

    EXEC SQL
        DECLARE C CURSOR FOR Q;

    EXEC SQL
        OPEN C;
    EXEC SQL
        FETCH C USING DESCRIPTOR sqlda;

    while (SQLCODE == 0) {
        printf ("%s ", state_name);

        if (flag2 >= 0)
            printf("%12s\n", state_date);
        else
```

```
                printf("<date missing>\n");

            EXEC SQL
                FETCH C USING DESCRIPTOR sqlda;
        }

        EXEC SQL
            ROLLBACK RELEASE;
        return;

    ERR: printf ("Data base error, SQLCODE = %d\n", SQLCODE);
         gds_$print_status (gds_$status);

         EXEC SQL
            ROLLBACK RELEASE;
    }
```

Steps for Pascal Programs

To cast a date field as a character string in Pascal programs:

1. Set the SQLTYPE field to SQL_TEXT or SQL_TEXT + 1:

   ```
   sqlda1.sqlvars[2].sqltype := SQL_TEXT + 1;
   ```

2. Change the default value of SQLLEN to be at least 11 characters. If you want the time as well as the date returned, set the default value of SQLLEN to be 25 characters:

   ```
   sqlda1.sqlvars[2].sqllen := 11;
   ```

Pascal Program Example

The following program shows how to access a date field through DSQL by casting it as a character string. When you code the program, be sure to code the DSQL query on a single line.

```
program dsql_date_char (INPUT, OUTPUT);

DATABASE DB = 'atlas.gdb';

EXEC SQL
    INCLUDE SQLCA;

EXEC SQL
```

```
    WHENEVER SQLERROR GO TO 999

var
    query : array [1..61] of char :=
       'SELECT STATE_NAME,STATEHOOD
             FROM STATES ORDER BY STATEHOOD';
    new_state_name : array [1..25] of char;
    new_state_date : array [1..11] of char;
    flag1, flag2 : integer16;
    loop : integer16;
    sqlda1 : sqlda;

label
   999;

begin

    READY;
    START_TRANSACTION;

    sqlda1.sqln := 2;

    EXEC SQL
        PREPARE Q FROM :query;

    sqlda1.sqlvars[1].sqldata := addr(new_state_name);
    sqlda1.sqlvars[1].sqlind  := addr(flag1);
    sqlda1.sqlvars[1].sqltype := SQL_TEXT + 1;
    sqlda1.sqlvars[1].sqllen := 25;

(* Manually change the date's default size of eight chars *)

    sqlda1.sqlvars[2].sqldata := addr(new_state_date);
    sqlda1.sqlvars[2].sqlind  := addr(flag2);
    sqlda1.sqlvars[2].sqltype := SQL_TEXT + 1;
    sqlda1.sqlvars[2].sqllen := 11;

    EXEC SQL
        DECLARE C CURSOR FOR Q;

    EXEC SQL
        OPEN C;
```

```
        EXEC SQL
            FETCH C USING DESCRIPTOR sqlda1;

        while (SQLCODE = 0) do
        begin
            write(new_state_name);

            if flag2 >= 0 then
                writeln(new_state_date)
            else
                writeln('<date missing>');


            EXEC SQL
                FETCH C USING DESCRIPTOR sqlda1;
        end;

        EXEC SQL
            COMMIT RELEASE;

        if (sqlcode <> 0 and SQLCODE <> 100) begin
    999: writeln ('Data base error, SQLCODE = ', SQLCODE);
         gds_$print_status (gds_$status);


         EXEC SQL
            ROLLBACK RELEASE;
        end;
    end.
```

Using Gds Date Routines

Instructions for using **gds** date routines in C and Pascal are presented below.

Steps for C Programs

To use the **gds** date routines to access a date field in C programs:

1. Define the UNIX time structure:

   ```
   #include <sys/time.h>
   struct tm d;
   char state_date[12], state_name[26];
   ```

2. Declare the local variable using **GDS_$QUAD**:

```
GDS_$QUAD state_date;
```

3. Set the SQLTYPE field to SQL_DATE or SQL_DATE + 1:

```
sqlda->sqlvar[1].sqltype = SQL_DATE + 1;
```

4. Use the **gds_$decode_date** routine to access the date information. This routine transfers the date from the database field to the UNIX time structure:

```
if (flag2 >= 0) {
            gds_$decode_date(state_date,&d);
            printf("%2d/%02d/%4d\n",
                    d.tm_mon+1,d.tm_mday,d.tm_year+1900);
```

To store a date field, use the **gds_$encode_date** routine. This routine transfers the date from the UNIX time structure to the database field.

C Program Example

The following program shows how to access a date field through DSQL by using the **gds_$decode_date** routine:

```
#include <stdio.h>
#include <sys/time.h>

DATABASE atlas = FILENAME "atlas.gdb";

EXEC SQL
    INCLUDE SQLCA;

EXEC SQL
    WHENEVER SQLERROR GO TO ERR;

char *string = "SELECT STATE_NAME, STATEHOOD FROM STATES       \
    ORDER BY STATEHOOD";
struct tm d;
char state_name[26];
short flag1, flag2;
SQLDA *sqlda;
GDS_$QUAD state_date;

main ()
{
    READY;
```

```
START_TRANSACTION;

sqlda = (SQLDA*) malloc (SQLDA_LENGTH (2));
sqlda->sqln = 2;

EXEC SQL
    PREPARE Q INTO sqlda FROM :string;

sqlda->sqlvar[0].sqldata = state_name;
state_name[25] = 0;
sqlda->sqlvar[0].sqlind = &flag1;
sqlda->sqlvar[0].sqltype = SQL_TEXT + 1;

sqlda->sqlvar[1].sqldata = (char*) &state_date;
sqlda->sqlvar[1].sqlind = &flag2;
sqlda->sqlvar[1].sqltype = SQL_DATE + 1;

EXEC SQL
    DECLARE C CURSOR FOR Q;

EXEC SQL
    OPEN C;

EXEC SQL
    FETCH C USING DESCRIPTOR sqlda;

while (SQLCODE == 0) {
    printf ("%s ", state_name);

    if (flag2 >=0) {
        gds_$decode_date (&state_date,&d);
        printf("%2d/%02d/%4d\n",
                 d.tm_mon+1,d.tm_mday,d.tm_year+1900);
    }
    else
        printf("<date missing>\n");

    EXEC SQL
        FETCH C USING DESCRIPTOR sqlda;
}

EXEC SQL
    ROLLBACK RELEASE;
```

```
    return;

ERR: printf ("Data base error, SQLCODE = %d\n", SQLCODE);
     gds_$print_status (gds_$status);

     EXEC SQL
        ROLLBACK RELEASE;
}
```

Steps for Pascal Programs

To use the **gds** date routines to access a date field in Pascal programs:

1. Define the UNIX time structure:

   ```
   new_state_date : gds_$quad;
   d : gds_$tm;
   ```

2. Declare the local variable using **gds_$quad**:

   ```
   new_state_date : gds_$quad;
   ```

3. Set the SQLTYPE field to SQL_DATE or SQL_DATE + 1:

   ```
   sqlda1.sqlvars[2].sqltype := SQL_DATE + 1;
   ```

4. Use the **gds_$decode_date** routine to access the date information. This routine transfers the date from the database field to the UNIX time structure:

   ```
   gds_$decode_date (new_state_date,d);
   ```

   To store a date field, use the **gds_$encode_date** routine. This routine transfers the date from the UNIX time structure to the database field.

Pascal Program Example

The following program shows how to access a date field through DSQL by using the **gds_$decode_date** routine. When you code the program, be sure to code the DSQL query on a single line.

```
program dsql_date_tm (INPUT, OUTPUT);

DATABASE DB = 'atlas.gdb';

EXEC SQL
    INCLUDE SQLCA;

EXEC SQL
```

```
     WHENEVER SQLERROR GO TO 999


var
     query : array [1..61] of char := 'SELECT STATE_NAME,
            STATEHOOD FROM STATES ORDER BY STATEHOOD';
     new_state_name : array [1..25] of char;
     new_state_date : gds_$quad;
     flag1, flag2 : integer16;
     sqlda1 : sqlda;
     d : gds_$tm;


label
   999;

begin

     READY;
     START_TRANSACTION;

     sqlda1.sqln := 2;

     EXEC SQL
         PREPARE Q FROM :query;

     sqlda1.sqlvars[1].sqldata := addr(new_state_name);
     sqlda1.sqlvars[1].sqlind  := addr(flag1);
     sqlda1.sqlvars[1].sqltype := SQL_TEXT + 1;
     sqlda1.sqlvars[1].sqllen  := 25;

     sqlda1.sqlvars[2].sqldata := addr(new_state_date);
     sqlda1.sqlvars[2].sqlind  := addr(flag2);
     sqlda1.sqlvars[2].sqltype := SQL_DATE + 1;

     EXEC SQL
         DECLARE C CURSOR FOR Q;

     EXEC SQL
         OPEN C;

     EXEC SQL
         FETCH C USING DESCRIPTOR sqlda1;
```

```
    while (SQLCODE = 0) do
    begin
        write(new_state_name:25);

        if flag2 >= 0 then
         begin
            gds_$decode_date (new_state_date,d);
            writeln((d.tm_mon+1):2,'/',d.tm_mday:2,'/
',(d.tm_year+1900):4);
         end
        else
            writeln('<date missing>');

        EXEC SQL
            FETCH C USING DESCRIPTOR sqlda1;
    end;

    EXEC SQL
        COMMIT RELEASE;

    if (sqlcode <> 0 and sqlcode <> 100) begin
999: writeln ('Data base error, SQLCODE = ', SQLCODE);
     gds_$print_status (gds_$status);


     EXEC SQL
        ROLLBACK RELEASE;
    end;
end.
```

For More Information

For more information on DSQL, refer to:

- Chapter 1, *Introduction to DSQL*, for an overview of DSQL
- Chapter 2, *Processing DSQL Statements*, for information on processing DSQL statements
- Chapter 4, *Setting up an SQLDA*, for information on setting up an SQLDA and on the structures that InterBase provides
- The chapter on SQL statements in the *Programmer's Reference*, for information on specific DSQL-related statements

For more information on processing blob and date fields, refer to Chapter 8, *Using Blob Fields*, and to Chapter 10, *Using Date Fields* in the *Programmer's Guide*.

Chapter 4
Setting up an SQLDA

This chapter describes how to set up an SQLDA in C, Pascal, PL/1, Ada, COBOL, BASIC, and FORTRAN programs.

Overview

InterBase supplies an include file that declares the structure of an SQLDA for you. You can use this include file in C, Pascal, PL/1, and Ada programs, because these programs support type declarations. You can't use this file in COBOL, BASIC, and FORTRAN programs. Instead, you must set up the SQLDA yourself in these programs.

Setting up an SQLDA in C

To set up an SQLDA in C, declare an SQL communications area, which implicitly includes the definitions of DSQL structures and macros. You declare an SQL communications area by using the following statement:

```
exec sql
      include sqlca
```

This statement defines the following data structures and constants:

```
typedef struct {
  short  sqltype;
  short  sqllen;
  char   *sqldata;
  short  *sqlind;
  short  sqlname_length;
  char   sqlname [30];
} SQLVAR;

typedef struct {
  char   sqldaid [8];
  long   sqldabc;
  short  sqln;
  short  sqld;
  SQLVAR sqlvar[1];
} SQLDA;

#define SQLDA_LENGTH(n) (sizeof (SQLDA) + n * sizeof (SQLVAR))

#define SQL_TEXT =      452
#define SQL_VARYING =   448
#define SQL_SHORT =     500
#define SQL_LONG =      496
#define SQL_DOUBLE =    480
#define SQL_FLOAT =     482
#define SQL_DATE =      510
#define SQL_BLOB =      520
```

This declaration includes a macro called SQLDA_LENGTH, which determines how much memory is required to hold a specified number of variables. For examples of its use, refer to Chapters 2 and 3.

You can declare as many SQLDAs as necessary and can use the SQLDA_LENGTH macro to allocate the exact number of entries you require for each SQLDA.

To see complete examples of DSQL C programs, refer to Chapter 2, *Processing DSQL Statements*.

Setting up an SQLDA in Pascal

To set up an SQLDA in Pascal, declare an SQL communications area, which implicitly includes the definitions of DSQL structures. You declare an SQL communications area by using the following statement:

```
exec sql
      include sqlca;
```

This statement defines the following data structures and constants:

```
type
  sqlvar = record
  sqltype : integer16;
  sqllen : integer16;
  sqldata: UNIV_PTR;
  sqlind: UNIV_PTR;
  sqlname_length: integer16;
  sqlname: array [1..30] of char;
        end;

sqlda = record
  sqldaid: array [1..8] of char;
  sqldabc: integer32;
  sqln: integer16;
  sqld: integer16;
  sqlvar: array [1..100] of sqlvar;
end;

const
  SQL_TEXT= 452;
  SQL_VARYING= 448;
  SQL_SHORT= 500;
  SQL_LONG= 496;
  SQL_DOUBLE= 480;
  SQL_FLOAT= 482;
  SQL_DATE= 510;
  SQL_BLOB= 520;
```

The SQLDA record type defined by default allocates memory for either 100 parameters or 100 **select** fields. If you want your program to handle more than 100 parameters or **select** fields, you must declare a different SQLDA record type.

To see complete examples of DSQL Pascal programs, refer to Chapter 2, *Processing DSQL Statements*.

Setting up an SQLDA in PL/1

To set up an SQLDA in PL/1, declare an SQL communications area, which implicitly defines the DSQL datatypes. You declare an SQL communications area by using the following statement:

```
exec sql
      include sqlca;
```

You must also explicitly declare each sqlda using a structure like this.:

```
DECLARE 1 SQLDA BASED (SQLDAPTR),
   2 SQLAID        CHAR(8)
   2 SQLDABC       BIN FIXED(31),
   2 SQLN          BIN FIXED,
   2 SQLD          BIN FIXED,
   2 SQLVAR        (SQLSIZE REFER (SQLN)),
      3 SQLTYPE    BIN FIXED,
      3 SQLLEN     BIN FIXED,
      3 SQLDATA    PTR,
      3 SQLIND     PTR,
      3 SQLNAME    CHAR(30) VAR;

DECLARE SQLSIZE BIN FIXED;
DECLARE SQLDAPTR PTR;
```

The following definitions are included with the SQLCA:

```
%REPLACE SQL_TEXT       452;
%REPLACE SQL_VARYING    448;
%REPLACE SQL_SHORT      500;
%REPLACE SQL_LONG       496;
%REPLACE SQL_DOUBLE     480;
%REPLAE  SQL_FLOAT      482;
%REPLACE SQL_DATE       510;
%REPLACE SQL_BLOB       520;
```

In PL/1, you can declare as many SQLDAs as necessary and make them the size you require.

To see a complete example of a DSQL PL/I program, refer to the *Sample Programs* guide.

Setting up an SQLDA in Ada

To set up an SQLDA in Ada, you can declare it by including the InterBase Ada package. You still need the **include sqlca** statement to get global definitions for your program.

The InterBase Ada package defines the following data structures and constants:

```
TYPE sqlvar is RECORD
  sqltype: short_integer;
  sqllen : short_integer;
  sqldata: SYSTEM.ADDRESS;
  sqlind : SYSTEM.ADDRESS;
  sqlname_length: short_integer;
  sqlname: chars (1..30);
END RECORD;

FOR sqlvar use record at mod 2;
  sqltypeat 0 range 0..15;
  sqllen at 2 range 0..15;
  sqldataat 4 range 0..31;
  sqlind at 8 range 0..31;
  sqlname_length at 12 range 0..15;
  sqlname at 14 range 0..239;
END RECORD;

TYPE sqlvar_array IS ARRAY (short_integer range <>) of sqlvar;

SQL_TEXT:         :: CONSTANT short_integer := 452;
SQL_VARYING:      :: CONSTANT short_integer := 448;
SQL_SHORT         :: CONSTANT short_integer := 500;
SQL_LONG:         :: CONSTANT short_integer := 496;
SQL_DOUBLE:       :: CONSTANT short_integer := 480;
SQL_FLOAT         :: CONSTANT short_integer := 482;
SQL_DATE:         :: CONSTANT short_integer := 510;
SQL_BLOB:         :: CONSTANT short_integer := 520;
```

In Ada, you must specify the exact layout of the SQLDA, so that the DBMS subroutines can access it. You must specify the layout of the SQLDA record in your program.

You specify this as follows:

- Set the length of the SQLN and SQLVAR fields to the number of input parameters or select list items you want the SQLDA to hold.
- Set the position of the SQLVAR field to span bits 0 to ((n * 352) -1) where n is the number of input parameters or select list items you want the SQLDA to hold.

- Set all other lengths and positions as shown in the example below.

The Ada program below shows how to specify an SQLDA that holds three parameters or select list items. When you code the program, be sure to code the DSQL query on a single line.

```
***************************************************************
*
-- *
-- *The query to be executed is simply from a string constant
-- *In a real application the query string would be developed
-- *through interaction of some sort with the user.
-- *
***************************************************************
*

WITH basic_io, interbase;

PROCEDURE dsql IS

EXEC SQL
  INCLUDE SQLCA

-- *  Set up the query statement
query: CONSTANT string(1..61) :=
   "SELECT CITY, STATE, POPULATION FROM CITIES
          WHERE STATE = 'NY'";

-- *  Declare variables to receive data. Note: if you don't
-- *  known before hand the nature of the fields to be
-- *  retrieved, you must allocate buffer space for the fields
-- *  at run time, rather than use predeclared fields.


TYPE sqlda IS RECORD
  sqldaid    : interbase.chars (1..8);
  sqldabc    : integer;
  sqln       : short_integer := 3;
  sqld       : short_integer;
  sqlvar     : interbase.sqlvar_array (1..3);
END RECORD;
FOR sqlda use record at mod 2;
  sqldaid    at 0 range 0..63;
  sqldabc    at 8 range 0..31;
```

```
   sqln       at 12 range 0..15;
   sqld       at 14 range 0..15;
   sqlvar     at 16 range 0..1055;
END RECORD;

sqlda1: sqlda;

city: BASED_ON CITIES.CITY;
state: BASED_ON CITIES.STATE;
population: BASED_ON CITIES.POPULATION;
flag1, flag2, flag3: interbase.isc_short;

begin

EXEC SQL
   WHENEVER SQLERROR GO TO Error_processing;

-- *  Prepare the query
    EXEC SQL
        PREPARE Q INTO sqlda1 FROM :query;

-- *  Set up SQLDA to point to declared fields, adjusting
-- *  datatypes and setting null terminators on the way
-- *  through. Change the type for city & state fields to be
-- *  TEXT rather than VARYING, since C does not know
-- *  about VARYING fields. The format <datatype> + 1
-- *  indicates that the field is of <datatype> and has
-- *  an indicator variable.
    sqlda1.sqlvar(1).sqldata := city'address;
    sqlda1.sqlvar(1).sqlind := flag1'address;
    sqlda1.sqlvar(1).sqltype := interbase.SQL_TEXT + 1;
    sqlda1.sqlvar(2).sqldata := state'address;
    sqlda1.sqlvar(2).sqlind := flag2'address;
    sqlda1.sqlvar(2).sqltype := interbase.SQL_TEXT + 1;
    sqlda1.sqlvar(3).sqldata := population'address;
    sqlda1.sqlvar(3).sqlind := flag3'address;

    EXEC SQL
        DECLARE C CURSOR FOR Q;

    EXEC SQL
        OPEN C;
```

```
-- *  Fetch and print out records
     EXEC SQL
         FETCH C USING DESCRIPTOR sqlda1;
     WHILE SQLCODE = 0 LOOP
   basic_io.put (city);
   basic_io.put (state);
   basic_io.put (population);
   basic_io.new_line;
        EXEC SQL
         FETCH C USING DESCRIPTOR sqlda1;
   END LOOP;

     EXEC SQL
   COMMIT RELEASE;

<<Error_processing>>
     IF SQLCODE /= 0 THEN
   basic_io.put ("Data base error, SQLCODE = ");
   basic_io.put (SQLCODE);
   basic_io.new_line;
     END IF;

END dsql;
```

Setting up an SQLDA in BASIC

To set up an SQLDA in BASIC, you must build the SQLDA as a record. This example shows how to build an SQLDA for two parameters or **select** fields:

```
RECORD SQLDA
      STRING SQLDAID = 8
      LONG SQLDABC
      WORD SQLN
      WORD SQLD
      GROUP SQLVAR (2)
          WORD SQLTYPE
          WORD SQLLEN
          LONG SQLDATA
          LONG SQLIND
          WORD SQLNAME_LENGTH
          STRING SQLNAME = 30
            END GROUP SQLVAR
  END RECORD SQLDA
  DECLARE SQLDA SQLDA1
```

In BASIC, you can set up as many SQLDAs as your program needs.

To see a complete example of a DSQL BASIC program, refer to the *Sample Programs* guide.

Setting up an SQLDA in COBOL

To set up an SQLDA in COBOL, you must allocate space in working storage. This example shows how to allocate an SQLDA for three parameters or **select** fields:

```
01  SQLDA.
    02 SQLDAID PIC X(8).
    02 SQLDABC PIC S9(9) USAGE IS COMP.
    02 SQLN PIC S9(4) USAGE IS COMP VALUE IS 3.
    02 SQLD PIC S9(4) USAGE IS COMP.
    02 SQLVAR OCCURS 3 TIMES.
       03 SQLTYPE PIC S9(4) USAGE IS COMP.
       03 SQLLEN PIC S9(4) USAGE IS COMP.
       03 SQLDATA PIC S9(9) USAGE IS COMP.
       03 SQLIND PIC S9(9) USAGE IS COMP.
       03 SQLNAME_LENGTH PIC S9(4) USAGE IS COMP.
       03 SQLNAME PIC X(30).
```

In COBOL, you can set up as many SQLDAs as your program needs.

To see a complete example of a DSQL COBOL program, refer to the chapter on DSQL examples in the *Sample Programs* guide.

Setting up an SQLDA in FORTRAN

To set up an SQLDA in FORTRAN, you must build the SQLDA as a set of integers and character strings. You then need to position these integers and character strings appropriately, so that they map into the SQLDA structure.

There are two ways to position these elements:

- Use a **common** statement.
- Use a set of **equivalence** statements.

The example below shows how to set up an SQLDA for three variables by using a **common** statement:

```
integer*2 sqlda(1)
      character*8 sqldaid
      integer*4 sqldabc
      integer*2 sqln, sqld
      integer*2 sqlvar_type1, sqlvar_type2, sqlvar_type3
      integer*2 sqlvar_len1, sqlvar_len2, sqlvar_len3
      integer*4 sqlvar_data1, sqlvar_data2, sqlvar_data3
      integer*4 sqlvar_ind1, sqlvar_ind2, sqlvar_ind3
      integer*2 sqlvar_namelen1, sqlvar_namelen2,
sqlvar_namelen3
      character*30 sqlvar_name1, sqlvar_name2, sqlvar_name3
      common /sqldacmn/ sqldaid, sqldabc, sqln, sqld,
      *    sqlvar_type1, sqlvar_len1, sqlvar_data1, sqlvar_ind1,
      *    sqlvar_namelen1, sqlvar_name1,
      *    sqlvar_type2, sqlvar_len2, sqlvar_data2, sqlvar_ind2,
      *    sqlvar_namelen2, sqlvar_name2,
      *    sqlvar_type3, sqlvar_len3, sqlvar_data3, sqlvar_ind3,
      *    sqlvar_namelen3, sqlvar_name3
      equivalence (sqlda, sqldaid)
```

In FORTRAN, you can set up as many SQLDAs as your program needs.

To see a complete example of a DSQL FORTRAN program, refer to the *Sample Programs* guide.

For More Information

For more information on DSQL, refer to:

- Chapter 1, *Introduction to DSQL*, for an overview of DSQL
- Chapter 2, *Processing DSQL Statements*, for information on processing DSQL statements
- Chapter 3, *Accessing Special Field Types*, for information on accessing blob and date fields with DSQL
- The chapter on DSQL examples in the *Sample Programs* guide and Chapter 2, *Processing DSQL Statements*, for sample DSQL programs
- The chapter on SQL statements in the *Programmer's Reference*, for information on specific DSQL-related statements

A

Accessing
- blob fields 3-1
- date field with DSQL 3-1, 3-6

Ada
- DSQL example 4-7
- SQLDA setup 4-6

B

BASIC
- DSQL considerations 1-8
- SQLDA setup 4-10

Blob
- accessing 3-1

C

C
- DSQL considerations 1-8
- missing values in DSQL 2-53
- null indicator, setting up local storage for DSQL in 2-53
- null values in DSQL 2-53
- setting up the SQLDA 2-6, 2-13, 2-21, 2-30, 4-2
- SQLDA setup 4-2

Casting
- date fields 3-6

COBOL
- DSQL considerations 1-8
- SQLDA setup 4-11

D

Database
- declaring in DSQL 1-8

Datatype
- SQLDA 1-4
- SQLTYPE fields 1-7

Date field
- accessing with DSQL 3-1, 3-6
- accessing with **gds** date routines 3-10
- casting 3-6

declare cursor
- DSQL 2-2

Declaring a database in DSQL 1-8

describe
- DSQL 2-2

DSQL
- Ada example 4-7
- BASIC programming considerations 1-8
- C programming considerations 1-8
- casting date fields 3-6
- COBOL programming considerations 1-8
- compared to embedded SQL 1-3
- date fields 3-6
- **declare cursor** 2-2
- declaring a database in 1-8
- **describe** 2-2
- **execute** 2-2
- **execute immediate** 2-2
- **fetch** 2-2
- FORTRAN programming considerations 1-8–1-9
- **gds** date routines 3-10
- general information 1-1
- invalid statements 1-2
- missing values 2-53
- multiple databases 1-8
- null values 2-53
- **open** 2-2
- **open cursor** 2-2
- overview 1-1
- **prepare** 2-2
- processing non-select statements 2-3–2-12
- processing **select** statements with the SQLDA 2-13–2-29
- processing statements with parameters 2-6–2-12
- processing unknown statements 2-30–2-52
- statement definition 1-2

statements 1-2, 2-2
valid statements 1-2
Dynamic SQL, see DSQL

E

Embedded SQL
advantage of using instead of DSQL 1-1
compared to DSQL 1-3
execute
DSQL 2-2
execute immediate
DSQL 2-2
non-select statements 2-3

F

fetch
DSQL 2-2
fields
SQLDA 1-5
FORTRAN
DSQL considerations 1-8–1-9
setting up the SQLDA in 1-8, 4-12
SQLDA setup 4-12

G

gds
date routines 3-10
gds_$decode_date 3-11
gds_$encode_date 3-11
gds_$quad 3-11

I

include sqlca in Ada 4-6

M

Missing values
checking for in DSQL 2-53
DSQL 2-53
Multiple databases
DSQL 1-8

N

Non-select statement processing in DSQL 2-3
Null values
checking for in DSQL 2-53
DSQL 2-53

O

open cursor, DSQL 2-2
open, DSQL 2-2

P

Pascal
missing values in DSQL 2-57
null indicator, setting up local storage for DSQL in 2-57
null values in DSQL 2-57
setting up the SQLDA 2-9, 2-17, 2-26, 2-42, 4-4
SQLDA setup 4-4
PL/1
SQLDA setup 4-5
prepare
DSQL 2-2
Processing non-select statements
DSQL 2-3–2-12
Processing **select** statements with the SQLDA, DSQL 2-13–2-29

S

SQL descriptor area, see SQLDA
SQL, dynamic, see DSQL
SQLABC field 1-5
SQLAID field 1-5
SQLD
field 1-5
setting up in Pascal 2-26
SQLDA
fields 1-4
function of 1-4
introduction 1-4
overview 1-4

purpose of 1-4
setting up in Ada 4-6
setting up in BASIC 4-10
setting up in C 2-6, 2-13, 2-21, 2-30, 4-2
setting up in COBOL 4-11
setting up in FORTRAN 1-8, 4-12
setting up in Pascal 2-9, 2-17, 2-42, 4-4
setting up in PL/1 4-5
SQLABC field 1-5
SQLAID field 1-5
SQLD field 1-5
SQLN field 1-5
SQLVAR field 1-5
SQLVAR subfields 1-6
subfields 1-4
valid datatypes 1-4
SQLDATA subfield 1-6
SQLIND subfield 1-6
SQLLEN subfield 1-6
SQLN field 1-5
SQLNAME subfield 1-6
SQLTYPE
subfield 1-6
subfield datatypes 1-7
valid datatypes 1-7
SQLVAR
field 1-5
SQLDATA subfield 1-6
SQLIND subfield 1-6
SQLLEN subfield 1-6
SQLNAME subfield 1-6
SQLTYPE subfield 1-6
subfields 1-6

InterBase
DDL Reference

Disclaimer

Borland International, Inc. (henceforth, Borland) reserves the right to make changes in specifications and other information contained in this publication without prior notice. The reader should, in all cases, consult Borland to determine whether or not any such changes have been made.

The terms and conditions governing the licensing of InterBase software consist solely of those set forth in the written contracts between Borland and its customers. No representation or other affirmation of fact contained in this publication including, but not limited to, statements regarding capacity, response-time performance, suitability for use, or performance of products described herein shall be deemed to be a warranty by Borland for any purpose, or give rise to any liability by Borland whatsoever.

In no event shall Borland be liable for any incidental, indirect, special, or consequential damages whatsoever (including but not limited to lost profits) arising out of or relating to this publication or the information contained in it, even if Borland has been advised, knew, or should have known of the possibility of such damages.

The software programs described in this document are confidential information and proprietary products of Borland.

Restricted Rights Legend. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subdivision (b) (3) (ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013.

© **Copyright 1993** by Borland International, Inc. All Rights Reserved. InterBase, GDML, and Pictor are trademarks of Borland International, Inc. All other trademarks are the property of their respective owners.

Corporate Headquarters: Borland International Inc., 100 Borland Way, P. O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-5300. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom.

Software Version: V3.0

Current Printing: October 1993
Documentation Version: v3.0.1

Reprint note

This documentation is a reprint of InterBase V3.0 documentation. It contains most of the information from *InterBase Previous Versions Documentation Corrections* and *InterBase Version* 3.2 *Documentation Corrections* and a new index. For information on features added since InterBase Version V3.0, consult the appropriate release notes.

Table Of Contents

Preface

Who Should Read This Book vii
Using this Book viii
Text Conventions ix
Syntax Conventions x
InterBase Documentation xi

1 Introduction

Overview 1-1

2 Gdef Syntax

Overview 2-1
Gdef 2-2

3 DDL Reserved Words

Overview 3-1
Reserved Words 3-2

4 DDL Expressions

Overview 4-1
Arithmetic Expression 4-2
Boolean Expression 4-3
Database Field Expression 4-9
First Expression 4-10
Null Expression 4-11
Numeric Literal Expression 4-12
Quoted String Expression 4-13
Record Selection Expression (RSE) 4-14

Statistical Expression 4-19
Username Expression 4-20
Value Expression 4-21

5 DDL Statements

Overview 5-1
Database 5-2
Define Field 5-6
Define Filter 5-22
Define Function 5-25
Define Index 5-28
Define Relation 5-31
Define Security_Class 5-37
Define Shadow 5-41
Define Trigger 5-43
Define View 5-49
Delete 5-54
Delete Shadow 5-58
Modify Database 5-59
Modify Field 5-62
Modify Index 5-65
Modify Relation 5-67
Modify Trigger 5-71
Modify View 5-74

A Reporting and Handling Errors

Overview A-1
Error Sources A-1
Error Format A-2
Error Messages A-3
Invoking gdef A-3
Other Messages A-3

Preface

This book contains information on the statements and expressions of InterBase's data definition language.

Who Should Read This Book

You should read the *DDL Reference* manual if you are a database architect who plans to design and define InterBase databases. This book is a companion to the *Data Definition Guide* and assumes you have already read that book, or that you are experienced with InterBase.

Using this Book

This book contains the following chapters:

Chapter 1	Introduces the book.
Chapter 2	Describes the options and syntax for the data definition compiler **gdef**.
Chapter 3	Lists **gdef**'s reserved words.
Chapter 4	Contains entries for each of the InterBase DDL expressions.
Chapter 5	Contains entries for each of the InterBase DDL statements and clauses.
Appendix A	Discusses error handling and lists error messages.

Text Conventions

This book uses the following text conventions.

boldface

Indicates a command, option, statement, or utility. For example:

- Use the **commit** command to save your changes.
- Use the **sort** option to specify record return order.
- The **case_menu** statement displays a menu in the forms window.
- Use **gdef** to extract a data definition.

italic

Indicates chapter and manual titles; identifies file-names and pathnames. Also used for emphasis, or to introduce new terms. For example:

- See the introduction to SQL in the *Programmer's Guide*.
- /usr/interbase/lock_header
- Subscripts in RSE references *must* be closed by parentheses and separated by commas.
- C permits only *zero-based* array subscript references.

`fixed width font`

Indicates user-supplied values and example code:

- $run sys$system:iscinstall
- add field population_1950 long

UPPER CASE

Indicates relation names and field names:

- Secure the RDB$SECURITY_CLASSES system relation.
- Define a missing value of *X* for the LATITUDE_ COMPASS field.

Syntax Conventions

This book uses the following syntax conventions.

{braces}	Indicates an alternative item: • option::= {**vertical** | **horizontal** | **transparent**}
[brackets]	Indicates an optional item: • dbfield-expression[**not**]**missing**
`fixed width font`	Indicates user-supplied values and example code: • $run sys$system:iscinstall • add field population_1950 long
commalist	Indicates that the preceding word can be repeated to create an expression of one or more words, with each word pair separated by one comma and one or more spaces. For example, field_def-commalist resolves to: `field_def[,field_def[,field_def]...]`
italics	Indicates a syntax variable: **`create_blob`** *`blob-variable`* **`in`** *`dbfield-expression`*
|	Separates items in a list of choices.
⇓	Indicates that parts of a program or statement have been omitted.

InterBase Documentation

The InterBase Version 3.0 documentation set contains the following books:

Getting Started with InterBase (INT0032WW2179A) provides an overview of InterBase components and interfaces.

Database Operations (INT0032WW2178D) describes how to use InterBase utilities to maintain databases.

Data Definition Guide (INT0032WW2178F) describes how to create and modify InterBase databases.

DDL Reference (INT0032WW2178E) describes the function and syntax for each of the data definition language clauses and statements. It also lists the standard error messages for **gdef**.

DSQL Programmer's Guide (INT0032WW2179C) describes how to program with DSQL, a capability for accepting or generating SQL statements at runtime.

Forms Guide (INT0032WW2178A) describes how to create forms using the InterBase forms editor, **fred,** and how to use forms in **qli** and GDML applications.

Programmer's Guide (INT0032WW2178I) describes how to program with GDML, a relational data manipulation language, and SQL, an industry standard language.

Programmer's Reference (INT0032WW2178H) describes the function and syntax for each of the GDML and InterBase supported SQL clauses and statements. It also lists the standard error messages for **gpre**.

Qli Guide (INT0032WW2178C) describes the use of **qli**, the InterBase query language interpreter that allows you to read to and write from the database using interactive GDML or SQL statements.

Qli Reference (INT0032WW2178B) describes the function and syntax for each of the data definition, GDML, and SQL clauses and statements that you can use in **qli.**

Sample Programs (INT0032WW2178G) contains sample programs that show the use of InterBase features.

Master Index (INT0032WW2179B) contains index entries for the entire InterBase Version 3.0 documentation set.

Chapter 1
Introduction

This chapter describes how this manual is organized.

Overview

The *DDL Reference* contains information about InterBase DDL reserved words, expressions, statements and clauses.

Each expression, statement and clause has the following sections:

- Function, which describes what the statement or expression does.
- Syntax, which provides a complete diagram of the statement or expression and its options.
- Options, which describes each option of the statement or expression.
- Example, which shows how to use the statement or expression in a program.
- Troubleshooting, which lists error messages and suggests corrective actions.

- See Also, which refers you to related statements or expressions, or other sources of related information.

In addition, some statements and expressions contain a section describing usage. The Usage section is an in depth discussion of how or why to use a *DDL* statement or expression.

Chapter 2
Gdef Syntax

This chapter contains the syntax for the data definition language compiler, **gdef**.

Overview

Gdef is InterBase's data definition language compiler. There are a number of command line options with **gdef**. These options are described on the following pages.

Gdef

Function

Gdef is the data definition language compiler. **Gdef** can process data definition statements from a source file or from interactive input. You can use **gdef** to:

- Create a database
- Modify objects in an existing database
- Extract metadata from an existing database into a file in order to create another database

Syntax

```
gdef [options][filespec]
```

Options

`e[xtract]`
Generates DDL from an existing database file.

`r[eplace]`
Overwrites an existing file with the same name as the one specified in the **define database** statement. (On an operating system with file versions, such as VMS, **gdef** always creates a new version of the file.)

`d[ynamic]`
Generates dynamic DDL (DYN) from a source file. Used in conjunction with language options listed below. When you use this option, you also specify an output file. See the chapter on modifying metadata with dynamic DDL in the *Data Definition Guide*.

`z`
Prints **gdef** software version number.

`t`
Prints tokens as they are read by **gdef**. This option can be helpful in debugging if error messages are not enough.

`c`
Language option for C used with the **dynamic** option. This language option causes **gdef** to generate DYN commands in C format.

`f`
: Language option for FORTRAN used with the **dynamic** option. This language option causes **gdef** to generate DYN commands in FORTRAN format.

`p`
: Language option for Pascal used with the **dynamic** option. This language option causes **gdef** to generate DYN commands in Pascal format.

`cob`
: Language option for COBOL used with the **dynamic** option. This language option causes **gdef** to generate DYN commands in COBOL format.

`ansi`
: Language option for ANSI used with the **dynamic** option. This language option causes **gdef** to generate DYN commands in ANSI COBOL format.

`bas`
: Language option for BASIC used with the **dynamic** option. This language option causes **gdef** to generate DYN commands in BASIC format.

`pli`
: Language option for PL/1used with the **dynamic** option. This language option causes **gdef** to generate DYN commands in PL/1 format.

`ada`
: Language option for Ada used with the **dynamic** option. This language option causes **gdef** to generate DYN commands in Ada format.

Example

The following example shows how to generate DYN commands in C format:

O/S	Command
UNIX	% gdef -d dyn.dat.c -c new_relation.gdl
VMS	$ gdef/dynamic dyn.dat.c /c new_relation.gdl
Apollo	% gdef -d dyn.dat.c -c new_relation.gdl

Troubleshooting

See Appendix A for a discussion of errors and error handling.

You may encounter the following messages when you invoke gdef:

- *%DLP-W-IVVERB, unrecognized command verb - check validity and spelling \gdef\.*

  This error occurs only on VMS. You have not included *sys$-system:gds_login.com* in your login file to define the InterBase utility names.

- *"gdef" - name not found (O/S naming server)*

  This error occurs on Apollo. **Gdef** does not show up anywhere along your shell's command search list. Check the search path, correct it if necessary, and try again.

- *gdef: not found*

  This error occurs on UNIX. **Gdef** does not show up anywhere along your shell's command search list. Check the search path, correct it if necessary, and try again.

- *operating system directive failed*
  -no active servers (library/MBX manager)
  -communication error with journal "journal_-directory_name"

  Although a journal has been defined for the database, there is no active journal server. Use **gltj** to start the server.

See Also

See the chapters on creating a database and modifying metadata with dynamic DDL in the *Data Definition Guide*.

Chapter 3
DDL Reserved Words

This chapter lists the reserved words for **gdef**, Interbase's data definition precompiler.

Overview

Reserved words are words defined in DDL for a special purpose. They cannot be used as user-declared identifiers.

Reserved Words

Gdef recognizes the following reserved words.Reserved words marked with an asterisk must *not* be used as relation or field names.

ABORT
ALP
ACTIVE
ADD *
AND
ANY
ASC
ASLPNDING
ASTERISK
AVERAGE *
BASED
BDSIN *
BETWEEN
BLOB
BLR
BY
CHAR
COMPUTED
CONTAINING
COUNT
CROSS
DATABASE *
DATE
DEFAULT_VALUE
DEFINE *
DELETE *
DESC
DESCENDING
DOUBLE
DROP *
DUPLICATE
EDIT_STRING
ELSE
END
END_FOR
END_MODIFY
END_STORE
END_TRIGGER
EQ
ERASE
DETERNAL_FILE
FIELD *
FILE
FIRST *
FIXED
FLOAT
FOR *
FROM
GE
GROUP
GT
IF
IN
INACTIVE
INDDE
IS
LE
LENGTH
LONG
LT
MATCHES
MATCHING
MAX *
MIN *
MINUS
MISSING
MODIFY *
NE
NOT
NULL
OF
ON
OR
OVER
PAGE
PAGES
PAGE_NUMBER
PAGE_SIZE
PERLPNT
POSITION
QUAD
QUERY_HEADER
QUERY_NAME
REDULPD
RELATION *
SCALE
SECURITY_CLASS *
SDSMENT_LENGTH
SHORT
SORTED
STARTING
STARTS
STORE *
SUB_TYPE
TDET
THEN
TO
TOTAL *
TRIGGER *
UNIQUE *
USER
USING
VALID_IF
VALUE
VARYING
VIEW *
WITH *

Chapter 4
DDL Expressions

This chapter contains entries for each of the InterBase supported DDL expressions.

Overview

Interbase supports the following expressions:

Arithmetic expression
Boolean expression
Database field expression
First expression
Null expression
Numeric literal expression
Quoted string expression
Record selection expression
Statistical expression
Username expression
Value expression

The remainder of this chapter describes each expression in detail.

Arithmetic Expression

Function

The **arithmetic expression** combines value expressions and arithmetic operators.

You can add (+), subtract (-), multiply (*), and divide (/) value expressions in assignment statements. Arithmetic operators are evaluated in the normal order. Use parentheses to change the order of evaluation.

You can use the concatenation operator (|) to combine field values in record selection expressions.

Syntax

```
value-expression-1{+|-|*|/||} value-expression-2
```

Example

The following relation definition includes arithmetic value expressions that calculate population density by dividing a state's census figure by its area:

```
define view population_density of p in populations
   cross s in states over state
   p.state,
   density_1950 computed by
      (p.census_1950 / s.area),
   density_1960 computed by
      (p.census_1960 / s.area),
   density_1970 computed by
      (p.census_1970 / s.area),
   density_1980 computed by
      (p.census_1980 / s.area);
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the entry in this chapter for:

- Record selection expression
- Value expression

Boolean Expression

Function

A **Boolean expression** evaluates to true, false, or missing. It describes the characteristics of a single value expression (for example, a missing value) or the relationship between two value expressions (for example, *x* is greater than *y*). Boolean expressions are used in validation expressions, views, and triggers.

The order of precedence for evaluating compound Boolean expressions is **not**, **and** (or **&&**), and **or** (or **||**).

Syntax

```
boolean-expression ::=
{[not]conditional-expression|
conditional-expression and conditional-
expression|conditional-expression or
conditional-expression}

conditional-expression ::=
{any|between|comparison|containing|missing|
matching|starting with|unique}
```

The following sections describe the conditions of the **Boolean expression**:

- Any condition
- Between condition
- Comparison condition
- Containing condition
- Matching condition
- Missing or null condition
- Starting with condition
- Unique condition

Any Condition

Function

The **any** condition tests for the existence of at least one qualifying record in a relation or relations. This expression is true if the

record stream specified by the RSE includes at least one record. If you add **not**, the expression is true if there are *no* records in the record stream.

Use **any** instead of joining records if all you want to do is establish that a record exists. As soon as InterBase finds one record that meets the search criteria, it stops. In a join, InterBase continues until it finds all qualifying records.

Syntax

```
[not]any rse
```

Example

The following trigger definition uses an **any** condition to check for the existence of a city in the CITIES relation. It checks for this whenever a new record is stored in the CROSS_COUNTRY relation:

```
define trigger check_and_store_cities for
cross_country
post store:
   if not any c in cities
      with c.city = new.city
      and c.state = new.state
   store x in cities
      x.city = new.city;
      x.state = new.state;
   end_store;
end_trigger;
```

Between Condition

Function

The **between** condition tests whether a value expression occurs between two other value expressions. This test includes the boundary values.

Syntax

```
value-expression-1 [not] {between | bt}
value-expression-2 and value-expression-3
```

Example

The following view includes only those cities at the specified coordinates:

```
define view middle_america of c in cities with
   c.longitude_degrees between 79 and 104 and
   c.latitude_degrees between 33 and 42
```

```
c.city,
c.state,
c.altitude;
```

Comparison Condition

Function

The **comparison** condition describes the characteristics of a single expression.

Syntax

```
value-expression-1 relational-operator value-
expression-2
```

relational-operator
One of the operators in the following table:

Operator	Relationship
eq *or* = *or* ==	Equal
ne *or* <> *or* !=	Not equal
gt *or* >	Greater than
ge *or* >=	Greater than or equal
lt *or* <	Less than
le *or* <=	Less than or equal

Example

The following view definition displays only the names of states in which the capital is not the largest city:

```
define view small_capital_city of
      s in states cross c in cities over state
      cross cs in cities with cs.state = c.state
      and cs.city = s.capital and cs.population>
      c.population
   reduced to s.state, s.capital;
```

Containing Condition

Function

The **containing** condition conducts a case-insensitive search for the presence of a substring anywhere in the value expression. It evaluates to true if the substring is contained in the expression. If the value of the value expression is missing, the result is missing.

The **containing** condition also works with blobs, searching every segment in a blob for an occurrence of the quoted string.

Syntax

```
value-expression-1 [not]{containing|ct|cont}
value-expression-2
```

Example

The following view includes only those CITIES records that contain “ville” somewhere in their name:

```
define view villes of c in cities
 with c.city containing 'ville'
 c.city, c.state, c.population;
```

The following view includes only those CROSS_COUNTRY records that contain the substring “varied” in the COMMENTS field:

```
define view varied_xc of c in cross_country
 with c.comments containing 'varied'
 c.area_name, c.state, c.comments;
```

Matching Condition

Function

The **matching** condition conducts a case-insensitive search for the presence of a wildcarded substring. This substring can contain the wildcard characters * and ?. The asterisk matches an unspecified run of characters, while the question mark matches a single character.

Syntax

```
value-expression-1[not]matching value-
expression-2
```

Example

The following view includes only those CITIES records containing the substring “ton” in the CITY field:

```
define view city_ton of c in cities
 with c.city matching '*ton*'
 c.city, c.state, c.population;
```

Missing Condition

Function

The **missing** condition tests for the absence of a value in a database field expression. It is true if the value of the database field is missing.

Syntax

dbfield-expression [**not**] **missing**

Example

The following view includes only those cities with no value stored for several fields:

```
define view city_cleanup of c in cities with
  c.latitude_degrees missing or
  c.longitude_degrees missing or
  c.population missing
c.city,
  c.state;
```

Starting With Condition

Function

The **starting with** condition conducts a case-sensitive search for the presence of a substring at the beginning of a value expression. It evaluates to true if the first characters of the value expression match the substring.

Syntax

value-expression-1[**not**]**starting with** *value-expression-2*

Example

The following view includes only those STATES records containing "New" at the beginning of the STATE_NAME field:

```
define view new_states of s in states
 with s.state_name starting with 'New'
 s.state_name,
 s.capital;
```

Unique Condition

Function

The **unique** condition tests for the existence of exactly one qualifying record. This expression is true if the record stream specified by RSE consists of only one record.

If you add **not**, the condition is true if there is more than one record in the record stream or if the record stream is empty. For instance, if you want states with more than one baseball team and you specify **not unique**, you get states without baseball teams as well.

Note

Gdef does not support the **over** clause of the RSE with the **unique** condition. Therefore, you must use fully qualified terms when using this condition in a join operation. The example below demonstrates this usage.

Syntax

```
[not]unique rse
```

Example

The following view includes only those states with a single record in the CITIES relation:

```
define view lazy_data_entry_person of s in states
 with unique c in cities with c.state = s.state
 s.state_name;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the entries in this chapter for:

- Value expression
- Record selection expression
- Predicate

Database Field Expression

Function

The **dbfield** expression references database fields. This expression can occur in several clauses of the RSE and Boolean expression.

Syntax

```
[context-variable.]field-name
```

context-variable
Qualifies the database field for multi-relation operations and for use in views. You must declare a context variable for a relation in the *relation-clause* of the record selection expression.

Example

The following view definition uses database field expressions in the view field list:

```
define view geo_cities of c in cities
   c.city,
   c.state,
   c.altitude,
   c.latitude,
   c.longitude;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the entries in this chapter for:

- Record selection expression
- Boolean expression

First Expression

Function

The **first** expression (**first...from**) forms a record stream and evaluates an expression. InterBase finds the first qualifying record in the record stream. If the stream is empty, it returns an error unless you supply an **else** clause. Otherwise, InterBase evaluates *value-expression-1* in the context of the record it found. The result of the evaluation is returned as the value of *first-expression* or the value specified in the **else** clause.

If you use the *first-expression* in a **print** command, you must enclose it in parentheses. Otherwise, **gdef** assumes you are using the *first-clause* of the record selection expression.

Syntax

```
first value-expression-1 from rse
```

Example

The following trigger definition includes a *first-expression* in an assignment statement:

```
define trigger check_states_and_cities for
big_cities
pre store:
   begin
      if not any s in states with
      s.state_name = new.state_name
         then abort 1;
      else if not any c in cities with
         new.city = c.city and
         c.state = first s.state from
         s in states with
         s.state_name = new.state_name
      then store c in cities using
         c.city = new.city;
         c.population = new.population;
         c.state = first s.state from
         s in states with
         s.state_name = new.state_name;
      end_store;
end;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the entry in this chapter for record selection expression.

Null Expression

Function

The **null** expression lets you assign a value of null to a database field. This expression is equivalent to a SQL **insert** statement's null assignment.

Syntax

```
null
```

Example

The following trigger definition includes an assignment of **null**:

```
define trigger cleanup_after_erase for big_cities
post erase:
   begin
   for st in states with
      st.capital = old.city and
      st.state = old.state
      abort 2;
   end_for;
   for sa in ski_areas with
      sa.city = old.city and sa.state = old.state
      modify sa using
         sa.city = null;
      end_modify;
   end_for;
   end;
end_trigger;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the chapter on defining fields in the *Data Definition Guide*.

Numeric Literal Expression

Function

The **numeric literal** expression represents a decimal number as a string of digits with an optional decimal point.

Syntax

```
[+|-]string[.string]
```

Example

The following view includes cities with populations less than half a million:

```
define view smaller_cities of c in cities
   with c.population < 500000
   c.city, c.state, c.population;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the discussion of specifying a datatype in the *Data Definition Guide*.

Quoted String Expression

Function

The **quoted string** expression represents a string of ASCII characters enclosed in single (') or double (") quotation marks. ASCII printing characters are shown in the following table.

Table 4-1. ASCII Printing Characters

Characters	Description
A—Z	Uppercase alphabetic
a—z	Lowercase alphabetic
0—9	Numerals
! @ # $ % ^ & * ( ) _ - + = ' ~ [ ] { }	Special characters

Syntax

```
"string"
```

Example

The following relation definition includes quoted string expressions in a validation expression:

```
define relation political_subdivisions
   code varying [4],
   name varying [25],
   area long,
   incorporation date,
   capital,
   pol_type char [1] valid_if
      (pol_type = 'S' or
      pol_type = 'P' or
      pol_type missing);
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the entry for valid if clause described in the entry for **define field**.

Record Selection Expression (RSE)

Function

The **record selection expression** specifies the conditions for record selection in trigger statements and record inclusion in view definitions.

Unlike the RSE in GDML and **qli**, the record selection expression in the DDL does not support the **first** and **sorted** clauses. (It does support the first *expression*.)

Syntax

```
record-source[with-clause][reduced-clause]
record-source::= {relation-clause|cross-clause}
relation-clause::= [context-variable in]
relation-name
cross-clause::= relation-clause cross record-
source
[over field-name-commalist]
```

The following sections describe the four clauses used with the record selection expression:

- Cross clause
- Reduced clause
- Relation clause
- With clause

Cross Clause

Function

The **cross** clause performs a join operation. It creates dynamic relationships by matching up records from two or more different relations in the same database.

The relationship can be based on the equality of common fields (equijoin), inequalities (non-equijoin), or where no relationship exists (cross product). Unlike most other RSE clauses, *cross-clause* can be repeated to include as many relations as are necessary.

Syntax

```
cross relation-clause[over field-name-commalist]
```

Options

over

Equivalent to a *with-clause* that equates a field in one relation with a field in another. The *field-name* must be exactly the same in both relations. Otherwise, you must use the *with-clause*, even if both fields are based on the same global field.

Examples

The following view definition joins the STATES relation and two instances of the CITIES relation to include cities that are bigger than the capital cities of their respective states:

```
define view large_non_capitals of s in states
   cross c in cities over state cross
   cs in cities with cs.state = c.state and
   cs.city = s.capital and cs.population <
c.population
   c.city query_header "BIG CITY",
   s.state_name,
   s.capital;
```

The following view definition joins the STATES and SKI_AREAS relations to include the area name, city, and full state name for ski areas:

```
define view ski_cities of s in states
   cross ski in ski_areas with s.state = ski.state
   ski.name,
   ski.city,
   s.state_name;
```

Reduced Clause

Function

The **reduced** clause performs a project operation, retrieving only the unique values for a field.

When you ask for a record stream projected on a field, the InterBase access method considers a list of fields and eliminates records that do not have a unique combination of values for the listed fields. If you include a **reduced** clause in an RSE for a view, the view can contain only the fields listed in the **reduced**

clause. Do not erase or modify records through views whose RSE includes a **reduced** clause.

Syntax

```
reduced to dbfield-expression-commalist
dbfield-expression::=context-variable.field-name
```

dbfield-expression
Specifies the field on which you want to project the result. You must specify the context variable declared in the *relation-clause.*

Example

The following view definition restricts the CROSS_COUNTRY relation to the first occurrence of a STATE value:

```
define view ski_states of c in cross_country
   reduced to c.state
   c.state;
```

Note

Because this view contains a reduced clause, its field list can include only the fields listed in the projection.

Relation Clause

Function

The **relation** clause identifies the source relations for the trigger or view.

Syntax

```
context-variable in relation-name
```

Options

context-variable
Associated with a relation to provide name recognition or to distinguish that relation from another. A context variable can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

Gdef is not sensitive to the case of the context variable. For example, it treats “B” and “b” as the same character.

Examples

The following view definition displays only the names of states in which the capital is not the largest city:

```
define view small_capital_city of
   s in states cross c in cities over state cross
   cs in cities with cs.state = c.state and
   cs.city = s.capital and
   cs.population < c.population
   reduced to s.state, s.capital
   s.state_name,
   s.capital;
```

The following trigger definition for a view uses the relation clause in an **if** statement to check a condition. It also uses the relation clause in a **store** statement to assign values:

```
   define trigger
check_and_store_political_subdivisions for
state_view
post store:
   if not any p in political_subdivisions
      with p.code = new.code
   store x in political_subdivisions
      x.code = new.state;
      x.name = new.state_name;
      x.area = new.area;
      x.capital = new.capital;
      x.incorporation = new.statehood;
      x.pol_type = 'S';
   end_store;
end_trigger;
```

With Clause

Function

The **with** clause specifies a search condition or combination of search conditions.

When you pass s search condition to InterBase, it evaluates the condition for each record that might possibly qualify. Conceptually, InterBase performs a record-by-record search, comparing the value you supplied with the value in the database field you specified. If the two values are in the relationship indicated by the operator you specified (for example, equals), the search condition evaluates to "true" and that record becomes part of the

record stream. The search condition can result in a value of "true," "false," or "missing" for each record.

Syntax

```
with boolean-expression
```

boolean-expression
Specifies a valid Boolean expression used to select records.

Examples

The following view limits the qualifying CITIES records to only those with values stored for latitude and longitude:

```
define view geo_cities of c in cities with
   c.latitude_degrees not missing and
   c.longitude_degrees not missing
   c.city,
   c.state,
   c.altitude,
   c.latitude,
   c.longitude;
```

The following trigger definition for a view uses the selection clause in an **if** statement to check a condition:

```
define trigger
check_and_store_political_subdivisions for
state_view
post store:
   if not any p in political_subdivisions
      with p.code = new.code
   store x in political_subdivisions
      x.code = new.state;
      x.name = new.state_name;
      x.area = new.area;
      x.capital = new.capital;
      x.incorporation = new.statehood;
      x.pol_type = 'S';
   end_store;
end_trigger;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the entries in this chapter for:

- Boolean expression
- Value expression

Statistical Expression

Function

The **statistical** expression calculates a value based on a value expression.

If a field value included in *value-expression* is missing for a record, that record is not included in the calculation. For **average**, **max**, and **min**, if the record stream created by RSE is empty, the value of the statistical expression is missing. For **total** and **count**, if the record stream is empty, the total is 0.

Syntax

```
{statistical-operation value-expression of rse|
count of rse}
```

Example

The following view includes only those cities with populations less than the average population of all cities in the relation:

```
define view lt_avg_cities of c in cities
   with c.population < average c1.population of c1
   in cities
   c.city, c.state;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the entries in this chapter for:

- Record selection expression
- Value expression

Username Expression

Function

The **username** expression is a value expression that automatically picks up the username or login of the person running the program and returns it in all uppercase. You can use this expression in triggers, views, and validation expressions. For example, when you combine this expression with a trigger that automatically stores the username of users storing or modifying records, you can keep track of who does what to which records.

Syntax

```
rdb$user_name
```

rdb$user_name

A value expression to which is assigned the username or login. This expression can only be used in assignments and RSEs, and cannot be qualified with a context variable. Returns the username in all uppercase.

Example

The following view uses the *username-expression* in a view's RSE to limit the available records to the querying user:

```
define view censored_employees of
   e in employees cross r in reviews over
   last_name, first_name with e.user_name =
      rdb$user_name
   e.last_name,
   e.first_name,
   e.user_name,
   e.job_code,
   r.review;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the entries in this chapter for:

- Record selection expression
- Value expression

Value Expression

Function

The **value** expression is a symbol or string of symbols from which InterBase calculates a value. InterBase uses the result of the expression when executing the trigger or materializing the view that includes the expression.

Syntax

```
value-expression::= {arithmetic-expression|
dbfield-expression|first-expression|numeric-
literal-expression|quoted-string-expression|
statistical-expression|(value-expression)|
-value-expression
```

Example

The following example uses database field value expressions in the field list for a view:

```
define view phone_list of
   e in employees with e.local_phone not missing
      e.name,
      e.dept_no,
      e.local_phone;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See entries in this chapter for:

- Arithmetic expression
- Database field expression
- First expression
- Null expression
- Numeric literal expression
- Quoted string expression
- Statistical expression
- Username expression
- Value expression

Chapter 5
DDL Statements

This chapter contains entries for the InterBase DDL statements.

Overview

To define and modify metadata for an InterBase database, you can use the following statements:

define database	**define view**
define field	**delete**
define filter	**delete shadow**
define function	**modify database**
define index	**modify field**
define relation	**modify index**
define security_class	**modify relation**
define shadow	**modify trigger**
define trigger	**modify view**

Database

Function

The **define database** statement creates a database definition file and provides the following information about the database:

- The name of the *primary file* that identifies the database for users
- The name of any *secondary files* in which the database is stored
- The page length of database pages
- The security class used for database access
- Any comments the database definer wants to include

By default, **gdef** defines databases that are contained in single files. If you expect your database to grow larger than the capacity of a single disk, you may want to create a multiple-file database when you first create it. Or you can create a single file database now and add secondary files later.

Syntax

```
define database quoted-filespec [length integer
[pages][page_size integer][{textual commentary}]
[security_class class-name][secondary-file];

secondary-file::= {file quoted file-spec [length
integer[pages] | starting [at][page] integer]}
```

Options

quoted-filespec

Specifies the primary file. It must be a valid file specification enclosed in single (') or double (") quotation marks. If the shell you regularly use is case-sensitive, make sure that you always reference the database file exactly as it is spelled in the **define database** statement.

The file specification can contain the full pathname to another node in the network. File specifications for remote databases have the following form:

Table 5-1. Remote Database Access

From	To	Syntax
VMS	VMS via DECnet	node-name::filespec
VMS	ULTRIX via DECnet	node-name::filespec
VMS	non-VMS and non-ULTRIX	node-name^filespec
ULTRIX	VMS via DECnet	node-name::filespec
Apollo	Apollo	//node-name/filespec
Everything Else	Whatever is left	node-name:filespec

length *integer* **pages**

Determines the maximum number of pages that are allocated to the primary file. If you have overflow pages, you can store them in secondary files.

When you define a multiple-file database, you may specify the maximum length of the primary file. If you do not, you must specify a starting position for the first secondary file.

page_size

Specifies a page size to override the default page size of 1024 bytes. You can create databases with page sizes of 1024, 2048, 4096, and 8192 bytes. A larger page size allows a more shallow "tree" structure in the index. Since each index bucket is one page long, longer pages mean larger buckets and fewer levels in the index hierarchy.

To change the page size of an existing database, back up the database with **gbak** and restore it using the **page_size** switch to specify the new size.

{textual commentary}
Stores the bracketed comments about the database in the database. The commentary can include any of the ASCII characters in the following table.

Table 5-2. ASCII Printing Characters

Characters	Description
A—Z	Uppercase alphabetic
a—z	Lowercase alphabetic
0—9	Numerals
! @ # $ % ^ & * ( ) _ - + = ' ~ [ ] { }	Special characters

security_class *class-name*
Associates a security class with the database. See the entry for **define security_class** in this chapter for more information. If you use the Interbase **security_class** scheme to secure the database, you can continue to secure it either with security classes, or with the SQL Grant/Revoke security scheme. However, once you are dealing with objects smaller that a database, you should use only one scheme or the other.

file *quoted file-spec*
Names one or more secondary files that hold database pages after the primary file is filled. You can specify secondary files when the database is defined, or add them if they become necessary later. If you have defined a remote database, secondary files must be associated with the same remote node, except in an Apollo ring.

starting [**at**][**page**] *integer*
Specifies that page number at which a secondary file starts.

When you define secondary files, you must declare a range of pages for in each file. You may do that by specifying:

- A length for each file with the **length** *integer* **pages** clause
- A starting page number for each secondary file with the **starting at page** *integer* clause
- A combination of length and starting page numbers

The *Data Definition Guide* discusses multiple-file databases in the chapter on creating a database.

Examples

The following statements each define a single file database:

```
define database "/gds/examples/atlas.gdb";

define database "atlas.gdb" page_size 2048;

define database "/usr/jimbo/boats.gdb"
 { the test database }
 security_class lmu;

define database "/usr/igor/datafiles/atlas.gdb";
```

The following statement creates a database that is stored in four 10,000-page long files:

```
define database "world_atlas.gdb"
file "world_atlas.gdba" starting at page 10001
length 10000 pages
file "world_atlas.gdbb"
   length 10000 pages
file "world_atlas.gdbc"
   length 10000 pages;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the chapter on creating a database in the *Data Definition Guide*.

See also the entry in this chapter for **define security_class**.

Define Field

Function

The **define field** statement describes the characteristics of a new field for later inclusion in a relation.

You can also define fields in relations. See the entries for **define relation** and **modify relation** in this chapter for more information.

To disallow duplicate values for a field, you define a unique index on that field. See the entry for **define index** in this chapter for more information about the unique constraint.

Syntax

```
define field field-name datatype
field-attributes [system_flag integer];

field-attributes ::= datatype[comments|
edit-string|missing-value|query-header|
query-name|security-class|valid_if]
```

Options

field-name
Names the field you want to create. A field name can contain up to 31 characters and can be alphanumeric, dollar signs ($), and underscores (_). It must start with an alphabetic character. It must also be unique among all global fields in the database.

datatype
Specifies the field's datatype. The datatype specification must precede other field attributes and descriptive comments. Interbase supports integer, float, character, date, blob, and multi-dimensional array datatypes.

field-attributes
Describes the characteristics of fields defined or modified by the following statements:

- **define field** (except *security-class*)
- **modify field**
- **define relation**
- **modify relation** (*comments*, *security-class*, and the **qli** attributes only)

The following sections describe the eight clauses for field attributes:

- Datatype clause
- Comments clause
- Edit-string clause
- Missing-value clause
- Query header clause
- Query name clause
- Security class clause
- Valid if clause

Datatype Clause

Function

The **datatype** clause specifies the datatype of a field. It is the only required field attribute.

Syntax

```
datatype ::= {short[array-clause][scale-clause]
|long [array-clause][scale-clause]|
float[array-clause]|double[array-clause]
|char[n][array-clause][sub_type fixed]|
varying[n][array-clause][sub_type fixed]|
date[array-clause]|blob-clause}

scale-clause ::= scale[-]n

array-clause ::= (dimension-comma-list)

dimension ::= range|range, dimension-comma-list

range :: = integer-value|integer-value:integer-
value

blob-clause ::= blob[sub_type {text|blr|acl|-n}]
[segment_length n]
```

The following table lists supported datatypes for languages other than ADA.

Table 5-3. Supported Datatypes by Language

Datatype	BASIC	C	COBOL	FORTRAN	Pascal	PL/I
short	word	short	s9(4)comp	1*2	integer	fixed binary(15)
long	long	long	s9(9)comp	1*4	integer32	fixed
float	single	float	comp-1	real	real*8	float binary(24)
double	double	double	comp-2	double_ precision	double	float(53)
char[n]	string	char[n]	x(n)	character dimension(n)	array[1...n] of char	character
varying[n]	string	char[n]	x(n)	character dimension(n)	array[1...n] of char	character
date	gds_$quad_t	gds_$quad	s9(18) comp	1*4 dimension(2)	gds_$quad	gds_$quad
blob	gds_$quad_t	gds_$quad	s9(18) comp	1*4 dimension(2)	gds_$quad	gds_$quad

The following table lists supported datatypes for the various supported implementations of Ada.

Table 5-4. Supported Datatypes for Ada

Datatype	VMS	Verdix	Alsys	Telesoft
short	short integer	short integer	integer	integer
long	integer	integer	long integer	long integer
float	float	short float	float	float
double	interbase double	float	long float	long float
char[n]	string (1..n)	string (1..n)	string (1..n)	string (1..n)
varying[n]	string (1..n)	string (1..n)	string (1..n)	string (1..n)
date	interbase.quad	interbase.quad	interbase.quad	interbase.quad
blob	interbase.quad	interbase.quad	interbase.quad	interbase.quad

The following table lists the datatypes by size and range/precision.

Table 5-5. Datatype Size and Precision

Datatype	**Size**	**Range/Precision**
short	16 bits	-32768 to 32767
long	32 bits	-2**31 to (2**31)-1
float	32 bits	Approximately 7 decimal digits
double	64 bits	Approximately 15 decimal digits
char[n]	*n* bytes	0 to 32767 characters
varying[n]	Varies up to *n* bytes	0 to 32767 characters
date	64 bits	1 January 100 to 11 December 5941
blob	Varies	None

Both the date and blob datatypes are represented above by **gds_$quad**, a quantity for which InterBase allocates 64 bits of storage. However, this quantity is functionally different for dates and blobs:

- For the date datatype, **gds_$quad** represents the date encoded in 64 bits. The GDML library includes two routines, **gds_$encode_date** and **gds_$decode_date**, for date manipulation. See the *Programmer's Guide* for more information.
- For the blob datatype, **gds_$quad** represents an identifier that points to the actual blob data. The format of the data depends on the application. The blob identifier stored in the record is a 64-bit quantity. The blob itself is of unlimited size; a blob can exceed 65,535 bytes and is limited only by the amount of physical storage available.

Varying string is a character datatype that includes a count at the beginning. This datatype is not directly supported by most host languages, so it is generally returned as a character string.

Note

When you use an InterBase datatype that is not supported by your host language, InterBase automatically converts such fields to equivalent types that are supported. To ensure that variables you define match the

datatypes in database fields, use GDML's **based on** clause to establish the datatype of your variables. See the *Programmer's Guide* and the entry in this chapter for **define field** for more information about the **based on** clause.

Options

`scale-clause`

Specifies the power of 10 by which InterBase multiplies the stored integer value for use by **qli**, COBOL, and PL/I.

For example, a negative scale of two means that there should be a decimal point two places to the left of the rightmost digit (as in the normal format for dollars and cents).

`array-clause`

Specifies the range and dimension of the array. InterBase supports multi-dimensional arrays limited in range to between -32768 and +32768. You can specify a range giving only the higher bound, and **gdef** will default the lower bound to 1. Further limitations to range and dimension may apply depending on what your host language accepts.

For more information on limitations with blobs, see the chapter on using blobs in the *Programmer's Guide*.

`blob-clause`

Provides the characteristics of blob fields. The optional **segment_length** clause specifies a segment length that system components use for various purposes. For example, **gpre** uses this value to set up a buffer for data transfer between the calling program and InterBase, and **qli** uses the segment length to format its display.

Gdef provides a default value of 80 if you do not include the **segment_length** clause. If you update the system relations directly and leave the segment length missing, **gpre** and **qli** supply lengths of 512 and 40, respectively, for their own purposes.

`subtype-clause`

For blob fields, there are three predefined subtypes: **text**, **blr** (request language statements), and **acl** (access control lists). **Qli** uses the subtype to determine how it should display a blob. If your application requires special blob handling, you can define your own subtype; the range of negative values from -1 to -32768 is reserved for users.

For **char** and **varying** fields, the **fixed** subtype is defined as a convenience for C programs. Text strings are normally passed to C as null-terminated strings. If your application requires that a field contain non-ASCII binary values that may include nulls, declare the field to have the **fixed** subtype so it will *not* be truncated at the first null byte.

Example

The following statements define fields with various datatypes:

```
define field tolerance long scale -2;

define field text_blurb blob sub_type text
   segment_length 60;

define field price long
   valid_if (price > 0);

define field manufacturer char[10]
   valid_if
      (manufacturer ne "SHIPPER_X" and
      manufacturer ne "SHIPPER_Y" and
      manufacturer ne "SHIPPER_Z");

define field encrypted_key char[20]
   sub_type fixed;

define field array1 long (0:2, -2:3) scale 2;

define relation parts
   item_code char[6],
    item_name char[25],
    manufacturer char[10],
    blurb blob segment_length 60,
    price long,
          ⇓
```

Comments Clause

Function

The **comments** clause lets you store bracketed comments about the field in the database.

Syntax

```
{textual-commentary}
```

Options

textual-commentary
A comment can include any of the ASCII characters in the following table.

Table 5-6. ASCII Characters

Characters	Description
A—Z	Uppercase alphabetic
a—z	Lowercase alphabetic
0—9	Numerals
! @ # $ % ^ & * ( ) _ - + = ' ~ [ ] { }	Special characters

Example

The following statements define fields and use comments:

```
define field standard_date date { all-purpose date
field };

define relation parts
   item_code  char[6]  {alphanumeric identifier},
   item_name  char[25] {abbreviated product name},
   supplier   char[10] {aka manufacturer},
   blurb blob segment_length 60 { this field stores
                                  the descriptions
                                  of the items in
                                  inventory },
   price long,
           ⇓
```

Edit String Clause

Function

The **edit-string** clause specifies an alphabetic, numeric, or date format for a field or computed value. Only **qli** uses edit strings.

Syntax

```
edit_string "edit-string"
```

The following syntax is used to delete an edit-string of a short and long integer:

```
QLI> modify relation <relation_name> modify field <field_name>
edit_string ''
```

Note that there is no space between the single quotes.

The following tables list the edit string characters.

Table 5-7. Characters for Alphabetic and Miscellaneous Fields

Character	Meaning of Edit String Character
A (*integer*)	Any alphabetic character. For example, "aaabxxba" yields "HAL 14L" from the value "HAL14L." **Qli** returns an error if the characters are not alphabetic.
x (*integer*)	Any alphanumeric character. See "A" above for an example.
B (*integer*)	A blank space. See "A" above for an example.
-	A hyphen. For example, "aaa-xx-a" yields "HAL-14-L" from the value "HAL14L."
'string' or "string"	Print the quoted string. For example, "xxx-xx-xxxx" yields "012-34-5678" from the value "012345678."

Table 5-8. Characters for Date Fields

Character	Meaning of Edit String Character
Y (*integer*)	The year, from right to left. For example, the field value "1987", "y(1)" yields "7" and y(2)" yields "87."
M (*integer*)	The name of the month. The integer specifies how many of the characters in the month name to print. For example, "m(3)" yields "Jan."
N (*integer*)	The numeric month. The best value for the integer is 2. For example, "n(2)" yields "01" for January and "11" for November.
D (*integer*)	The day of the month. The best value for the integer is 2. For example, "d(2)" yields "01" and "11" for the first and eleventh days of the month.
W (*integer*)	The name of the day of the week. The integer specifies how many of the characters in the day name to print. The minimum value of the integer should be 2. For example, "w(2)" yields "Mo."
B	A blank space. A "b" in a date edit string enhances readability. For example, "d(2)bm For example, "d(2)bm(3)by(4) yields dates formatted as "29 May 1956."
J (*integer*)	The Julian day of the year. For example, "01" is 1 January and "32" is 1 February.
T (*integer*)	The time portion of a date field. Unless you append the P edit string discussed below, the time is based on a 24-hour clock. You must supply your own punctuation. For example, "t(2):t(2):t(2): yields time formatted as "14:23:31."
P[P]	Changes the T time display to a 12-hour clock followed by the meridian value (ante or post). If you specify only one P, it displays "A" or "P;" otherwise, it displays "AM" or "PM." You must supply your own punctuation. For example, "tt:tt:bpp" yields time formatted as "2:23:31 PM."
X (*integer*)	The date and time portion of a date field, based on a 24-hour clock. **Qli** supplies the punctuation. For example, "x(25)" yields a date formatted as "23-SEP-1987 15:38:11.9721."

Table 5-9. Characters for Numeric Fields

Character	Meaning of Edit String Character
9 (*integer*)	An ordinary digit. For example, “9999.99” yields “0832.79” for the value “83279” if it has a scale of -2. With an integer value of scale 0, it prints a numeric overflow.
.	Decimal point. See “9” above for an example.
B (*integer*)	Blank space.
,	Comma for thousands, millions, etc. For example, “99,999.99” yields “65,832.79” from the value “6583279.”
z (*integer*)	A leading digit or blank if the leading position is zero.
+	Leading plus sign. Prints leading sign for positive and negative numbers. This sign takes up one character space.
-	Leading minus sign. Prints leading sign for negative numbers only. This sign takes up one character space.
$	Leading dollar sign. Multiple dollars sign “float;” an edit string of “$$$$$$.99” yields “$123.45,” for “123.45,” “$12.34” for “12.34,” and “$1234.56” for “1234.56.” This sign takes up one character space.
*	A leading asterisk (for checks). This sign takes up one character space.
H (*integer*)	Hexadecimal representation of a character.
(())	Parentheses to print around negative numbers.
DB	Prints DB for debit after negative numbers.
CR	Prints CR for credit after negative numbers.
“”	Quoted strings can be included in a number. For example, *print 27.95 using $$99.99b”tsk!btsk!”* prints “$37.95 tsk! tsk!”

Example

The following statements define a database and three relations, each containing several fields with edit strings:

```
define database "stuff.gdb";
define relation budgets
b1 long,
b2 long edit_string "999,999",
```

```
b3 long edit_string "((999,999))",
b4 long edit_string "-ZZZ,ZZZ,ZZ9";

define relation employee_stuff
social_security char [9] edit_string "xxx-xx-
xxxx",
phone_number char [10] edit_string "(xxx)Bxxx-
xxxx",
salary long edit_string "HHHHHHHHHBBB";

define relation family_dates
   name varying [10],
   birth date edit_string "w(3),bd(2)bm(12)by(4)",
   wedding date edit_string "d(2)bn(2)by(4)",
   awareness date edit_string "y(4)";
```

Missing Value Clause

Function

The **missing_value** clause provides a literal string (numeric or character) that is displayed if no value is stored for that field. If you store that value in the field, InterBase marks the field as missing. The missing value must be a legitimate value for the field's datatype, size, and scale.

Syntax

missing_value[**is**]{*fixed-point-number*|*quoted-string*}

Options

fixed-point-number
A number that is displayed as the missing value. The number must not exceed the length of the field. For integer datatypes, the number cannot include a decimal point unless the field has a scale factor. For floating datatypes, the number should include a decimal point.

quoted-string
A quoted literal expression that is displayed as the missing value. The string must not exceed the length of the field.

Example

The following statements define fields with explicit missing values:

```
define field price float
   missing_value is -15.75
```

```
   valid if
      (price > 0 or
      price missing);

define field headwater_state
   missing_value is "??";
```

Query Header Clause

Function

The **query_header** clause provides an alternate column label for use in **qli**. When **qli** displays selected records, it uses the query header instead of the field name. A **qli** user can override the query header by supplying one in the query.

Syntax

query_header [**is**] *quoted-string*[/*quoted-string*]

Options

quoted-string

A quoted literal expression that is displayed as the column header. If you want the query header to be displayed on more than one line, supply a quoted string followed by a slash (/) and another quoted string.

Examples

The following statement defines a field with a query header:

```
define field longitude_degrees char[2]
   query_header "longd";
```

The following statement defines a relation and assigns query headers to some fields:

```
define relation cities
   { largest 200 population centers }
   city,
   state,
   population,
   latitude_degrees char[2]
      query_name latd
      query_header "latd",
   latitude_minutes char[3]
      query_name latm
      query_header "latm",
   latitude_compass char[1]
      query_name latd
```

```
      query_header "latd",
   longitude_degrees char[2]
      query_name longd
      query_header "longd",
   longitude_minutes char[2]
      query_name longm
      query_header "longm",
   longitude_compass char[2]
      query_name longc
      query_header "longc";
```

The following statement defines a query header that is printed across three lines:

```
define field longitude_degrees char[2]
   query_header "long-"/"itude"/"degrees";
```

The following statement suppresses the query header for the field:

```
define field latitude_degrees char[2]
   query_header " ";
```

Query Name Clause

Function

The **query_name** clause provides an alternate field name for use in **qli**. You can reference a field by its full name or by the query name.

Syntax

query_name [**is**] *alternate-name*

Options

alternate-name
A query name can contain up to 31 characters that can be alphanumeric, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

Examples

The following statement defines a field with a query name:

```
define field longitude_degrees char[2]
   query_name longd;
```

The following statement defines a relation and assigns a query name to six fields:

```
define relation cities
   { largest 200 population centers }
   city,
   state,
   population,
   latitude_degrees char[2]
      query_name latd,
   latitude_minutes char[3]
      query_name latm,
   latitude_compass char[1]
      query_name latd,
   longitude_degrees char[2]
      query_name longd,
   longitude_minutes char[2]
      query_name longm,
   longitude_compass char[2]
      query_name longc;
```

Security_Class Clause

Function

The **security_class** clause associates a security class with a field in a relation or a view.

You can associate a security class with a field only in a **define relation**, **modify relation**, **define view**, or **modify view** statement. Because the security rules apply to a field in a relation, and thus are not a global characteristic, you cannot use a security_class clause in a **define field** statement.

Caution

Interbase also supports the SQL security scheme of **grant** and **revoke** statements. Since mixing InterBase security and SQL security may produce unexpected results, you should pick one security scheme and use it consistently when assigning security to database objects.

Syntax

```
security_class class_name
```

Options

class-name
Names the security class you want to associate with the field. You must define the security class in a **define security_class** statement.

Example

The following statement defines a security class and then associates it with a field in a relation:

```
define security_class lmu
   { limited metadata update. This class keeps
everyone but Zaphod from assigning rights. }
      zaphod pdrwc,
      %.zaphod rw,
      %.%.gds r,
      view less_of_a_secret r;

define relation r1
   no_name char [10] security_class lmu;
```

Valid_if Clause

Function

The **valid_if** clause provides a field-level integrity criterion that InterBase checks when it stores the record or updates the field in a record. If the new value fails the test, the field assignment fails. Because the validation criteria are stored in the database, they eliminate the need for such checks in the programs that access the database.

A validation expression differs from a trigger in that a trigger has full context and can perform updates. A validation expression can reference any field.

Syntax

```
valid_if (boolean-expression)
```

Options

boolean-expression
A valid Boolean expression.

Example

The following statements define fields with validation expressions:

```
define field price long
valid_if (price > 0 or price missing);
```

```
define relation manufacturers
   manufacturer char[10] valid_if
      (manufacturer ne "SHIPPER_X" and
      manufacturer ne "SHIPPER_Y" and
      manufacturer ne "SHIPPER_Z" and
        manufacturer not missing);
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the entries for:

- **define field**
- **define relation**
- **define view**
- **modify field**
- **modify relation**
- Boolean expression

See the *Programmer's Guide* for a discussion of **grant/revoke** security.

See the *Data Definition Guide* for a discussion of security classes.

Define Filter

Function

The **define filter** statement associates a blob filter with a database. A blob filter is a program that converts data stored in blob fields from one subtype to another. Before you define a blob filter to the database, you:

1. Write it in C or Pascal.
2. Compile it.
3. Define a library in which to store it.
4. Access blob filters from any host language program that uses embedded GDML statements.

If you want to redefine a filter, you must first use a **delete filter** statement and then create a new definition with a **define filter** statement.

Syntax

```
define filter filter-name [{textual commentary}]
input-type type output_type type
entry_point 'routine-name'
module 'module-name';
```

Options

filter-name
Names the filter you want to define. A filter name must be unique within a database. It can contain up to 31 characters that can be alphanumeric, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

`textual commentary`
Stores the bracketed comments about the filter in the database. The commentary can include any of the ASCII characters in the following table.

Table 5-10. ASCII Characters

Characters	Description
A—Z	Uppercase alphabetic
a—z	Lowercase alphabetic
0—9	Numerals
! @ # $ % ^ & * ( ) _ - + = ' ~ [ ] { }	Special characters

input-type and **output-type**
Input type specifies the pre-defined sub-type of the data stored in the blob field. Output type specifies the pre-defined sub-type the data in the blob field is to be filtered to. Each input_type output_type combination must be unique within a database.

'routine-name'
Specifies the name of the entry point as it is known in the link library. The entry point is the name of the filter as you specify it in your program. It is case-sensitive and can contain up to 31 characters that can be alphanumeric, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

'module-name'
Specifies the name of the library in which you store the filter. It is case-sensitive and can contain up to 31 characters that can be alphanumeric, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

Example

The following statement defines a blob filter to the database. In this example, compressed data has a pre-defined sub-type of -12, and decompressed data has a pre-defined sub-type of -30. When the filter is invoked, it converts the data from its input type of -12 to the output type -30:

```
define filter decompressor
{Convert data from compressed to decompressed.}
input-type -12 output_type -30
entry_point 'bf_decom';
```

Troubleshooting

You may encounter the following messages when defining a filter:

- *Database version is too old for the new syntax*

  You cannot define filters for pre-Version 3.0 databases. Once you upgrade your database, you can use the **define filter** statement.
- *Filter entry point must be specified*

  You forgot to include an entry point for the filter.
- *Filter module must be specified*

  You need to include a module name or dynamic link library for the filter.

See Also

See the entry in this chapter for **delete**.

See also the chapter on blob filters in the *Programmer's Guide.*

Define Function

Function

The **define function** statement defines user-defined functions, or executable routines, to the database. Before you define a function to the database, you:

1. Write it in any host language callable from C
2. Compile it.
3. Define a library in which to store it.

If you want to modify a function, you must first use a **delete function** statement and then create a new definition with a **define function** statement.

Syntax

```
define function function-name[query_name query-
name]module_name 'module-name'
entry_point 'entry-name'
[{textual-comments}]
[argument-description-comma-list]
argument-description::= datatype by
argument-mode return mode
argument-mode::={reference}
return-mode::={return_value}
```

Options

function-name
Names the function you want to define. A function name must be unique within a database. It can contain up to 31 characters that can be alphanumeric, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

query-name
Specifies an alias for the function name. It can contain up to 31 characters that can be alphanumeric, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

'module-name'
Specifies the name of the dynamic link library in which you store the function. It is case-sensitive and can contain up to 31

characters that can be alphanumeric, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

'entry-name'
Specifies the name of the entry point as it is known in the link library. The entry point is the name of the function as you specify it in your program. It is case-sensitive and can contain up to 31 characters that can be alphanumeric, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

`textual comments`
Stores the bracketed comments about the function in the database. The commentary can include any of the ASCII characters in the following table.

Table 5-11. ASCII Characters

Characters	Description
A—Z	Uppercase alphabetic
a—z	Lowercase alphabetic
0—9	Numerals
! @ # $ % ^ & * ( ) _ - + = ' ~ [ ] { }	Special characters

reference
Specifies the address of the argument passes to the function.

return_value
Specifies you want a value to be returned. To return a pointer value, designate the **return_value** with an argument mode of **by reference**.

Example

The following example defines the function upper to a database:

```
define function upper
module_name 'FUNCLIB'
entry_point 'FN_UPPER'
char[256] by reference return_value;
char[256] by reference;
```

Troubleshooting

You may encounter the following messages when defining a function:

- *Database version is too old for the new syntax*

  You cannot define functions for pre-Version 3.0 databases. Once you upgrade your database, you can use the **define function** statement.
- *Function entry point must be specified*

  You need to include an entry point for the function.
- *Function module must be specified*

  You need to include an library name for the function.
- *Argument mode is by value or by reference*

  You need to specify an argument mode of **by value** or **by reference** for the function.
- *Return mode is by return_value or return_argument*

  You need to specify a return mode of **return_value** or **return_argument** for the function.

See Also

See the entry in this chapter for **delete**.

See also the chapter on user define functions in the *Data Definition Guide*.

Define Index

Function

The **define index** statement defines an index on a field or fields in a relation. You must define a relation and its fields before you can create an index for it.

InterBase automatically maintains all indexes. You do not have to reference an index when you access data. The InterBase access method does it automatically.

Syntax

```
define index index-name [for] relation-name
  [{unique|duplicates}]
  [{asc[ending]|desc[ending]}]
  [{active|inactive}]
  [{textual-commentary}]
  field-name-commalist;
```

Options

index-name
Names the index you want to create. An index name can contain up to 31 characters that can be alphanumeric, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

relation-name
Specifies the relation for which you are defining the index. You cannot define an index on an external relation.

unique
Disallows duplicate values in the index.

Use **unique** when you define an index on fields used as *primary keys*, such as unique identification numbers, part numbers, and employee numbers. You cannot define a unique index on a field containing duplicate entries. If you specify this option, the values for the field names or combinations of field names must then be unique. If you try to store a value that already exists, the assignment operation fails. Finally, note that no part of a unique key may be null.

If you create a multi-segment index, first consider which of the key fields is likely to have the most unique values. Having done

so, you should list the field names in descending order by uniqueness. Such ordering improves index compression.

`duplicates`
Allows duplicate values in the index. This is the default.

`ascending | descending`
Specifies the order in which an index is built. If neither qualifier is specified, the default order is ascending.

For increased efficiency in returning sorted values, use the qualifier that corresponds to the order you are most likely to specify in an ordering clause. Using the qualifier does not replace using an ordering clause when you retrieve values.

`active | inactive`
Active specifies that the index should be built when the current transaction ends. Inactive specifies that the index should be built at a later time. If the definition of an index is marked as inactive, InterBase maintains only the index definition in the database. When you change the state of the index to active with the **modify index** statement, the index is built and becomes available to all users.

Because InterBase automatically maintains all indexes, you may want to specify an inactive index for a relation if you are going to store many records at one time.

{*textual-commentary*}
Stores the bracketed comments about the relation in the database. The commentary can include any of the ASCII characters in the following table.

Table 5-12. ASCII Printing Characters

Characters	Description
A—Z	Uppercase alphabetic
a—z	Lowercase alphabetic
0—9	Numerals
! @ # $ % ^ & * ( ) _ - + = ‘ ~ [ ] { }	Special characters

field-name

Specifies one or more fields from *relation-name* that will be indexed. You can create a single or multi-segment index for a relation. A single-segment index consists of a single field, while a multi-segment index consists of two or more fields. In both cases, you should avoid indexing a field that has few unique values. Such indexes provide little performance improvement and can reduce update performance. Finally, because of the nature of the blob datatype, you cannot index a blob field.

Examples

The following statements define relations and some indexes for them:

```
define relation states
   ⇓

define relation cities
   ⇓

define index state_idx1 for states
   unique state;

define index state_idx2 for states
   inactive unique state, state_name;

define index river_idx1 for rivers
   unique  { speed access and
      eliminate duplicates }
   descending river;

define index rivstat_idx1 for river_states
   duplicate river, state;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the entry in this chapter for **delete**.

Define Relation

Function

The **define relation** statement creates a relation.

Syntax

```
define relation relation-name
[system_flag integer]
[external_file quoted-filespec]
[{textual-commentary}]
[security_class class-name]
field-description-commalist;
field-description::=
{included-field|new-field|renamed-field|
computed-field}
```

Options

relation-name
Names the relation you want to create. A relation name can contain up to 31 characters that can be alphanumeric, dollar signs ($), and underscores (_). However, it must start with an alphabetic character. "

system_flag *integer*
Assigns a value to the RDB$SYSTEM_FLAG relation for this database object. The only reserved values are: 0 (or missing), the default for user data; 1, for metadata maintained by the InterBase access method; and 2, for **qli** and **pyxis** metadata. These are assigned by **gdef**. For more information on system relations, see the appendix in the *Data Definition Guide*."

external_file *quoted-filespec*
Names an operating system (Apollo stream file and VMS sequential and indexed files only) that contains data for the relation.

{textual-commentary}
Stores the bracketed comments about the relation in the database. The commentary can include any of the ASCII characters in the following table.

Table 5-13. ASCII Printing Characters

Characters	Description
A—Z	Uppercase alphabetic
a—z	Lowercase alphabetic
0—9	Numerals
! @ # $ % ^ & * ( ) _ - + = ' ~ [ ] { }	Special characters

`security_class` *`class-name`*
Associates a security class with the relation or fields within the relation. See the entry for **define security_class** in this chapter.

`field-description-comma-list`
Specifies the name of the field in the relation. The **define relation** statement supports several types of field definitions, all of which you can use in the same relation definition.

`included-field`
Such fields are defined in previous **define field** or **define relation** statements, and can have optionally specified local attributes. The local attributes are described at length in the entry for **define field** in this chapter.

`new-field`
Fields that are defined in the relation.

```
field-name [local-attributes]
local-attributes::= {comments|edit-string|query-
name|security-class|position n}
```

If you specify a local attribute that conflicts with the attribute defined in the field definition, the local attribute overrides the global attribute for this use of the field. However, you cannot override the field's missing value, validation criteria, and datatype and related subclauses such as scale or segment length because they are part of the core definition of the global field.

Except for the position clause discussed in this section, these local attributes are described at length in the entry for **define field** in this chapter.

field-name field-attributes

Defines a new field within the relation, instead of using existing or virtual fields. Because the field is defined from scratch, you must include the field's datatype. **Gdef** adds these fields to the global list of fields for that database, thereby making those fields available for inclusion in subsequent relation definitions.

renamed-field

Fields defined elsewhere and renamed for the relation.

local-name **based on** *field-name*[*local-attributes*]
local-attributes::={*comments*|*edit-string*|*query-name*
|*security-class*|**position** n}

This clause renames a field for use in the relation being defined, at the same time retaining all characteristics except those you change explicitly. You can include a textual description, an edit string, a query name, a security class, and a position clause.

If you specify a local attribute that conflicts with the attribute defined for the global field in the **define field** statement, the local attribute overrides the global attribute. However, you cannot override the field's missing value, validation criteria, and datatype and its related subclauses (scale, segment length, and so on), because they are part of the core definition of the global field.

Except for the position clause (discussed below), these local attributes are described at length in the entry for the **define field** statement in this chapter.

computed-by-field

A computed or virtual field that consists of a formula rather than a storage location. Note that the value expression must be in parentheses.

local-name [*datatype*] **computed by** (*value-expression*) *local-attributes*

InterBase does not stores data in such fields, but it calculates the formula and retrieves requested data. If you do not specify the optional *datatype*, InterBase calculates an appropriate datatype.

Because the computed field depends on values from its context, it cannot be used in arbitrary relations. **Gdef** generates a unique name for the global portion of the field. Therefore, a computed field does not need a name that is unique in the database.

position *n*
Specifies the position (left to right) that **qli** uses to print when displaying the relation. The first field is position 0. For example, if there are three fields, A, B, and C, with defined positions of 1, 0, and 2, respectively, **qli** displays these fields in the order B, A, and C.

If you do not specify a position, **qli** uses the order in which the fields are defined or included in the relation.

Examples

The following example first defines several fields with **define field** statements and then includes them in a relation:

```
define field state char[2];
define field state_name varying [25];
define field city varying[25];

define relation states
   { basic information about states }
   state,
   state_name,
   area long,
   statehood char[4],
   capitol based on city;
```

The following statement defines several fields within a relation, defines some computed values, and also includes several existing fields:

```
define relation cities
   { info about capitols and largest cities }
   city,
   state,
   population long,
   altitude long,
   latitude_degrees varying[3]
      query_name latd,
   latitude_minutes char[2]
      query_name latm,
```

```
   latitude_compass char[1]
      query_name latc,
   longitude_degrees varying[3]
      query_name longd,
   longitude_minutes char[2]
      query_name longm,
   longitude_compass char[1]
      query_name longc,
   latitude computed by (
      latitude_degrees | " " |
      latitude_minutes |
      latitude_compass),
   longitude computed by (
      longitude_degrees | " " |
      longitude_minutes |
      longitude_compass);
```

The following example defines a field and then includes it in a relation under different names:

```
define field population long;

define relation populations
   { US census data by state }
   state,
   census_1950 based on population
      query_name c1950,
   census_1960 based on population
      query_name c1960,
   census_1970 based on population
      query_name c1970,
   census_1980 based on population
      query_name c1980;
```

The following example defines a computed field in a relation:

```
define relation aldermen
   first_name varying [10],
   last_name varying [20],
   aldermen_name computed by (first_name | ' ' |
      last_name);
```

In this example, note that the fields that comprise the computed field are both varying strings. Were the constituent fields fixed-length strings, they would be padded with blanks, a fact that

would defeat the purpose of this computed field. For example, consider two relations, ALDERMEN, as defined above, and OLDERMEN, with fixed-length values for the name fields that make up the full name. Suppose you stored a record with "Louis" and "Lapine" as the values for FIRST_NAME and LAST_NAME in the ALDERMEN relation and F_NAME and L_NAME in the OLDERMEN relation. The following displays demonstrate the difference:

```
        ALDERMEN
         NAME
===============================

Louis Lapine

        OLDERMEN
         NAME
===============================

Louis    Lapine
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the entries for:

- **define field**
- Value expression

See also the *Data Definition Guide* for a discussion on defining fields.

Define Security_Class

Function

The **define security_class** statement establishes access control lists that you can associate with objects (databases, relations, views, and fields in relations and views).

In the InterBase security class scheme, all users have total access to all objects by default. Once you assign a security class to an object, however, only those members of the class have access. All others users are locked out.

To modify a security_class, you must use the **delete** statement and then redefine the class by using the **define security_class** statement.

Caution

InterBase also supports a security scheme that uses the **grant** and **revoke** statements in SQL. You should not mix the InterBase security class method with SQL security since doing so may give you unexpected results. For more information on **grant** and **revoke**, refer to the *Programmer's Guide*.

Syntax

```
define security_class class-name element-
commalist;
element::= {grantee|view view-name}
privilege-list
privilege-list::= {R|W|P|C|D}
VMS: grantee:== [uic]
UNIX: grantee:== [group,user]
Apollo: grantee:==
user[.project[.organization[.node]]]
```

Options

class-name
Names the security class you want to create. A security class name can contain up to 31 characters that can be alphanumeric, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

element
Defines individual and group members of a security class or group. You can use the wildcard character (% for UNIX and

Apollo, * for VMS) to substitute for any of the group or user identifiers.

`grantee`
Identifies the individual or group receiving privileges.

`view` *`view-name`*
Identifies the view for which an individual or group has privileges.

`privilege-list`
The privilege list specifies the following privileges.

Privilege	Access
R (read)	Users with read privilege can read the object.
W (write)	Users with write privilege can write to the object.
P (protect)	Users with protect privilege can change the security class for the object.
C (control)	Users with control privilege can change the metadata for the object.
D (delete)	Users with delete privilege can delete the definition for the relation.

On Apollo, **gdef** automatically orders *element* entries with views appearing first, followed by other entries in the order that an Apollo ACL would be processed. For example, julie. forms.documentation.node is processed before "%.%.%.%." On other systems, access control elements are applied in the order they are listed.

Usage

InterBase security views objects in a hierarchy, with the database being the highest level object and the field the lowest. The higher-level access controls override the lower. For example, if you have write privileges for a database, and read privileges for a specific relation in that database, you can only write to that relation.

You can write to a relation in which there is field that you cannot read. Provided you can write to the relation, you can store a record in the relation as long as you don't reference the field you can't access. When the record updates, the InterBase access method automatically sets that field's value to missing.

You can assign security classes to objects three different ways:

- Assign a security class that has been defined in an earlier data definition statement
- Assign a security class and then define it in the same data definition statement
- Assign a security class in one data definition statement, and then define it in another statement at another time.

Example

The following statements define a database, fields, a relation, views and security classes, and associates security class with each of the entities:

```
define database"personnel.gdb"
      security_class personnel;

define relation employees
      security_class employees
   emp_no   short,
   name     varying[25],
   address  varying[25],
   home_phonechar[10]
            edit_string "(xxx)Bxxx-xxx",
   local_phonechar[4]
   salary   long scale -2
            edit_string "$,$$$,$$9.99"
            security_class salary_clearance,
   mgr_no   based on emp_no
            query_header "Mgr"/"No",
   dept_no  short;

define view phone_list of
   e in employees with e.local_phone not missing
      security_class public_read
   e.name,
   e.dept_no,
   e.local_phone;

define view staff of
   mgr in employees cross staff in employees with
      mgr.emp_no = staff.mgr_no
      security_class public_read
   mgr_no from mgr.emp_no query_header 'Mgr'/'No',
```

```
   mgr_name from mgr.emp_no query_header
"Manager",
   staff.emp_no,
   staff.name  query_header "Staff Name",
   staff.salary;

define security_class database_administrator
   dana pcrwd,
   dh   rw,
   jcf  r;

define security_class employees
    dana pcrwd,
    dh   rw,
    jcf  r,
view phone_list r,
view staff r;

define security_class salary_clearance
    dana  pcrwd,
    dh    rw,
    jcf   r;

define security_class public_read
    dana  pcrwd,
    dh    r;

modify relation rdb$security_classes
   security_class database_administrator;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the entries in this chapter for:

- **define database**
- **define relation**
- **define view**
- **delete**

Define Shadow

Function

The **define shadow** statement creates a physical copy of a database and stores it on disk. Once created, a shadow maintains a duplicate copy of the database. A shadow can consist of one or more files. A shadow consisting of multiple-files is called a shadow set.

Syntax

```
define shadow integer {manual|auto} 'filespec'
[file-length] [additional-file(s)];
additional-file::= {file 'filespec' {file-
length} | {starting-page}}...
file_length::= length integer [pages]
starting-page::=starting [at] [page] integer
```

Options

integer
A number that identifies the shadow or the shadow set.

manual|auto
The required **auto|manual** keyword controls what action is taken if a user attempts to attach to a database whose shadow file(s) are not available. If the keyword is **manual**, the database attach fails. Database attaches will continue to fail until either the shadow file(s) becomes available or the -k(ill) option of **gfix** is used to delete all references to the unavailable shadow files from the database metadata. (The shadows are not deleted from the metadata; only references to the shadow file(s) are deleted.) If the keyword is **auto**, references to the unavailable shadow file(s) are automatically deleted and the database attach proceeds.

filespec
An explicit pathname and filename for the shadow file.

length *integer* [**pages**]
Length of the database pages of an additional shadow file. Pages is an optional word.

additional-file
Any of one or more additional shadow files. When you create multiple shadow files, you must specify either a file length or a starting page number for each file. Multiple shadow files are

good to use if your database is larger than the space available on one disk. You can define multiple shadow files and spread them over several disks.

starting [**at**] [**page**] *integer*
Memory page on the disk at which an additional shadow file begins. At and page are optional words.

Examples

The following example creates a single shadow file of the atlas database:

```
define shadow 1 auto 'atlas.shadow';
```

The following statements create shadow sets of the atlas database. The first example specifies starting pages for the shadow files; the second example specifies page length for the shadow files:

```
define shadow 1 auto
   '/interbase/casey/atlas.shadow'
   file '/interbase/casey/atlas.shadow1'
      starting at page 100
   file '/interbase/casey/atlas.shadow2'
      starting at page 200;

define shadow 2 manual 'employees.shadow'
   length 100
   file '/interbase/casey/employees.shadow1'
      length 100
   file '/interbase/casey/employees.shadow2'
      length 100;
```

Troubleshooting

You may encounter the following message when defining a shadow:

- *Shadow number must be a positive integer*

  You used an unacceptable character for the shadow number. Redefine the shadow using a positive integer.

See Also

See the entry in this chapter for **delete shadow**.

See also the chapter on shadowing in the *Database Operations* guide.

Define Trigger

Function

The **define trigger** statement specifies an action InterBase performs automatically whenever you execute a store, modify, or erase operation on a specified relation. You can also use triggers to define and keep track of events (database insertions, modifications, or deletions) of interest to other applications.

Syntax

```
define trigger trigger-name for relation-name
[{active|inactive}]{pre|post}
{store|modify|erase}[sequence-number]:
trigger-action
end trigger
[message-description-comma-list];
message-description::= message abort-code:
"message-text"]
[{textual-comments}]:
```

Options

trigger-name
Names the trigger you want to write. A trigger name can contain up to 31 characters and can be alphanumeric, dollar signs ($), and underscores (_). It must start with an alphabetic character. It must also be unique among all global fields in the database.

relation-name
Associates the trigger with the relation.

active|inactive
This action indicator specifies whether or not the trigger should take effect when the transaction ends. The default is active.

pre|post
This time indicator specifies if the trigger is to fire before or after the associated store (insert), modify (update), or erase (delete) operation.

store|modify|erase
Specifies on which operation a trigger is to fire. A store trigger fires on an insert, a modify trigger fires on an update, and an erase trigger fires on a delete.

`sequence-number`

This sequence indicator groups triggers together and specifies when they are executed in relation to other groups of triggers. This only applies when you define multiple triggers of the same type for the same relation (for example, several pre store type triggers).

A group with a lower indicator executes before one with a higher indicator, and within each group, the order of execution is random. The default is 0.

`trigger-action`

Specifies a GDML statement InterBase executes whenever you store a new record into the relation, modify a field from a record in the relation, or erase a record from the relation.

Gdef supplies two predefined context variables, **old** and **new**, for use in the trigger action. **Old** refers to the record you are modifying or erasing, and **new** refers to the new record or version you are creating.

`message` *`abort-code`***`:`**

The abort code is an integer of type short. It cannot exceed 255.

`message-text`

Specifies the message associated with the abort code. You define individual abort codes and messages for each trigger. The same abort code in different triggers does not have the same trigger message unless you define it to be so. You can define multiple abort codes and messages for each trigger. You can use any of the ASCII characters listed in Table 5-14 for the message text.

The text of the message is limited to 78 characters. If you define a trigger message using more than 78 characters, you will see an error mesage when you process the trigger through **gdef**.

`textual-comments`

Stores or modifies comments about the trigger in the database. The *textual-comments* can include any of the ASCII characters in the following table.

Table 5-14. ASCII Characters

Characters	Description
A—Z	Uppercase alphabetic
a—z	Lowercase alphabetic
0—9	Numerals
! @ # $ % ^ & * ( ) _ - + = ' ~ [ ] { }	Special characters

Trigger Language Usage

The trigger definition language is a limited form of embedded GDML with several significant extensions. Like GDML, the trigger language requires context variables to eliminate ambiguous references. The differences are the:

- **start_stream** form of data retrieval is not supported.
- **on error** clause is not supported.
- Blob handling language is not supported.

These extensions to the trigger language form of GDML provide capabilities normally supplied by the host language in embedded GDML applications:

- **If-then-else**. Because the primary function of a trigger is to test conditions and take appropriate actions, the trigger language includes an if-then-else capability. The **else** portion is optional. Both **then** and **else** take a single statement as their arguments. To include multiple statements, use a begin-end block.
- **Abort** (*n*). This statement causes a statement to terminate, returning the value *n* (numeric). It is returned as the fourth word in the status vector, and is printed in the error message. The abort message is also returned.
- Assignments. Embedded GDML relies on host language assignment expressions to move data from database fields to

program variables. A simple assignment (=) is included in the trigger language. An assignment in the *new* context of a trigger is made to the record whose **store** or **modify** caused the trigger to be executed.

- **Begin-end**. A begin-end pair defines a block of statements that executes as one.
- An expanded choice of value expressions, including **null**, **rdb$user_name**, **average**, **max**, **min**, **count**, **total**, **any**, **unique**, and **first...from** Because the trigger language includes an assignment statement, all the database value expressions can be used in an assignment as well as in a record selection expression.

Additional extensions to the trigger language include:

- **post**. If you want to keep track of a particular event, you can define a unique string in the trigger that identifies the event, and include a **post** statement. When the action occurs, notification is sent to an event manager, which alerts interested applications.
- **matching using**. Using the InterBase matching language wildcards, you can define your own wildcard search characters. For more information on the **matching using** operator, see the discussion of retrieving data in the *Programmer's Guide.*
- **uppercase** (*fieldname*). By placing the keyword uppercase in front of a field name, you can uppercase the field values referenced in a trigger.

For a more complete discussion of triggers, see the chapter on preserving data integrity in the *Data Definition Guide.*

Examples

The following example is a trigger that prevents a user from deleting a city from the CITIES relation if that city is a foreign key in the TOURISM and SKI_AREAS relations. It is also a multiple message trigger that contains two abort codes. Each of the abort code is associated with its own message:

```
define trigger restricting_delete for cities
pre erase 0:
   begin
      for t in tourism with t city = old.city
      and t.state = old.state
      abort 1;
```

```
        end_for;
        for s in ski_areas with s.city = old.city
        and s.state = old.state
        abort 2;
        end_for;
        end;
     end_trigger
message 1: "This city cannot be deleted, because
     it exists in the TOURISM relation"
message 2: "This city cannot be deleted, because
    it exists in the SKI_AREAS relation";
```

The following trigger posts an event to the events manager when the stock price changes beyond a specifies range:

```
define trigger stock_change for stocks
   pre modify:
      if new.price / old.price not between
      0.99 and 1.01 then post new.company;
end trigger;
```

The following statements define a relation and a trigger. The trigger logs the name and date into the log_new_tons relation when you store a city name containing TON:

```
define relation log_new_tons
   name char [10]
   when date;

define trigger tabs_on_ton for cities
post store:
   if new.city matcning '*TON*' using '-s(* = ?*)'
   store x in log_new_tons
x.name = uppercase (new.city)
x.when = 'today'
end_store;
end_trigger;
```

Troubleshooting

You may encounter the following messages when defining a trigger:

- Expected (store , modify, erase, end_trigger) found (incorrect syntax)

  This is a standard syntax error. Check your program for inconsistent syntax. Correct any errors and try again.

- *Database version too old for the trigger syntax.*

  You cannot use this syntax in pre-Version 3.0 databases. Once you upgrade your database, you can use the new syntax.

- *Message number exceeds 255*

  You used an unacceptable integer for the abort code. Redefine the trigger using an integer within range.

See Also

See the entries in this chapter for:

- Boolean expression
- Record selection expression
- Value expression

Define View

Function

The **define view** statement creates a view definition that can include fields from one or more relations. A view can be:

- A vertical subset of a relation. The view definition limits the fields that are displayed.
- A horizontal subset of a relation. The view definition limits the records that are displayed.
- A single relation subset vertically and horizontally. The view definition limits both the fields displayed and the records in those fields.
- A combination of relations subset horizontally, vertically, or both.

You can access views as if they were relations. You can:

- Select records from it
- Project on its fields
- Join it with another relation
- Join it with itself
- Include it in a union

However, the source of the view determines which, if any, update operations you can perform on the view:

- If a view is a vertical subset of a single relation, you can treat it as a relation for both retrieval and update purposes, provided that all excluded fields allow missing values.
- If a view references more than one relation, you must define its update logic using triggers.

Syntax

```
define view view-name of rse
[system_flag integer][{textual-commentary}]
[security_class class-name]
field-name-commalist;

field-name ::= {included-field|renamed-field|
computed-field}
```

Options

view-name
Names the view you want to create. A view name can contain up to 31 characters and can be alphanumeric, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

rse
Selects the records that constitute the view. You can use any option of the record selection expression in defining the view except the *first-clause* and the *sorted-clause*. See the entry in Chapter 3 for RSE for more information on record selection.

system_flag *integer*
Assigns a value to the RDB$SYSTEM_FLAG relation for this database object. The only reserved values are: 0 (or missing), the default for user data; 1, for metadata maintained by the InterBase access method; and 2, for **qli** and **pyxis** metadata. For more information on RDB$ relations, see the chapter on system relations in the *Data Definition Guide*.

{ *textual-commentary* }
Stores a bracketed descriptive comment about the view. The commentary can include any of the ASCII characters in the following table.

Table 5-15. ASCII Characters

Characters	Description
A—Z	Uppercase alphabetic
a—z	Lowercase alphabetic
0—9	Numerals
! @ # $ % ^ & * ( ) _ - + = ' ~ [ ] { }	Special characters

security_class *class-name*
Associates a security class with the view. You can also treat a view as if it were a user. This can allow users of a view different access rights to the view's base tables than the same users have when accessing the base tables directly. See the entry for **define security_class** in this chapter for more information.

field-name

Specifies the field(s) you want to include in the view. The **define view** statement supports several types of field definition, all of which you can use in the same view definition.

included-field

Fields from any of the source relations, with optionally specified local attributes.

dbfield-expression [*local-attributes*]

The *dbfield-expression* is a field name qualified by a context variable. The context variable is declared in the record selection expression used to limit the records for the view.

renamed-field

Fields from any of the source relations, renamed for use in the view.

local-name **from** *dbfield-expression* [*local-attributes*]

The **from** clause renames a qualified field name for use in the view being defined, at the same time retaining all characteristics except those you change explicitly.

computed-field

Fields from any of the fields in the source relations that have value expressions as data.

local-field[*datatype*]**computed by**(*value-expression*) [*local-attributes*]

The **computed by** clause defines a virtual field that is a formula rather than a reference to stored fields.

local-attributes ::= {*comments*|*edit-string*|*query-name*|*query-header*|*security-class*|**position** n}

For each of the fields in a view, you can include a textual description, an edit string, a query name, a query header, and a security class. Except for the **position** clause described below, these local attributes are described at length in the entry for *field-attributes* in this chapter.

If you specify a local attribute that conflicts with the attribute defined for the global field in the **define field** or **define rela-**

tion statement, the local attribute overrides the global attribute. The only attributes you cannot override are the field's validation criteria, its missing value, and datatype and related subclauses such as scale or segment length.

position *n*
Specifies the position (left to right) that **qli** uses to print when displaying the view or that **gpre** uses for an SQL **select** * statement. The first field is at position 0. For example, if there are three fields, A, B, and C, with defined positions of 1, 0, and 2, respectively, **qli** displays these fields in the order B, A, and C. If you do not specify a position, **qli** uses the order in which the fields are defined or included in the view.

Examples

The following statement defines a vertical subset of a relation (that is, a subset of fields):

```
define view geo_cities of c in cities
   { subset of CITIES with geographic data only }
   c.city,
   c.state,
   c.altitude,
   c.latitude,
   c.longitude;
```

The following view defines both a horizontal and vertical subset (that is, selected records and a subset of fields):

```
define view middle_america of c in cities
   with c.longitude_degrees between 79 and 104 and
   c.latitude_degrees between 33 and 42
   c.city,
   c.state,
   c.altitude;
```

The following view defines a horizontal subset of a relation by using an existential qualifier to test another relation for field values from the first relation:

```
define view ski_states of s in states
with any shush_boom in ski_areas
with s.state = shush_boom.state
   s.state,
   s.capitol,
   s.area,
   s.population;
```

The following view joins two relations, using values from fields in both relations to compute population densities for each decade's census:

```
define view population_density of p in populations
   cross s in states over state
   p.state,
   density_1950 computed by
      (p.census_1950 / s.area),
   density_1960 computed by
      (p.census_1960 / s.area),
   density_1970 computed by
      (p.census_1970 / s.area),
   density_1980 computed by
      (p.census_1980 / s.area);
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the entries in this chapter for:

- **define field**
- **define security_class**
- Value expression

Delete

Function

The **delete** statement erases the specified database entity *and all data associated with that entity*. Therefore, you must be very sure that you want to delete something before you do issue this statement.

Although objects can be deleted from an active database, it is generally better to wait until the database is not in use if you want to delete relations and indexes.

Syntax

```
delete {database database-name|field field-name|
filter filter-name|function function-name|index
index-name|relation relation-name|security_class
class-name|trigger trigger-name|view view-name};
```

Options

database-name
Specifies the database to delete.

field-name
Specifies the field to delete. You can delete a field that was defined in a **define field** or **define relation** statement. However, because fields are included in a relation, you must first delete the field from each relation in which it is included. To do so, you must use the drop clause option of the **modify relation** statement.

filter-name
Specifies the filter to delete.

index-name
Specifies the index to delete. Although objects can be deleted from an active database, it is generally better to wait until the database is not in use if you want to delete an index.

relation-name
Specifies the relation to delete. Although objects can be deleted from an active database, it is generally better to wait until the database is not in use if you want to delete a relation.

class-name
Specifies the security class to delete.

`trigger-name`
Specifies the trigger to delete.

`view-name`
Specifies the view to delete.

Usage

You can delete a field that was defined in a **define field** or **define relation** statement. However, because fields are included in a relation, you must first delete the field from each relation in which it is included. To do so, you must use the *drop-clause* option of the **modify relation** statement.

Examples

The following statements delete a field from a relation and then from the database:

```
modify relation states
   drop statehood;

delete field statehood;
```

You must explicitly delete a field defined in a **define relation** statement after dropping it from relations in which it is used. For example, suppose you defined the field BIRTH_DATE in an EMPLOYEES relation, subsequently included it in another relation, and then decided not to keep it. The following sequence deletes the field from all its instances and then finally from the database itself:

```
modify relation employees
   drop birth_date;

modify relation demographics
   drop birth_date;

delete field birth_date;
```

You cannot delete a field that is used in a computed field, trigger or view definition.

Deleting a function eliminates all parts of the function definition. You can delete a function that someone else is using and it will not cause an error. The following statement deletes a function:

```
delete function upper;
```

Deleting a blob filter eliminates all parts of the filter definition. You can delete a filter that someone else is using and it will not cause an error. The following statement deletes a filter:

```
delete filter decompressor;
```

You can delete any index you want, except one that is actively being used. The following statement deletes an index:

```
delete index idx4;
```

You can delete any relation you want, but if you delete a relation that someone else is using, the other user's program will get an unrecoverable error. Because the **delete relation** statement removes a relation and all its records, you should use this statement with caution. The following statements delete relations:

```
delete relation gudgeons;

delete relation non_eeoc_approved_data;
```

You can delete a security class without first deleting it wherever it is referenced. The objects associated with the deleted security class are then unprotected.

InterBase treats views much like relations. However, because a view is only a virtual relation, the effect of deleting a view that is being used by someone else is less catastrophic than deleting a relation that is being used. In general, when you delete a view, other users should not encounter any problems if they are already running their programs. However, if they start up a program that references the deleted view, the program fails when it tries to compile the request that mentions that view.

The following statements delete views:

```
delete view population_density;
delete view geo_cities;
delete view riv_vu;
```

When you delete a trigger for a relation, all associated trigger messages are deleted as well.

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the entries for:

- **define database**
- **define field**
- **define filter**
- **define function**
- **define index**
- **define relation**
- **define security_class**
- **define trigger**
- **define view**

Delete Shadow

Function

The **delete shadow** statement deletes a shadow or a shadow set.

Syntax

```
delete shadow integer
```

Options

integer
Specifies the shadow you want to delete.

Example

The following example deletes a shadow file from its database:

```
delete shadow 1;
```

Troubleshooting

See the *Data Definition Guide* for standard **gdef** error messages.

You may also encounter the following message when you delete a shadow:

- database version is to old for the new syntax

You cannot delete shadows for pre-Version 3.0 databases. Once you upgrade your database, you can use the **delete shadow** statement.

See Also

See the entry in this chapter for **define shadow**.

Modify Database

Function

The **modify database** statement:

- Changes the security class or comments for a database
- Adds secondary files to the database

To change the page size of an existing database, back it up with **gbak** and restore it using the **page_size** switch to specify the new size.

Syntax

```
modify database quoted-filespec[length integer
pages]
[{textual commentary}]
[drop description]
[security_class class-name]
[drop security-class]
[add-file quoted-filespec]
[length integer[pages]]|starting[at][page]
integer]];
```

Options

quoted-filespec

A valid file specification enclosed in single (') or double (") quotation marks. If the shell you regularly use is case-sensitive, make sure that you always reference the database file exactly as it is spelled in the **define database** statement.

The file specification can contain the full pathname to another node in the network. File specifications for remote databases are listed in Table 5-16.

Table 5-16. Remote Database Access

From	To	Syntax
VMS	VMS via DECnet	node-name::filespec
VMS	ULTRIX via DECnet	node-name::filespec
VMS	non-VMS and non-ULTRIX	node-name^filespec
ULTRIX	VMS via DECnet	node-name::filespec

Table 5-16. Remote Database Access continued

From	To	Syntax
Apollo	Apollo	//node-name/filespec
Everything Else	Whatever is left	node-name:filespec

length *integer* **pages**
Sets an upper bound on the number of pages allocated to the file. Overflow pages are created in the secondary files. If you do not specify a length for the primary file, you must specify a starting position for the first secondary file.

If you specify a primary file length that is less than the current length, **gdef** ignores it.

textual commentary
Stores the bracketed comments about the database in the database. The commentary can include any of the ASCII characters in the following table:

Table 5-17. ASCII Characters

Characters	Description
A—Z	Uppercase alphabetic
a—z	Lowercase alphabetic
0—9	Numerals
! @ # $ % ^ & * ( ) _ - + = ‘ ~ [ ] { }	Special characters

drop description
Deletes an existing description for the database.

security_class *class-name*
Associates a security class with the database. See the entry for **define security_class** in this chapter for more information.

add file *quoted-filespec*
Adds one or more secondary files to a database in which InterBase stores database pages after the primary file is filled.

starting [**at**] [**page**] *integer*
Specifies the page number at which a secondary file starts. When you define secondary files, you must declare a range of pages for in each file by giving a starting page number.

Examples

The following statements each appear as the first statement in a source file used to define or modify a database:

```
modify database "/usr/igor/war_effort.gdb"
   { metadata last updated 7 November 1985 }
   security_class lmu;

modify database "atlas.gdb";

modify database "/usr/hector/dingies.gdb";
```

The following statement adds secondary files to an existing database:

```
modify database "atlas.gdb"
   length 10000 pages
   add file "atlas.gdba"
      starting at page 10001
      length 10000
   add file "atlas.gdbb"
      length 10000
   add file "atlas.gdbc"
      length 10000;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the entry in this chapter for **define database**.

See the chapter on the **gbak** utility in the *Database Operations* guide.

See the chapter on creating databases in the *Data Definition Guide*.

Modify Field

Function

The **modify field** statement changes the characteristics of an existing field.

When you modify fields, note that:

- Because of its ability to keep track of older record versions, InterBase does not have to make massive updates to the disk when you change a field definition. No matter how many times the definition changes, InterBase can access the appropriate version of the field definition.
- The changes you make will not affect existing users of the database until they re-attach to it.
- InterBase always writes to disk the newest version of the field definition when a record is updated.

If you want to change the characteristics of a field defined inside a relation, you may not be able to change it within that relation. For example, physical attributes, such as the datatype, can be changed only with the **modify field** statement. Physical characteristics of a field become part of the global field definition, even if they are defined within a relation definition.

Fields defined in relations (*locally defined fields*) are automatically added to the list of fields in the database (*globally defined fields*). Although you may define a field inside a relation, **gdef** disregards the source of the definition and makes it a globally defined field, as if it were defined with a **define field** statement.

If you want to change anything other than the field's position, commentary as a part of that relation, edit string, query header, query name, or security class as a part of that relation, you must do so with the **modify field** statement. **Gdef** then makes the change for all uses of that global field in all relations of the database.

If you change the datatype or length of fields upon which a **computed by** field is based, the length of the computed field does not change automatically. This does not cause a problem if you decrease the size of the base fields, but causes a data conversion

error if you increase them, particularly if the computation consists of concatenated strings. The preferred way to fix this problem is to delete and re-create the computed field. The easy way is to manually increase the value of RDB$FIELD_LENGTH of the RDB$FIELDS record for the computed field.

Syntax

```
modify field field-name field-attributes
[system_flag integer];
```

Options

drop valid_if (boolean_expression)
Removes the valid_if clause from the field. You cannot remove any field validation while modifying a relation.

field-name
Names the field you want to create or the field you want to modify.

field-attributes
Specifies the datatype, length or scale if appropriate, and several optional field characteristics. You can change the textual description of the field or its query name. For detailed information about the syntax of these options, see the description of field attributes in the entry for **define field** in this chapter.

system_flag *integer*
Assigns a value to the RDB$SYSTEM_FLAG relation for this database object. The only reserved values are:

- 0 (or missing), the default for user data
- 1, for metadata maintained by the InterBase access method
- 2, for **qli** and **pyxis** metadata

For more information on the system relations, see the appendix on sytem relations in the *Data Definition Guide.*

Example

The following statements modify fields:

```
modify field zip char[9];

modify field headwater_state
   missing_value is "??";

modify field price long;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See also the entry in this chapter for **modify relation**.

See also the description of field attributes in the entry for **define field**.

Modify Index

Function

The **modify index** statement changes the uniqueness, activeness, or direction of an index. If you want to add or drop fields from an index, you must delete the index and then redefine it. Modifying an index causes it to be rebuilt. There may be some delay while the index is rebuilding.

Syntax

```
modify index index-name
[unique|duplicate]
[active|inactive]
[asc{ending}|desc{ending}];
```

Options

index-name
Specifies the index you want to modify.

unique
Changes an index to allow duplicate index values to one that does not allow duplicate values. You cannot change from duplicate to unique if the field already contains duplicate values. You should not modify an index to unique while there are active users in the database.

duplicate
Changes an index to disallow duplicates to one that allows duplicates.

active
Changes an inactive index to an active one.

inactive
Changes an active index to an inactive one. When you are loading many records, you may want to turn off indexes for the relation into which you are storing.

ascending
Changes a descending index to one that is built in ascending order.

descending
Changes an ascending index to one that is built in descending order.

Example

The following statements modify indexes:

```
modify index idx_1 duplicate active;

modify index idx_2 unique;
```

Troubleshooting

See Appendix for a discussion of errors and error handling.

See Also

See the entries in this chapter for:

- **define index**
- **modify relation**

Modify Relation

Function

The **modify relation** statement can change a relation's complement of fields and local field characteristics.

Syntax

```
modify relation relation-name[{textual-
commentary}]{operation-commalist];

operation::= {add field field-name[field-
attributes] [position integer]system_flag
integer|
drop description|drop field field-name|
external_file quoted-filespec
modify field field-name[field-attributes|
drop security_class[security-class-name]|
security_class security-class-name|
system_flag integer}
```

Options

relation-name
Identifies the relation you want to modify.

textual-commentary
Stores comments about the relation in the database. The *textual-commentary* can include any of the ASCII characters in the following table.

Table 5-18. ASCII Characters

Characters	Description
A—Z	Uppercase alphabetic
a—z	Lowercase alphabetic
0—9	Numerals
! @ # $ % ^ & * ( ) _ - + = ‘ ~ [ ] { }	Special characters

add field *field-name*
Adds a field to the relation:

- If the field has already been defined in the database, *field-attributes* can specify the position of the field, textual description of the field, a query name for use with **qli**, and a security class.
- If the field does not exist elsewhere, you must specify a datatype. You can also specify an explicit missing value, a validation expression, the position of the field, textual description of the field, a query name for use with **qli**, and a security class.
- The addition of fields to a relation is identical to the inclusion of fields when you define the relation. See the entry for **define relation** in this chapter for more information.

`position` *`integer`*
Specifies where in the relation you want an added field to appear.

`system_flag` *`integer`*
Assigns a value to the RDB$SYSTEM_FLAG relation for this database object. The only reserved values are:

- 0 (or missing), the default for user data
- 1, for metadata maintained by the InterBase access method
- 2, for **qli** and **pyxis** metadata

For more information on system relations, see the appendix in the *Data Definition Guide*.

`drop description`
Deletes the comment field for the relation.

`drop field` *`field-name`*
Removes the named field from the relation. When you delete a field from a relation, other users should not encounter any problems if they are already running their programs. However, if they start up a program that references the deleted field, the program fails when it tries to compile the request that mentions that field.

You cannot delete fields that are used in views based on this relation without first deleting the field from those views.

external_file *quoted-filespec*
Names an operating system (Apollo stream file, Unix stream file, and VMS sequential and indexed fields only) that contains data for the relation. The *Data Definition Guide* discusses external relations.

modify field *field-name*
Identifies the field whose relation-specific characteristics you want to change. You can change only the position of the field, the textual description, the edit string, the query header, the query name, the security class of the field, the field on which the local field is based (for fields defined with the **based on** option), and the system flag.

You cannot change the datatype, the missing value, or the validation expression. If you want to change one of these attributes, you must do so with the **modify field** statement.

drop security_class [*security-class-name*]
Removes the named security class from the relation.

security_class *security-class-name*
Associates the specified security class with that relation.

Example

The following example modifies a relation by adding fields, dropping fields, and modifying fields:

```
modify database "test_atlas.gdb";

modify relation cities
   add field year_incorporated char[4]
      query_name inc
      position 6,
   add field type_of_government char[1]
      query_name gov
      valid_if
         (type_of_government = "C" or
          type_of_government = "M" or
          type_of_government missing),
   drop field population;
modify relation states
   security_class cabinet_level;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See also the entries in this chapter for:

- **define field**
- **define relation**
- **modify field**

Modify Trigger

Function

The **modify trigger** statement changes the action InterBase performs automatically whenever you execute a store, modify, or erase operation on the relation.

Syntax

```
modify trigger trigger-name for relation-name
{active|inactive}{pre|post}{modify|store|erase}
[sequence-number]:
[textual-commentary]
trigger-action
{message abort-code: "message-text"|drop message
abort-code}
end trigger
[msg-operation-commalist]

msg-operation::={msgadd abort-code:"message-
text"|
msgmodify abort-code:"message-text"|
msgdrop abort-code:"message-text"};
```

Options

`active|inactive`
Changes the action indicator specifying whether or not the trigger should take effect when the transaction ends.

`pre|post`
Changes the time indicator specifying if the trigger is to fire before or after the associated store (insert), modify (update), or erase (delete) operation.

`modify|store|erase`
Changes the trigger action performed on a modify (update), store (insert), or erase (delete) operation. Each of these operations can have a separate trigger action.

`textual-commentary`
Stores or modifies comments about the trigger in the database. The *textual-commentary* can include any of the ASCII characters in the following table.

Table 5-19. ASCII Characters

Characters	Description
A—Z	Uppercase alphabetic
a—z	Lowercase alphabetic
0—9	Numerals
! @ # $ % ^ & * ( ) _ - + = ' ~ [ ] { }	Special characters

trigger-action
Specifies a GDML statement that InterBase executes whenever you store a new record into the relation, modify a field from a record in the relation, or erase a record from the relation. See the *Programmer's Guide* for information about GDML data manipulation.

message *abort-code*:"*message-text*"
Adds an abort code and associated message to the trigger. The abort code must be an integer of type short. It cannot be larger than 255. The message can include any of the ASCII characters in the above table.

drop message *abort-code*
Deletes a specified message from a trigger.

msgadd | **msgmodify** | **msgdrop**
Adds a new message, modifies an existing message, or drops an existing message. This is the alternate way to add, modify or delete a message from a trigger.

Examples

The following statements modify the store trigger defined in the **define relation** statement:

```
modify trigger tabs_on_ton for cities
inactive
pre store 99:
{Sequence number indicates this is one of the last
pre store triggers to fire. It is currently
inactive.}
if new.city matching '*TON*' using '-s(* = ?*)'
   store x in log_new_tons
      x.name = uppercase (new.city)
      x.when = 'today'
```

```
   end_store;
end_trigger;
```

The following example modifies the trigger messages, using the preferred syntax:

```
modify database "foo.gdb"

modify trigger r1_store for relation1
message 1: "greater than 10"
message 2: "less than 5"
drop message 4,
message 23: "useless message"
end_trigger;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See also the entries in this manual for:

- **define trigger**
- Boolean expression
- Record selection expression
- Value expression

For a complete discussion of triggers, see the discussion on preserving data integrity in the *Data Definition Guide*.

Modify View

Function

The **modify view** statement:

- Drops or adds a security class for a view
- Drops or modifies fields for a view

Syntax

```
modify view view-name [{textual-commentary}]
[operation-commalist]
operation ::={drop-field field-name |
drop security_class[security-class-name] |
security_class security-class-name
modify field field-name[field-attributes] |
system_flag integer}
```

Options

view-name
Identifies the view you want to change.

{*textual-commentary*}
Stores the bracketed comments about the view in the database. The *textual-commentary* can include any of the ASCII characters in the following table.

Table 5-20. ASCII Characters

Characters	Description
A—Z	Uppercase alphabetic
a—z	Lowercase alphabetic
0—9	Numerals
! @ # $ % ^ & * ( ) _ - + = ‘ ~ [ ] { }	Special characters

drop field *field-name*
Removes the named field from the view, but not from the source relation(s). You cannot delete fields that are used in views based on this view without first deleting the field from those views.

drop security_class [*security-class-name*]
Removes the named security class. If you do not specify *security-class*, **gdef** removes the security class associated with the view.

security_class *security-class-name*
Associates the specified security class with the view.

modify field *field-name* [*field-attributes*]
Identifies the field whose view-specific characteristics you want to change. You can change only the position of the field, the textual description, the edit string, the query header, the query name, the security class of the field, and the field on which the view field is based.

system_flag *integer*
Assigns a value to the RDB$SYSTEM_FLAG relation for this database object. The only reserved values are:

- 0 (or missing), the default for user data
- 1, for metadata maintained by the InterBase access method
- 2, for **qli** and **pyxis** metadata

For more information on system relations, see the appendix in the *Data Definition Guide*.

Example

The following statement removes a field and drops a security class. Then it adds a new security class:

```
modify view geo_cities
{ new comment goes here }
drop field altitude,
drop security_class,
add security_class top_secret;
```

Troubleshooting

See Appendix A for a discussion of errors and error handling.

See Also

See the entries in this chapter for:

- **define field**
- **define view**
- **define security_class**

Appendix A
Reporting and Handling Errors

This appendix discusses reporting and handling errors and lists error messages.

Overview

Gdef processes a source file line by line until it reaches the end of the file or encounters a fatal error (for example, the database you are trying to create already exits). If it encounters non-fatal errors, it returns them to you in a list that specifies the line number each error is on. When you debug your file, you should work on the first error message first. Frequently, an initial error can cascade through a file. Subsequent error occurrences are related to the first error occurrence.

Error Sources

An error might come from any of three sources:

- **Gdef** itself. These are generally errors **gdef** encounters when parsing a command, such as an unrecognized word, invalid syntax, and so on.
- A database error. Database errors can be any one of many problems. The most likely is the nonexistence of the database specified. Check the file name or pathname and try again.
- A bugcheck. Bugchecks reflect software problems you should report. If you encounter a bugcheck, you should execute a traceback and save the output. Submit the output and the script that led to the bugcheck along with a copy of the database to:

  Borland International Inc.
  InterBase Support
  1800 Green Hills Road
  P. O. 660001
  Scotts Valley, CA 95067

You can also call Customer Support at 800-437-7367 from anywhere in the United States and Canada or at 408-431-5400 from outside of the United States. In addition, you can fax a description of the problem to 408-439-7808.

If you encounter an error and cannot decide why there was an error, review the entry for that statement before submitting the bug report.

Error Format

Gdef reports errors in this format:

filename:integer: message

Filename is the file in which **gdef** encounters an error. The *integer* is the line number on which **gdef** found an error. The *message* may not immediately follow the incorrect line. For example, the following **gdef** session results in an error being reported for line 1:

```
% gdef
GDEF> modify database "carberry.gdb";
I/O error during "ms_$map1" operation for file "carberry.gdb"
-name not found (OS/naming server)
standard input:1: Couldn't attach database
GDEF>
```

Error Messages

You can get error messages when you:

- Invoke **gdef**
- Reference an object that does not exist
- Perform an illegal operation

The following sections describe the types of messages you can receive.

Invoking gdef

You may encounter the following messages when you invoke **gdef**:

- `%DLP-W-IVVERB, unrecognized command verb - check validity and spelling \gdef\` (VMS only).

  This means that you have not included `sys$system:gds_login.com` in your login file to define the InterBase utility names. See the *Database Operations* guide and your installation instructions for more information.

- `"Gdef" - name not found (O/S naming server)`

  (APOLLO only). This means that **gdef** does not show up anywhere along your shell's command search list. Check the search path, correct it if necessary, and try again.

- `Gdef: not found`

  (UNIX) This means that **gdef** does not show up anywhere along your shell's command search list. Check the search path, correct it if necessary, and try again.

- ```
  operating system directive failed
  -no active servers (library/MBX manager)
  -communication error with journal "journal_directory_name"
  ```

  Although a journal has been defined for the database, there is no active journal server. Use **gltj** to start the server.

Other Messages

Most of the messages you receive with **gdef** are self-explanatory. For example, you may reference an object that does not exist, perform some data modification that is not legal, and so on. If the message you receive from **gdef** is not clear, check the reference section
  ```

for the statement causing the problem to see if you overlooked a restriction or part of the command.

A list of messages follows. Many of these messages contain a substitution variable, such as <*string*>, for which **gdef** prints a name or token from your input file:

- A computed expression can not be changed or added
- Action not implemented yet
- Can't drop system relation <relation-name>
- Can't resolve field "<field-name>", Field is undefined or used out of context. Syntax error, flushing input
- Computed by expression must be parenthesized
- Couldn't attach database
- Couldn't create database "<database-name>"
- Data type is a global, not local, attribute
- Data type required for global field
- Database "<database-name>" already exists
- Database "<database-name>" exists but can't be opened
- (EXE)make_desc: don't understand node type. Report this error to Interbase Software Corporation.
- (EXE)string_length: No defined length for blobs
- Expected <string>, encountered "<string>"
- Field <field-name> already exists in relation <relation-name>
- Field <field-name> doesn't exist in relation <relation-name>
- Field <field-name> from relation <relation-name> is referenced in view <view-name>
- Field <field-name> is unknown in relation <relation-name>
- Field <field-name> is used in relation <relation-name> (local name <field-name>) and cannot be dropped
- Gdef: can't open <file-name>
- Gdef: unknown switch <switch-name>
- Global field <field-name> already exists
- Global field <field-name> isn't defined
- Index <index-name> doesn't exist
- Missing value is a global, not local, attribute

- No database declared
- Only SECURITY_CLASS can be dropped, User tried to drop field attribute other than SECURITY_CLASS from a database
- Relation <relation-name> already exist
- Relation <relation-name> doesn't exists
- Security class can appear only on local field references
- Security class <class-name> already exists
- Security class <class-name> doesn't exist
- Segment_length is a global, not local, attribute
- Segment length is valid only for blobs"
- Segment length must positive
- Subtypes are valid only for blobs and text
- Sub_type is a global, not local, attribute
- Unmatched parenthesis
- Unrecognized privilege "<character>"
- Valid if is a global, not local, attribute
- Validation expression must be parenthesized
- **** view definition <view-name> must be recreated ****
- <clause-name> clause is not yet implemented, DEFAULT_VALUES not handled
- **** <operation> trigger source for relation <relation-name> must be recreated ****
- <relation-name> referenced by view <view-name>

A

active 5-29
Aggregate expression
  GDML 4-19
any
  DDL 4-4
Arithmetic expression
  DDL 4-2
ASCII printing characters 4-13, 5-4

B

between
  DDL 4-4
Blob
  datatype 5-9
  gds 5-9
  gds_$quad 5-9
  input_type 5-23
  output_type 5-23
  searching 4-6
  subtypes 5-10
Blob filter
  defining to database 5-22
Boolean expression
  DDL 4-3
  operator precedence 4-3

C

Case sensitivity
  containing 4-6
  matching 4-6
  starting with 4-7
Char datatype 5-11
Column
  alternate name 5-17
Comments 5-12, 5-29
comparison
  DDL 4-5
computed by 5-33
cont, see **containing**
containing
  DDL 4-6
cross
  DDL 4-14–4-15
Customer support information A-2

D

Database
  defining 5-2
  field expression 4-9
  page size 5-3
Datatype
  fixed 5-11
  options 5-10
  overview 5-7
  size and precision of 5-10
  syntax 5-7
Date field
  edit string characters 5-14
DDL
  first 4-10
  list of statements 5-1
define database
  DDL 5-2
define field
  DDL 5-6–5-11
define filter 5-22
define function 5-25–5-27
define index
  DDL 5-28–5-30
define relation
  DDL 5-31–5-36
define security_class 5-37–5-40
define shadow 5-41–5-42
define trigger 5-43–5-48
define view 5-49–5-53
delete
  overview 5-54
delete field 5-54
delete shadow 5-58
delete view 5-56
DYN
  language options 2-3
  relation to **gdef** 2-2

E

edit_string 5-13–5-16
Errors
  DDL, App. A
  messages, App. A

F

Fax number for InterBase A-2
Field
  alternate names 5-18
  defining 5-6–5-11
first
  DDL 4-10
Function
  defining 5-25–5-27

G

gdef
  example 2-3
  modifying databases 5-60
  overview 2-2
  relationship to DYN 2-2
  reserved words 3-2
  syntax 2-2
  troubleshooting 2-4
gds
  date routines 5-9
gds_$decode_date 5-9
gds_$encode_date 5-9
Global field
  modifying 5-62

I

in
  DDL 4-16
inactive 5-29
Index
  active/inactive 5-29
  ascending 5-29
  defining 5-28
  descending 5-29
  duplicates 5-29
  unique 5-28–5-29
input_type 5-23
InterBase
  fax number A-2

J

Joining relations
  overview 4-14

L

Language
  DDL options 2-3

M

matching
  DDL 4-6
max (maximum) 4-19
min (minimum) 4-19
missing
  DDL 4-7
Missing values
  assigning 5-16
  DDL 4-7
modify database
  DDL 5-59–5-61
modify field
  DDL 5-62–5-64
modify index
  DDL 5-65–5-66
modify relation
  DDL 5-67–5-70
modify trigger
  DDL 5-71–5-73
modify view
  DDL 5-74–5-75

Multi-file database
  primary file 5-2
  secondary file 5-2

N

not
  DDL 4-3
null
  DDL 4-11
Null values
  DDL 4-11
Numeric field characters 5-15
numeric literal
  DDL 4-12
Numeric literal expression
  DDL 4-12

O

or
  DDL 4-3
output_type 5-23

P

Page size
  overriding default 5-3
Primary key 5-28

Q

query_header 5-17
query_name 5-18
quoted string
  DDL 4-13

R

Record selection expression
  DDL 4-14
reduced to
  DDL 4-15–4-16
Relation
  defining in DDL 5-31–5-36
Relation clause 4-16
Remote database
  accessing 5-3
Reserved words, **gdef** 3-2
RSE, see Record selection expression

S

Scale of numerics 5-9
Security
  access privileges 5-38
  assigning to an object 5-19
  defining 5-37
  hierarchy 5-38
security_class
  DDL 5-19–5-20, 5-32
Shadowing
  defining 5-41–5-42
  multi-file 5-41
starting with
  DDL 4-7
Statistical expressions 4-19
Statistical functions
  DDL 4-19
Substring 4-6
Subtype 5-10
Supported datatypes 5-8

T

Trigger
  defining 5-43–5-48
  language usage 5-45–5-46

U

unique 5-28–5-29
  DDL 4-8
User defined function
  defining 5-25–5-27

V

valid_if
  DDL 5-20–5-21
value
  DDL 4-21

Value expressions 4-21
arithmetic expression 4-2
between condition 4-4
boolean expression 4-3
comparison condition 4-5
containing condition 4-6
matching condition 4-6
starting with condition 4-7
statistical expression 4-19
username expression 4-20
Varying datatype 5-11
View
defining in DDL 5-49–5-53
modifying 5-74–5-75

W

Wildcard characters 4-6
with
DDL 4-17–4-18