InterBase
Programmer's 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.

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

This book describes the syntax for each InterBase GDML and SQL expressions, statements, declarations, and commands.

Who Should Read this Book

You should read the *Programmer's Reference* guide if you are an applications programmer who wants to program GDML or SQL commands against an InterBase database. You should also read this guide if you are an end-user who wants to use interactive GDML or SQL to query an InterBase database using **qli**. This book is a companion to the *Programmer's Guide* and assumes you have read that book, or you are experienced with InterBase.

Using this Book

This book contains the following chapters:

Chapter 1	Introduces the book.
Chapter 2	Lists the punctuation and symbols the Interbase preprocessor **gpre** recognizes. Also lists reserved words for GDML and SQL.
Chapter 3	Contains entries for each GDML expression.
Chapter 4	Contains entries for each GDML statement, declaration, command and clause.
Chapter 5	Contains entries for each InterBase SQL expression.
Chapter 6	Contains entries for each InterBase supported SQL statement and command.
Appendix A	Discusses error handling in GDML, SQL and DSQL 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.

italics

Used for chapter and manuals titles; to identify 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}	Indicate an alternative item: • option::= **{vertical | horizontal | transparent}**
[brackets]	Indicate 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 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 syntax variable: • **`create_blob`** *`blob-variable`* **`in`** *`dbfield-expression`*
|	A vertical bar separates items in a list of choices.
⇓	A down arrow 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.

In addition, platform-specific installation instructions are available for all supported platforms.

Chapter 1
Introduction

The *Programmer's Reference* has information about all InterBase GDML and SQL expressions, statements, clauses and declarations.

Overview

Each expression and statement has the following sections:

- Function, which describes what the expression or statement does
- Syntax, which provides a complete diagram of the expression or statement and its options
- Options, which describes each option of the expression or statement
- Example, which shows how to use the expression or statement in a program
- Troubleshooting, which list error messages and suggests corrective actions
- See Also, which refers you to related expressions or statements, 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 GDML or SQL statement or expression.

Chapter 2
Reserved Words

This chapter lists the punctuation and symbols that the Interbase preprocessor **gpre** recognizes. It also lists reserved words for GDML and SQL.

Overview

Reserved words are words defined in GDML or SQL for special purposes. They cannot be used as user-declared identifiers.

Preprocessor Punctuation and Symbols

Gpre recognizes the following punctuation and symbols:

&	->	| |
&&	++	--
>	>=	^<
<	<=	^>
<>	!=	^=
~=	==	
:	,	{}
[]	()	;
/	*	\
+		

Preprocessor Words

Gpre recognizes the following reserved words:

ADD	ALL	ALLOCATION
ALTER	AND	AS
ANY	ASC	ASCENDING
AT	AVERAGE	AVG
BASED	BASED_ON	BEGIN
BETWEEN	BT	BY
BUFFERCOUNT	BUFFERSIZE	CASE
CHAR	CLOSE	CLOSE_BLOB
COMMENT	COMMIT	COMMIT_TRANSACTION
COMPILETIME	COMPILE_TIME	CANCEL_BLOB
CONCURRENCY	CONNECT	CONSISTENCY
CONTAINING	CONTINUE	COUNT
CREATE	CREATE_BLOB	CROSS
CURRENT	CURSOR	DATABASE
DATE	DBA	DECIMAL
DECLARE	DELETE	DESC
DESCENDING	DESCRIBE	DISTINCT
DOUBLE	DROP	ELSE
END	END_ERROR	END_EXEC
END-EXEC	END_FETCH	END_FOR
END_MODIFY	END_STORE	END_STREAM
EQ	ERASE	ERROR
EXCLUSIVE	EXEC	EXECUTE
EXISTS	EXTERN	EXTRACT
FETCH	FILENAME	FINISH
FINISH_DATABASE	FIRST	FLOAT
FOR	FORWARD	FOUND
FROM	FUNCTION	GE
GET_SEGMENT	GO	GRANT
GROUP	GT	HAVING
IN	INCLUDE	INDEX
INSERT	INTEGER	INTO
IS	LE	LENGTH
LEVEL	LIKE	LOCK
LONG	LT	MAIN
MATCHES	MATCHING	MAX

MIN
MISSING
MODIFY
NE
NO_WAIT
NOWAIT
NOT
NULL
OF
ON
ON_ERROR
OPEN (add option to list)
OPEN_BLOB
OPTION
OR
ORDER
OVER
PAGESIZE
PATHNAME
PREPARE
PREPARE_TRANSACTION
PROC
PROCEDURE
PROTECTED
PUBLIC
PUT_SEGMENT
RDB$DB_KEY
READ
READ_ONLY
READ_WRITE
READY
READY_DATABASE
REDUCED
RELEASE
RELEASE_REQUESTS
REM
REQUEST_HANDLE
RESERVING
RESOURCE
REVOKE
ROLLBACK
ROLLBACK_TRANSACTION
RUN
RUNTIME
SCALE
SCHEDULE
SECTION
SEGMENT
SELECT
SET
SHARED
SHORT
SMALLINT
SORTED
SQL
SQLCODE
SQLERROR
SQLWARNING
START_STREAM
START_TRANSACTION
STARTING
STARTING_WITH
STATEMENT
STATIC
STOGROUP
STORE
STRING
SUB
SUBROUTINE
SUM
SYNONYM
TABLE
TABLESPACE
TO
TOTAL
TRANSACTION_HANDLE
UNION
UNIQUE
UPDATE

Chapter 3
GDML Expressions

This chapter contains entries for GDML expressions.

Overview

GDML supports the following expressions:

- Boolean expression, which evaluates to true, false, or missing
- Record selection expression, which specifies the search and delivery conditions for record retrieval
- Value expression, a symbol or string of symbols from which InterBase calculates a value

Boolean Expressions

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*).

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

Syntax

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

conditional-expression ::=
{any-condition| between-condition|
comparison-condition|containing-condition
matching-condition|matching using condition|
missing-condition|not-condition|
starting-condition|unique-condition}
```

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

- Any condition
- Between condition
- Comparison condition
- Containing condition
- Matching condition
- Matching using condition
- Missing condition
- Not condition
- Starting 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 RSE includes at least one record. If you add **not**, the expression is true if there are *no* records in the record stream.

You might want to use **any** instead of joining records if all you want to do is establish a record exists. As soon as InterBase finds one record that meets the search criteria, it stops, whereas a join would continue until it found all qualifying records

Syntax

```
[not] any rse
```

Example

The following statement prints the name of any state for which there are cities stored:

```
for s in states with any c in cities
   with c.state = s.state
      printf ("%s\n", s.state_name);
end_for;
```

Between Condition

Function

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

Syntax

```
value-expression-1[not]between
value-expression-2 and value-expression-3
```

Options

value-expression-1
Specifies an expression for which to test.

value-expression-2
value-expression-3
Sspecify the lower and upper inclusive boundaries.

Example

The following statement looks for cities with populations between 100,000 and 250,000:

```
for c in cities with c.population
   between 100000 and 250000
      printf ("%s\t%s\t%d\n", c.city, c.state,
            c.population);
end_for;
```

Comparison Condition

Function

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

Syntax

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

Options

relational-operation

Any of the operators in the following table.

Table 3-1. Relational Operators

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 statement looks for cities with populations less than 100,000:

```
for c in cities with c.population < 100000
    printf ("%s\t%s\t%d\n", c.city, c.state,
          c.population);
end_for;
```

Containing Condition

Function

The **containing** condition tests for the presence of *string* (case-*in*sensitive) anywhere in *value-expression*. It evaluates to true if *string* is contained in *value-expression*. If the value of *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
value-expression-2
```

Options

value-expression-1
Specifies an expression for which the substring search occurs.

value-expression-2
Specifies an expression for which the substring search occurs.

Examples

The following statement looks for cities with the substring "ville" somewhere in their name:

```
for c in cities with c.city containing 'ville'
   printf ("%s\t%s\n", c.city, c.state);
end_for;
```

The following fragment looks for a COMMENTS entry in the CROSS_COUNTRY relation that contains the substring "varied":

```
for c in cross_country with c.comments
   containing 'varied'
      printf ("%s\t%s\t%s\n\n", c.area_name,
            c.city, c.state);
         for blob in c.comments
            blob.segment[blob.length]=0;
            printf("%s", blob.segment);
         end_for;
   printf ("\n");
end_for;
```

Matching Condition

Function

The **matching** condition tests for the presence of *wildcarded-string*, a string containing the wildcard characters * and **?**. The asterisk matches an unspecified run of characters, while the question mark matches a single character. This Boolean test is case *in*sensitive.

Syntax

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

Options

value-expression-1
Specifies an expression for which the wildcard substring search occurs.

value-expression-2
Specifies an expression for which the substring search occurs.

Examples

The following example looks for states with the state abbreviation equal to "N" followed by exactly one character:

```
for c in cities with c.city matching 'N?'
   printf ("%s\t%s\n", c.city, c.state);
end_for;
```

The following example looks for cities that have "ton" somewhere in their names:

```
for c in cities with c.city matching '*ton*'
   printf ("%s\t%s\n", c.city, c.state);
end_for;
```

Matching Using Condition

Function

The **matching using** condition lets you define your own wild-card search characters.

Syntax

```
matching value-expression using 'control-string'

control-string ::= [prequalifier][definition-
commalist][postqualifier]

prequalifier ::= [-S(|+S(]

definition ::= wildcard =definition-character
               [definition-character...]
postqualifier ::= [)]
```

Options

value-expression
Specifies the expression for which the substring search occurs.

prequalifier
The prequalifier string **-S(** disables case sensitivity of the *value-expression* in the **matching** clause. The prequalifier string **+S(** enables case sensitivity of the *value-expression* in the **matching** clause.

definition
Specifies the punctuation or symbol character that you want to define and sets it equal to one or more of the characters in the following table:

Definition Character	Operation
?	Matches any single character.
[]	Defines a class of character.
*	Modifies previous definition or class: indicates zero or more occurrences.
@	Treats the next character as literal.
~	Excludes the following character or class of characters.

A class of characters can be a list or range of characters that you specify inside the square brackets. For example, the range

[0-9] or the list [0123456789] represents any numeral. If you define a class with &=[0-9A-Za-z], the ampersand represents all alphanumeric characters. You can use the tilde only as the first character in a class definition. The class definition of [~0-9] means any non-numerals.

postqualifer
The postqualifier) is optional.

Example

The following example searches for cities that have "ton" somewhere in their names. The **matching using** clause defines "+" as zero or more occurrences of any single character:

```
for c in cities with c.city matching '+ton+' using
   '+=?*' print city
```

Missing Condition

Function

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

Unless you specify otherwise in the field's definition, blanks are returned for numbers, characters, and dates, and nothing is returned for blobs.

Table 3-2.

Syntax

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

Example

The following statement looks for states that have a missing value for the CAPITAL field:

```
for s in states with s.capital missing
   printf ("%s\n", s.state_name);
end_for;
```

Not Condition

Function

The **not** condition negates a Boolean expression. The syntax diagrams for each of the Boolean expressions includes the correct position for the **not**.

Syntax

```
not value-expression
```

Example

The following statement looks for cities with populations not between 100,000 and 250,000:

```
for c in cities with c.population not between
      100000 and 250000
   printf ("%s\t%s\t%d\n", c.city, c.state,
         c.population);
end_for;
```

Starting Condition

Function

The **starting** condition tests for the presence of *string* (case-sensitive) at the beginning of *value-expression*. It evaluates to true if the first characters of *value-expression* match *string*. The search is case-sensitive.

Syntax

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

Options

value-expression-1
Specifies an expression at the start of which the substring search is to occur.

value-expression-2
Specifies an expression for which the substring search occurs.

Example

The following statement looks for cities that start with the string "New":

```
for c in cities with c.city starting with 'New'
   printf ("%s\t%s\n", c.city, c.state);
end_for;
```

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.

Syntax

```
[not]unique rse
```

Example

The following query prints the names of states that have only one ski area:

```
for s in states with unique ski in ski_areas with
      ski.state = s.state
   printf ("%s\n", s.state_name);
end_for;
```

Troubleshooting

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

See Also

See the chapter on defining fields in the *Data Definition Guide* for more information about defining alternate missing values.

See the entries in this chapter for:

- Value expression
- Record selection expression

Record Selection Expression

Function

The RSE (record selection expression) clause specifies the search and delivery conditions for record retrieval.

Syntax

```
[first-clause] record-source [with-clause]
[reduced-clause] [sorted-clause]

record-source ::= {relation-clause | cross
                   source}

relation-clause ::= [context-variable in]
                     relation-name

cross-source ::= relation-clause cross
                 record-source
```

The following sections describe the clauses of the record selection expression:

- First clause
- Relation clause
- Cross clause
- With clause
- Reduced clause
- Sorted clause

Troubleshooting

See the discussion of errors and error handling in the chapter on getting started with GDML in the *Programmer's Guide*. See also the Appendix for a list of errors.

See Also

See the entries in this chapter for:

- Boolean-expression
- Value expression

First Clause

Function

The *first-clause* limits the records in a stream to the number you specify with an integer. The format of the *first-clause* follows.

Syntax

```
first integer
```

Options

integer
Specifies the number of records to select. Any fractional portion of the integer is truncated. Unless you sort the record stream when you use the *first-clause*, *integer* random records are returned.

Example

The following query uses a *first-clause*, a *relation-clause*, and a *sorted-clause* to display the two youngest states:

```
for first 2 s in states sorted by descending
s.statehood
   printf ("%s was admitted to the Union on %s\n",
      s.state_name, s.statehood);
end_for;
```

Relation Clause

Function

The *relation-clause* identifies the target relation. The format of the *relation-clause* follows.

Syntax

```
context-variable in [database-handle]
relation-name
```

Options

context-variable
The context variable is used for name recognition and is associated with a relation. A context variable can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

Except for C programs, **gpre** is not sensitive to the case of the context variable. For example, it treats "B" and "b" as the same character. For C programs, you can control case sensitivity of context variables with the **either_case** option, when you preprocess your program.

database-handle
The optional *database-handle* identifies the database for multiple database access.

relation-name
Identifies the relation to use in the *relation-clause.*

Example

The following query declares a context variable for the SKI_AREAS relation:

```
for ski in ski_areas reduced to ski.state
   printf ("%s\n", ski.state);
end_for
```

Cross Clause (Join)

Function

The *cross-clause* performs a join operation. It joins records from two or more different relations in the same database. The relationship can be:

- An equijoin, which is based on the equality of common fields
- A non-equijoin, which is based on inqualities
- A cross product, where no relationship exists

Unlike most other clauses of the record selection expression, the *cross-clause* can be repeated to include as many relations as are necessary.

Syntax

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

Options

relation-clause
Identifies the target relation.

over *field-name-commalist*
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 field.

Examples

The following query uses two *relation clauses* and a *cross clause* to list a SKI AREA, CITY, and STATE in which the ski area is located:

```
for s in states cross ski in ski_areas over state
   printf ("%s\t%s\t%s\n", ski.name, ski.city,
         s.state_name);
end_for;
```

The following query does the same thing as the preceding query, but uses an explicitly qualified join condition in place of the **cross** shortcut:

```
for s in states cross ski in ski_areas with
      s.state = ski.state
   printf ("%s\t%s\t%s\n", ski.name, ski.city,
      s.state_name);
end_for;
```

The following statement displays the names of cities that are larger than the capitals of their states:

```
for 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
   sorted by s.state, c.city
   printf ("%s, %s is larger than %s\n", c.city,
         s.state_name,s.capital);
end_for;
```

The following statement displays the names of states in which the capital is not the largest city:

```
for 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
   sorted by s.state
   reduced to s.state, s.capital
   printf ("%s contains cities larger than %s\n",
         s.state_name,s.capital);
end_for;
commit;
finish;
```

With Clause (Select)

Function

The *with-clause* specifies a search condition or combination of search conditions.

When you specify a search condition, InterBase evaluates the condition for each record that might possibly qualify. InterBase compares 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 is true and that record becomes part of the record stream.

Syntax

```
with Boolean-expression
```

Options

Boolean-expression
Specifies the predicate used in selecting records.

Example

The following query uses a *with-clause* to limit the display cities in Texas for which the value of the POPULATION field is not missing:

```
for c in cities with c.state = 'TX' and
c.population not missing
   printf ("%s\t%d\t%d\n", c.city, c.population,
          c.altitude);
end_for;
```

Reduced Clause (Project)

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, InterBase considers a list of fields and eliminates records that do not have a unique combination of values for the listed fields. When you reduce a record stream, you can only reference fields that were mentioned in the **reduced** clause.

Syntax

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

Options

dbfield-expression
Specifies the field on which you want to project the result.

context-variable.field-name
Qualifies the database field for multi-relation operations.

Example

The following query uses a *reduced-clause* to list the states in which there are ski areas:

```
for ski in ski_areas reduced to ski.state
   printf ("%s\n", ski.state);
end_for;
```

Sorted Clause

Function

The *sorted-clause* orders the output, returning the record stream sorted by the values of one or more sort keys.

Syntax

```
sorted [by] sort-key-commalist

sort-key ::= [ascending|descending]
[anycase|exactcase] db-field-expression

db-field-expression ::= context-variable.field-
name
```

Options

sort-key
Specifies the field on which you want to sort. You can sort a record stream alphabetically, numerically, by date, and by any combination of these. The *sorted-clause* lets you have as many sort keys as you want. The greater the number of sort keys, the longer it takes for InterBase to execute the query.

ascending | descending
For each sort key, you can specify whether the sorting order of the sort key is **ascending** or **descending**. If you have more than one sort key, the sorting order you specify cascades down the list; that is, if you do not specify whether a particular sort key in the list is **ascending** or **descending**, **gpre** assumes

that you want the order specified for the most recent key. Therefore, if you list several sort keys, but only include the keyword **descending** for the first key, InterBase sorts all keys in descending order. The default order is **ascending**.

anycase | **exactcase**
For each sort key, you can specify whether the sorting order of the sort key is **anycase** or **exactcase.** If you have more than one sort key, the case you specify cascades down the list; that is, if you do not specify whether a particular sort key in the list is **anycase** or **exactcase**, **gpre** assumes you want the case specified for the most recent key. The default case is **exactcase.**

`context-variable.field-name`
Qualifies the database field for multi-relation operations.

Examples

The following query uses a *first-clause*, a *relation-clause*, a *sorted-clause* and the **descending** keyword to display the two youngest states:

```
for first 2 s in states sorted by descending
s.statehood
   printf ("%s was admitted to the Union on %s\n",
      s.state_name, s.statehood);
end_for;
```

The following query uses a *sorted-clause* and the **exactcase** keyword to display a list of names. If you had an employee with a last name input as CASEY and another with a last name input as Casey, CASEY would appear first:

```
for employees sorted by exactcase lname
   print fname, lname
```

Troubleshooting

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

See Also

See the entries in this chapter for:

- Boolean 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 statement in which the expression appears.

Syntax

```
value-expression::= {arithmetic-expression |
dbfield-expression | numeric-literal-expression
| quoted-string-expression | username-expression
| (value-expression) | - value-expression | host-
language-variable }
```

The following sections describe the expressions of the value expression:

- Arithemetic expression
- Database field expression
- Numeric literal expression
- Quoted string expression
- Username expression

Troubleshooting

See the Appendix for a listing of errors and error handling.

See Also

See the entries in this chapter for:

- Boolean expression
- Record selection expression

Arithmetic Expression

Function

The *arithmetic-expression* combines value expressions and arithmetic operators. The format of the *arithmetic-expression* follows.

Syntax

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

You can add (+), subtract (-), multiply (*), or divide (/) value expressions in record selection expressions. Arithmetic operators

are evaluated in the normal order (addition, subtraction, multiplication, division). Use parentheses to change the order of evaluation.

Example

The following statement uses a database field expression to display the city and an arithmetic expression to calculate and display the altitude in meters:

```
for c in cities with c.altitude * 0.3048 > 500
   printf ("%s\n" c.city)
end_for;
```

Database Field Expression

Function

The *dbfield-expression* references database fields. This expression can occur in several clauses of record selection expressions and Boolean expressions.

Syntax

context-variable.field-name[**.null** | .*datatype*]

Options

context-variable.field-name

Qualifies the database field for multi-relation operations. Declare the context variable for a relation in the *relation clause* of the record selection expression.

.null

Allows access to the null flag for the field. If you reference the null flag in a store or modify statement, you must set it explicitly. If the null flag remains true (that is, non-zero) the field is stored as missing, even if you supply a value.

.*datatype*

Lets you "cast" a database field with a datatype other than that with which it is stored. **Gpre** automatically takes care of datatype conversion, but you can "convert" a field for the duration of a request to the datatype of your choice.

Example

The following statement uses a database field expression to display the city and an arithmetic expression to calculate and display the altitude in meters:

```
for c in cities with c.altitude * 0.3048 > 500
   printf ("%s\n" c.city)
end_for;
```

Numeric Literal Expression

Function

The *numeric-literal-expression* represents a decimal number.

Syntax

```
numeric-string[.numeric-string]
```

Options

numeric-string
A string of digits with an optional decimal point.

Example

The following statement uses a database field expression to display the city, an arithmetic expression to calculate and display the altitude in meters, and a numeric literal expression used in the arithmetic operation:

```
for c in cities with c.altitude * 0.3048 > 500
   printf ("%s\n" c.city)
end_for;
```

Quoted String Expression

Function

The *quoted-string-expression* specifies a string of ASCII characters enclosed in single (') or double (") quotation marks, depending on host language requirements.

Syntax

```
"string"
```

Options

string
Any of the ASCII characters in the following table:

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

Example

The following statement uses database field expressions to display the city and state, an arithmetic expression to calculate and display the altitude in meters, a numeric literal expression used in the arithmetic operation, and two quoted string expressions to anglicize the C **printf** display:

```
for c in cities cross s in states over state
   printf ("%s, %s is situated at %f meters above
         sea level.\n",
      c.city, s.state_name, c.altitude * 0.3048 );
end_for;
```

Username Expression

Function

The *username-expression* is a value expression that automatically picks up the username or login of the person running the program. Combined 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
```

Options

rdb$user_name
A value expression to which is assigned the username or login. This expression can only be used in RSEs and cannot be qualified with a context variable.

Example

The following statement picks up the username and uses an RSE that selects records based on the value of the USER_NAME field:

```
for e in employees with
   e.user_name = rdb$user_name
      printf ("%s, %s\n", e.emp_name,
e.user_name);
end_for;
```

Troubleshooting See the list of error messages in the Appendix.

See Also See the entries in this chapter for:

- Boolean expression
- Record selection expression

Chapter 4
GDML Statements, Commands, Declarations and Clauses

This chapter contains entries for GDML statements, commands, declarations and clauses.

Overview

To manipulate data in an InterBase database, you can use the following statements, commands, declarations and clauses:

based_on declaration	cancel_blob	case_menu
close_blob	commit	create_blob
database declaration	display	erase
event_init	event_wait	fetch
finish	for	for blob

for form
for_item
for_menu
get_segment
modify
on_error
open_blob
prepare
put_item
put_segment
ready
release_requests
request_options
rollback
save
start_stream
start_transaction
store
store blob
transaction handle

Based_On

Function

The **based_on** declaration declares a program variable by referencing a database field. The preprocessor supplies the host variable with all the attributes defined for the database field.

Syntax

```
variable based_on dbhandle relation-name.field-name
```

Options

variable
Names a host language variable that inherits the characteristics of a database field.

In Pascal, you cannot use the **based_on** clause in a parameter list for a routine. Instead, declare a type and then declare the formal parameter to be that type.

dbhandle
Specifies the source of the database field. The database handle must have been declared in an earlier **database** statement.

relation-name.field-name
Specifies the relation and field on which to base the host variable.

Examples

The following example shows two **based_on** declarations as they would appear in a C program:

```
based_on states.state_name state_name;
based_on states.capitol capitol_city;
```

The following example shows the **based_on** declaration as it would appear in a Pascal program:

```
var state : based_on states.state;
```

Troubleshooting

See the Appendix for a discussion of error handling and a listing of errors.

See Also

Host language documentation for information on the declaration of variables.

Cancel_Blob

Function

The **cancel_blob** statement releases internal storage used by a discarded blob and sets the blob handle to null.

When you create a blob, InterBase temporarily stores it in the database. If you fail to close the blob, the temporary storage space remains allocated.

Because this statement does *not* produce an error if the handle is null, it is good practice to call this routine before you open or create a blob. This practice ensures that the InterBase cleans up earlier blob operations. If you abort a blob operation, you should use a **cancel_blob** statement before opening or creating another blob.

Syntax

```
cancel_blob blob-variable [on_error]

on_error ::= on_error statement...end_error
```

Options

blob-variable
A temporary name used for name recognition. It is associated with individual segments in the field and is used very much like a context variable. You must have assigned the blob variable in an earlier **create_blob** or **open_blob** statement.

on_error
Specifies the action to be performed if an error occurs during the cancel operation.

statement
A host language or GDML statement. If you include more than one *statement*, you must separate them using the host language convention.

Example

The following example creates a record stream, creates a blob field, and writes segments to the blob field. It uses a **cancel_blob** statement to release internal storage once processing has ended:

```
store tour in tourism using
   printf ("Enter state code: ");
   gets (tour.state)
```

```
        printf ("Enter zip code: ");
        gets (tour.zip)
        printf ("Enter city: ");
        gets (tour.city)
        create_blob b in tour.guidebook;
        printf ("Enter new blurb one line at a time\n");
        printf ("A line containing '-30-' ends input");
        gets (b.segment);
        while (strcmp(b.segment, "-30-")) {
              for (i=strlen(b.segment);i>=0;i--){
                 if (b.segment[i] != ' ') {
                 b.segment[i+1]='\n';
                 b.segment[i+2]=0;
                 b.length = i+1;
                 i=0;
                 }
                 }
              put_segment b;
              gets (b.segment);
           }
        printf("Do you swith to continue? ")
        gets(answer)
        if (answer[0] =='y' || answer [0] == 'Y'
        close_blob b
        else
        cancel_blob b
end_store;
```

Troubleshooting

See the Appendix for a discussion of error handling and a listing of errors.

See Also

See the entries in this chapter for:

- **open_blob**
- **create_blob**
- **on_error**

Case_Menu

Function

The **case_menu** statement displays a menu in the forms window and executes the code associated with the user's choice.

Syntax

```
case_menu [(options)] title-string menu-entrees
end_menu

menu-entrees::= (menu_entree entree-string}...

options::= {vertical | horizontal | transparent]
```

Options

title-string

A quoted string that provides the title line for the menu.

menu_entree

Establishes a line that appears in a menu and introduces a block of code that executes if the line is chosen:

- All code between the keywords **case_menu** and **end_menu** must be introduced by **menu_entree** labels.
- To specify an option to continue without taking any action, include a null statement under the **menu_entree** label.

Because the **case_menu** statement is like a Pascal **case** statement, and not like a C **switch** statement, choosing a menu item executes only the code between that item and the next item or **end_menu**.

entree-string

A quoted string that becomes a line in a vertical menu or a selection item in a horizontal menu.

vertical

Displays the menu choices in a vertical format. This display option is the default. A vertical menu overwrites the contents of the current form with its menu choices.

horizontal
Displays the menu choices in a horizontal format. A horizontal menu, also called a "tag-line menu," displays the menu choices on the bottom line of the menu.

transparent
Displays the menu choices vertically, obscuring only those parts of the form directly behind the menu.

Example

The following example cycles through the atlas database, displaying a menu for each displayed state and offering the user a chance to update the state's capital or exit from the iteration:

```
#include <stdio.h>

database db = "atlas.gdb";

#define CONTINUE 0
#define STOP 1
main()
{
intflag;

flag = CONTINUE;

for form x in show_state
   for s in states sorted by s.statehood
      strcpy (x.state_name, s.state_name);
      x.statehood = s.statehood;
      x.area = s.area;
      strcpy (x.state, s.state);
      strcpy (x.capital, s.capital);
      display x displaying *;

         case_menu (transparent) "Alter State?"
            menu_entree "No Changes":
            ;
            menu_entree "Change Capital":
            display x accepting capital
            cursor on capital waking on capital;
            if (x.capital.state ==
            PYXIS_$OPT_USER_DATA)
            modify s
            strcpy (s.capital, x.capital);
```

```
                end_modify;
                menu_entree "Exit" :
                flag = STOP;
             end_menu;

          if (flag == STOP)
             break;
      end_for;
end_form;
delete_window;
finish;
}
```

Troubleshooting See also the Appendix for a listing of errors and a discussion of error handling.

See Also See the entries in this chapter for:

- display
- for form

Close_Blob

Function

The **close_blob** statement closes an open blob field and releases system resources associated with blob retrieval or update.

You should close the blob as soon as you finish reading or writing. If you fail to close a blob to which you wrote data, you are not able to make the blob permanent. Closing a blob is especially important when you access remote databases. Because InterBase's remote interface buffers segment transfer between participating nodes, it may truncate the last segment you write unless you explicitly signal that the blob is closed.

Once you close a blob, you cannot read from or write to that blob without re-opening it with an **open_blob** or **for blob** statement.

Syntax

```
close_blob blob-variable [on-error]
     on-error: := on_error statement... end_error
```

Options

blob-variable
A temporary name used for name recognition. It is associated with individual segments in the field and is used very much like a context variable. You must have assigned the blob variable in an earlier **create_blob** or **open_blob** statement.

on_error
Specifies the action to be performed if an error occurs during the close operation.

statement
A host language or GDML statement. If you include more than one *statement*, you must separate them using the host language convention.

end_error
Terminates the **on_error** clause.

Example

The following program creates a record stream from two relations, opens a blob field, reads segments from the blob field, and then closes the blob field:

```
/* program update_guide */
```

```
#include <stdio.h>

database atlas = filename 'atlas.gdb';

main()
{

ready atlas;
start_transaction;
for s in states cross t in tourism over state
   sorted by s.state
      printf ("%s %s\n", s.state_name, t.city);
      open_blob b in t.office
      get_segment b;
      while ( (gds_$status [1] == 0) ||
         (gds_$status [1] == gds_$segment) ) {
            b.segment[b.length]=0;
            printf ("%s", b.segment);
            get_segment b;
         }
      printf("\n");
      close_blob b;
end_for;
commit;
finish atlas;
}
```

Troubleshooting

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

See Also

See also the entries in this chapter for:

- **open_blob**
- **on_error**

For some guidance on the best approach to processing blobs, see the chapter on using blobs in the *Programmer's Guide*.

Commit

Function

The **commit** command ends a transaction and makes the transaction's changes visible to other users.

The **commit** commit affects all databases in the transaction, writing to the database(s) all changes made during the transaction. It flushes all modified buffers and closes any record streams that are open.

Syntax

```
commit [transaction-handle] [on-error]
     on-error::= on_error statement... end_error
```

Options

transaction-handle
Specifies the transaction you want to commit. If the transaction you want to commit has a transaction handle associated with it, you must use that handle when you commit the transaction. If you do not specify a transaction handle on a **commit** command, InterBase commits the "default" transaction. The default transaction is what InterBase starts when you use a **start_transaction** command without a handle.

on_error
Specifies the action to be performed if an error occurs during the commit operation.

statement
A host language or GDML statement. If you include more than one *statement*, you must separate them using the host language convention.

end_error
Terminates the **on_error** clause.

Examples

The following example starts an unnamed transaction, performs some unspecified data manipulation, and then writes the changes to the database:

```
start_transaction concurrency;
   ⇓
commit;
```

The following C program starts two separate transactions, one to get a badge number, and the other to store a new employee. This simplified program does not contain error handling.

```
/* program map */

#include <stdio.h>
#include <ctype.h>

database db = filename 'emp.gdb';

based on badge_num.badge badge_type;

int   store_emp_tr,
        to_be_stored;

char   store_num [8];
char   super_id[8];

main()
{
ready;
   start_transaction store_emp_tr;
   printf ("Enter the number of new employees: " );
   gets (store_num);
   to_be_stored = atoi(store_num);
   while (to_be_stored > 0) {
      store (transaction_handle store_emp_tr) e in
      employees using
         e.badge = get_badge();
         printf ("Enter first name: ");
         gets (e.first_name);
         printf ("Enter last name: ");
         gets (e.last_name);
         printf ("Enter supervisor's id: ");
         gets (super_id);
         e.supervisor = atoi(super_id);
         printf ("Enter department: ");
         gets (e.department);
      end_store;
      to_be_stored--;
      }
   commit store_emp_tr;
   finish;
```

```
}

get_badge ()
{
   int   get_badge_tr;

   get_badge_tr = 0;
   start_transaction get_badge_tr;

   for (transaction_handle get_badge_tr) b in
   badge_num
      badge_type = b.badge;
      modify b using
         b.badge = b.badge + 1;
      end_modify;
   end_for;
   commit get_badge_tr;
   return badge_type;
}
```

The following example is the Pascal version of the preceding program:

```
program map (input_output);

database db = filename 'emp.gdb';

type  badge_type = based on badge_num.badge;
var
   store_emp_tr: gds_$handle:= nil;
        to_be_stored: integer;
function get_badge : badge_type;
var
   get_badge_tr: gds_$handle;
begin
        get_badge_tr := nil;
   start_transaction get_badge_tr;
   for (transaction_handle get_badge_tr) b in
   badge_num
      get_badge := b.badge;
      modify b using
         b.badge := b.badge + 1;
      end_modify;
   end_for;
```

```
   commit get_badge_tr;
end;  { function get_badge }

begin
ready;
   start_transaction store_emp_tr;
   write ('Enter the number of new employees: ' );
   readln (to_be_stored);
   while to_be_stored > 0 do
   begin
      store (transaction_handle store_emp_tr) e in
         employees using
            e.badge := get_badge;
            write ('Enter first name: ');
            readln (e.first_name);
            write ('Enter last name: ');
            readln (e.last_name);
            write ('Enter supervisor''s id: ');
            readln (e.supervisor);
            write ('Enter department: ');
            readln (e.department);
      end_store;
                to_be_stored := to_be_stored - 1;
   end;
commit store_emp_tr;
finish;
end.
```

Troubleshooting

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

See Also

See the entries in this chapter for:

- **start_transaction**
- **transaction-handle**
- **on_error**

See also the chapter on transactions in the *Programmer's Guide*.

Create_Blob

Function

The **create_blob** statement creates a blob.

Syntax

```
create_blob blob-variable in dbfield-expression
[from subtype to subtype][on-error]
on-error::= on_error statement... end_error
```

Options

blob-variable
A temporary name used for name recognition. It is associated with individual segments in the field and is used much like a context variable.

dbfield-expression
A value expression that identifies a field containing blob data.

from *subytpe* **to** *subtype*
Specifies the pre-defined subtype a blob filter converts from and the pre-defined subtype it converts to.

on_error
Specifies the action to be performed if an error occurs during the create operation.

statement
A host language or GDML statement. If you include more than one *statement*, you must separate them using the host language convention.

end_error
Terminates the **on_error** clause.

Example

The following example creates a blob field and writes segments to the blob field:

```
database db = 'atlas.gdb';
main()
{
int i;

ready;
start_transaction;
store tour in tourism using
```

```
    printf ("Enter state code: ");
    gets (tour.state);
    printf ("Enter city name: ");
    gets (tour.city);
    printf ("Enter zip code: ");
    gets (tour.zip);
    create_blob b in tour.guidebook;
    printf ("Enter new blurp one line at a\
       time.\n");
         printf ("Terminate with <EOF>.\n");
    while (gets(b.segment) != 0) {
       for (i=strlen(b.segment);i>=0;i--){
          if (b.segment[i] != ' ') {
             b.segment[i+1]='\n';
             b.segment[i+2]=0;
             b.length = i+1;
             i=0;
             }
          }
          PUT_SEGMENT;
       }
    close_blob b;
end_store;
commit;
finish;
}
```

The following example is the Pascal version of the preceding program:

```
begin
ready;
start_transaction;
store tour in tourism using
    write ('Enter state code: ');
    readln (tour.state);
    write ('Enter city name: ');
    readln (tour.city);
    write ('Enter zip code: ');
    readln (tour.zip);
    create_blob b in tour.guidebook;
    writeln ('Enter new blurp one line at a time.');
         writeln ('Terminate with <EOF>.');
    repeat
```

```
        begin
           readln (input, b.segment);
           for i := sizeof (b.segment) downto 1 do
              if b.segment[i] <> ' ' then exit;
           i := i + 1;
           b.segment [i] := chr(10);
           b.length := i;
           put_segment b;
        end;
        until eof (input);
        reset (input);
        close_blob b;
end_store;
commit;
finish;
end.
```

Troubleshooting See the Appendix for a discussion of errors and error handling.

See Also See also the entry for **on_error**

See also the entry in Chapter 3 for value expression.

See also the chapter on using blobs in the *Programmer's Guide*.

Database

Function

The **database** declaration specifies the database to be accessed by a program or program module. Because the **database** declaration identifies the source of metadata, it must precede any database access. However, it is the **ready** command or equivalent action that actually opens the database for access.

The **database** declaration optionally supports the specification of a *runtime* database. However, the runtime database is more appropriately referenced in (and opened by) the **ready** command. For example, your program may access a number of databases that use common metadata, but contain different data. Applications of this type include CAD/CAM and test control systems, in which a boilerplate database supplies metadata (for example, relations for wing struts and other aircraft assemblies) while instances of that database contain actual data (individual databases for IL-62, IL-70, and IL-82 aircraft designs).

The use of the boilerplate database for metadata helps ensure that you are keeping track of the same data for all your aircraft designs.

If you find that the runtime database is always the same, and different from the compiletime database, you can add the **runtime** clause to the **database** declaration. If you choose only one **compiletime** identifier, **gpre** uses that identifier for both compilation and runtime, unless you provide a runtime file in the **ready** command.

Syntax

```
database database-handle = [declaration-scope]
[compiletime] [filename] database-filespec
[runtime {[filename] database-filespec |
host-variable}]
declaration-scope::={static | extern}
```

Options

database-handle
Declares a name you can use when you have to reference multiple databases in a program.

Note

Many of the examples in this manual use the database handle DB, which is a reserved word in VAX COBOL. You cannot use reserved words as database handles.

compiletime
Specifies the database that **gpre** uses to look up field references when it preprocesses a file.

runtime
Specifies the database that the program uses at runtime.

declaration-scope
Declares the scope of the handle specified by the *database-handle* clause. If you do not specify a declaration scope, the scope of the handle defaults to **global**.

static
Specifies the scope of the declaration as only the module containing the **database** declaration.

extern
Specifies the database handle corresponds to one in another module with a **global** scope.

If all database handles in a module have the same scope, the handle for the default transaction also uses that scope. Otherwise, the handle for the default transaction has a **global** scope.

filename
An optional word.

database-filespec
Specifies the database from which the preprocessor reads the metadata. The *database-filespec* can be:

- A filename enclosed in single (') or double (") quotation marks, depending on your host language conventions.
- A logical name that resolves to a quoted file specification.

The file specification can contain the full pathname, including the name of the node on which the database is stored. If you are in a directory other than the one that contains the database file, the file specification *must* include the pathname. If the database is on another node, the *filespec* must include the node

name and pathname. You can define links or logical names for the database file.

File specifications for remote databases have the following form:

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

Be sure that what follows the node name and punctuation is a valid file specification on the target system. Use brackets, slashes, and spaces as appropriate.

host-variable
Specifies a host language variable to accept the location of a database at runtime.

Examples

The following C program includes two **database** declarations:

```
/* program mapper */
database atlas = compiletime filename 'atlas.gdb';
database gazetteer = compiletime filename
'atlas.gdb';

main()
{
ready atlas;
ready gazetteer;
for s in atlas.states sorted by s.state
   printf ("%s\n", s.state);
   for c in gazetteer.cities with c.state = s.state
      printf ("%s\t%s\t%s\n",
         c.city, c.latitude, c.longitude);
   end_for;
```

```
end_for;
}
```

The following program is a Pascal version of the program above:

```
program mapper (input, output);

database atlas = compiletime filename 'atlas.gdb';
database gazetteer = compiletime filename
'/usr/gds/examples/atlas.gdb';

begin

ready atlas;
ready gazetteer;
for s in atlas.states sorted by s.state
   begin
      writeln (s.state);
      for c in gazetteer.cities with c.state =
         s.state
         writeln (c.city, c.latitude,
c.longitude);
      end_for;
   end;
end_for;
finish atlas;
finish gazetteer;
end.
```

The following program uses a host variable to accept a user-supplied database name:

```
database db1 =compiletime filename "atlas.gdb"
runtime filename database_name;

#include <stdio.h>

⇓

main()
{
long_tourism_count;
char database_name[30]

printf ("Enter database name:\n");
```

```
scanf ("%s", database_name);
⇓
}
```

Troubleshooting See the Appendix for a discussion of errors and error handling.

See Also See the entry for **ready** in this chapter.

Display

Function

The **display** statement displays a form or a menu on the user's screen. In a form, it also controls the fields that are displayed, those that can be updated, the cursor position, and other characteristics of the form. In a menu, it controls the orientation of the display and how the menu appears in relation to other menus on the screen.

In a form, each display attribute can appear at most once per **display** statement.

A **display** statement must occur inside a **for_form—end_form** block or inside a **for_menu—end_menu** block.

Syntax

Form format:

```
display form-context-variable[display-
attribute...]

display-attribute::=
accepting field-list
cursor on field-name|
displaying field-list|
no_wait|
overriding field-list|
waking on field-list

field-list::= {*|field-commalist}::=
 {field-name|subform.subform-field-name}
```

Menu format:

```
display menu-context-variable
[horizontal|vertical]
[transparent|opaque]
```

Options

form-context-variable

Provides a name associated with this instance of the form in the **for_form** statement.

field-list

Specifies an asterisk (*) indicating that all fields are listed, a commalist of form field names without any qualifiers, or a field

in a subform. The subform variant allows you to both read and write a field from a subform, a capability not available in the **for_item** and **put_item** statements by themselves.

`accepting`
Specifies which fields can be updated.

`cursor`
Specifies the field on which the cursor is positioned when the form appears.

`displaying`
Lists the fields for which values established in the program should replace the fill characters established in the form definition. If you want to update the value between **display** statements, you must signal the change to the form manager by including the field in the **displaying** list of the second **display** statement.

`no_wait`
Updates the information on the screen, but does not pause for user input.

`overriding`
Lists the fields whose display attributes are controlled at runtime by the program.

`waking on`
Lists the fields that cause control to return to the program if the user changes their value. If you supply more than one field in the **waking on** list, you should test the special field TERMINATING_FIELD when control returns to your program to see which field caused the wake-up.

If the wakeup is on a repeating group item, you can reference other items from the repeating group.

`menu-context-variable`
A qualifier that references the context of the menu in the **for_menu** statement.

`horizontal | vertical`
Specifies the orientation of the menu on the screen. The default is vertical.

transparent | opaque
Transparent specifies that the menu displays on the screen without obscuring what is already there. Opaque specifies that the menu displays on the screen and covers what is already there. The default is opaque.

Examples

The following code fragment displays records from the STATES relation through a form:

```
for form x in states
   for s in states sorted by s.statehood
      strcpy (x.state_name, s.state_name);
      x.statehood = s.statehood;
      x.area = s.area;
      strcpy (x.state, s.state);
      strcpy (x.capital, s.capital);
      display x displaying statehood, area, state,
         state_name, capital;
      if (x.terminator == PYXIS_$KEY_PF1)
         break;
   end_for;
end_form;
```

The following code fragment creates a dynamic menu displaying the six New England states plus an "Exit" option:

```
   FOR_MENU M
   strcpy (M.TITLE_TEXT, "Choose a state");
   M.TITLE_LENGTH = strlen (M.TITLE_TEXT);
   for (i = 0; i < 7; i++)
      {
      PUT_ITEM E IN M
         strcpy (E.ENTREE_TEXT, state_list [i]);
         E.ENTREE_LENGTH = 2;
         E.ENTREE_VALUE = i;
      END_ITEM;
      }
   DISPLAY M;
   return M.ENTREE_VALUE;
END_MENU
```

Troubleshooting

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

See Also

See the entries in this chapter for:

- **case_menu**
- **case form**
- **for_menu**

For more information, see the chapter on using forms in GDML in the *Forms Guide*.

Erase

Function

The **erase** statement removes records from an open record stream.

You cannot erase records from views or joins. Rather, you must erase them through the source relations.

Syntax

```
erase context-variable [on-error]
   on-error::= on_error statement... end_error
```

Options

context-variable
Specifies the record stream from which to erase the record(s). You must declare the *context-variable* in a **for** or **start_stream** statement.

on_error
Specifies the action to be performed if an error occurs during the erase operation.

statement
A host language or GDML statement. If you include more than one *statement*, you must separate them using the host language convention.

end_error
Terminates the **on_error** clause.

Example

The following statements prompt for a field value and then delete records with that value:

```
based_on states.state statecode;
   ⇓
printf ("State to depopulate: ");
gets (statecode);
for c in cities with c.state = statecode
   erase c;
end_for;
```

Troubleshooting

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

See Also

See the entries in this chapter for:

- **on_error**
- **for**
- **start_stream**

Event_Init

Function

Th **event_init** statement is part of the synchronous event wait mechanism. This statement registers interest in one or more events to the event manager. Each event of interest is entered into the event table with an initial event count of zero. Once the event manager knows a process has an interest in an event, it keeps track of any occurrences of that event.

A process indicates it is ready to receive notification by executing the **event_wait** statement. The event manager sends notice the event occurred to processes that registered an interest and are ready to receive notification.

During precompiling, when **gpre** finds an **event_init** statement, it establishes the vector **gds_$events** and sets each element to one of the argument strings from the **event_init** statement. When your program expects to handle only one event, it can refer to the event either by the name given in the trigger definition following the **post** verb or by the first element in the vector. When your program expects to handle more than one event, your program uses the array to determine which event occurred.

Syntax

```
event_init event-handle [database-handle]
event-string commalist [on-error]

event-string ::= {quoted-string|
context-variable.database-field|
host-language-variable

on-error ::= on_error statement...end_error
```

Options

event_handle
: Specifies the event you want to wait for. The event-handle lets you use the variable in different parts of the routine to refer to the event.

database_handle
: Specifies which database to use when waiting for an event. This ensures clarity if multiple databases are used in the same module.

If no database handle is specified, this value defaults to the database named in the last **database** declaration.

event_string
The name of an event of interest.

quoted-string
A text literal enclosed in double quotes.

context-variable
A qualifier that references the context of an RSE in the code. Can only be used within a **for** loop, or whenever else context-variables are available.

database-field
A valid database field name. Can only be used within a **for** loop, or whenever else context-variables are available.

host-language-variable
A host-language variable of the character array type.

on_error
Specifies the action to be performed if an error occurs during the event_init operation.

statement
A host language or GDML statement. If you include more than one *statement*, you must separate them using the host language convention.

end_error
Terminates the **on_error** clause.

Example

The following program depends on a trigger to post an event naming a stock that changed by more than fifteen cents. This program registers its interest in such an event for stocks named in the **event_init** statement, then waits for such an event. When one of the events of interest occur, the event manager notifies the process. It " wakes up" and the program prints a message.

```
DATABASE DB = 'stocks.gdb';
#define number_of_stocks 5
char *event_names [] =
   {"APOLLO", "DEC", "HP", "IBM", "SUN"};
main()
{
int i;
```

```
double old_prices[number_of_stocks];

READY DB;

EVENT_INIT PRICE_CHANGE ("APOLLO", "DEC", "HP",
"IBM", "SUN");

START_TRANSACTION;

for (i=0;i<number_of_stocks;i++)
   {
   FOR S IN STOCKS WITH S.COMPANY = event_names[i]
      old_prices[i]=S.PRICE;
   END_FOR;
   }
COMMIT;
while (1)
   {
   EVENT_WAIT PRICE_CHANGE;

   START_TRANSACTION;

   for(i=0;i<number_of_stocks;i++)
      {
      if (gds_$events[i])
         {
         FOR S IN STOCKS WITH
          S.COMPANY = event_names[i]
            printf("STOCK: %s OLD: %f NEW: %f\n",
            S.COMPANY,old_prices[i],S.PRICE);
            old_prices[i]=S.PRICE;
         END_FOR;
         }
      }
   COMMIT;
   }
}
```

Troubleshooting See the Appendix for a discussion of error handling and a list of errors.

See Also

See the entries in this chapter for:

- **event_wait**
- **on_error**
- **database**

Event_Wait

Function

The **event_wait** statement indicates that your program is ready to receive notification of the occurrence of an event. This statement causes the process running your application to "sleep" until the event of interest occurs.

Your program must have already registered interest in an event with the **event_init** statement.

Syntax

```
event_wait event-handle [on-error]

on-error::= on_error statement...end_error
```

Options

event_handle
Specifies the event you want to wait for. The event-handle lets you use the variable in different parts of the routine to refer to the event.

on_error
Specifies the action to be performed if an error occurs during the event_wait operation.

statement
A host language or GDML statement. If you include more than one *statement*, you must separate them using the host language convention.

end_error
Terminates the **on_error** clause.

Example

The following program depends on a trigger to post an event naming a stock that changed by more than fifteen cents. This program registers its interest in such an event for stocks named in the **event_init** statement, then waits for such an event. When one of the events of interest occur, the event manager notifies the process. It "wakes up" and the program prints a message.

```
DATABASE DB = 'stocks.gdb';
#define number_of_stocks 5
char *event_names [] =
   {"APOLLO", "DEC", "HP", "IBM", "SUN"};
main()
```

```
{
int i;
double old_prices[number_of_stocks];

READY DB;

EVENT_INIT PRICE_CHANGE ("APOLLO", "DEC", "HP",
"IBM", "SUN");

START_TRANSACTION;

for (i=0;i<number_of_stocks;i++)
   {
   FOR S IN STOCKS WITH S.COMPANY = event_names[i]
      old_prices[i]=S.PRICE;
   END_FOR;
   }
COMMIT;
while (1)
   {
   EVENT_WAIT PRICE_CHANGE;

   START_TRANSACTION;

   for(i=0;i<number_of_stocks;i++)
      {
      if (gds_$events[i])
         {
         FOR S IN STOCKS WITH
          S.COMPANY = event_names[i]
            printf("STOCK: %s OLD: %f NEW: %f\n",
            S.COMPANY,old_prices[i],S.PRICE);
            old_prices[i]=S.PRICE;
         END_FOR;
         }
      }
   COMMIT;
   }
}
```

Troubleshooting See the Appendix for a discussion of error handling and a listing of errors.

See Also

See the entries in this chapter for:

- **event_init**
- **on_error**

Fetch

Function

The **fetch** statement advances the record stream pointer to the next record in a record stream, thus selecting the current record of that stream for whatever retrieval or manipulation operation you choose.

The **fetch** statement:

- Can be used only in a record stream created by a **start_stream** statement.
- Must precede any other statement that affects the current record.

Syntax

```
fetch stream-name [at end statement... end_fetch]
[on-error]
on-error::= on_error statement... end_error
```

Options

stream-name
Specifies the stream from which to fetch records. You must open the stream with a **start_stream** statement.

statement
Specifies GDML or host language statements to be executed on each record in the stream.

on_error
Specifies the action to be performed if an error occurs during the fetch operation.

statement
A host language or GDML statement. If you include more than one *statement*, you must separate them using the host language convention.

end_error
Terminates the **on_error** clause.

at end
Specifies the action to be taken when the program reaches the end of the stream. If you include more than one *statement*, you must separate them using the host language convention.

Examples

The following program demonstrates the use of the **start_-stream** statement in a loop that may be terminated by user interaction:

```
/* program map */
#include <stdio.h>
#include <ctype.h>

database db = filename 'atlas.gdb';

char genug [3];
int end_of_stream;

main()
{
   start_stream geodata using c in cities
      sorted by c.latitude, c.longitude;

   end_of_stream = 0;

   fetch geodata
      at end end_of_stream = 1;
   end_fetch;

   while (!end_of_stream) {
      printf ("%s\t%s\t%s\t%s\t%s\n",
         c.latitude, c.longitude, c.altitude,
         c.city, c.state);
      printf ("Seen enough? (Y/N) ");
      gets (genug);
      if (genug[0] == 'Y' || genug[0] = 'y')
         end_of_stream = 1;

      fetch geodata
         at end {
            end_of_stream = 1;
            printf ("Sorry, there is no more.\n");
            }
      end_fetch;
      }
   end_stream geodata;
   commit;
   finish;
}
```

The following example is the Pascal version of the preceding C program:

```
program map (input_output);
database db = filename 'atlas.gdb';
varend_of_stream: boolean;
   genug : char;
begin
   start_stream geodata using c in cities
      sorted by c.latitude, c.longitude;
   end_of_stream := false;
   fetch geodata
      at end end_of_stream := true;
   end_fetch;
   while not end_of_stream do begin
      writeln (c.latitude, c.longitude,
c.altitude,
         c.city, c.state);
      write ('Seen enough? (Y/N) ');
      readln (genug);
      if genug = 'Y' then
         end_of_stream := true;
      fetch geodata
         at end begin
            end_of_stream := true;
            writeln ('Sorry, there is no more.');
         end;
      end_fetch;
   end;
   end_stream geodata;
   commit;
   finish;
end.
```

Troubleshooting

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

See Also

See the entries in this chapter for:

- **start_stream**
- **on_error**

Finish

Function

The **finish** command closes either the default database (that is, a database opened without a database handle) or a specific database identified by a database handle.

Syntax

```
finish [database-handle-commalist] [on-error]
on-error ::= on_error statement...end_error
```

Options

database-handle
Specifies which open database or databases you want to close. A **database** declaration declares this handle.

If you use the optional *database-handle* clause, the database handle must have been previously associated with a database in the **database** declaration. This clause lets you close specific databases if you are using multiple databases in your program.

If you do *not* specify a database handle and do not use the -m option with **gpre**, the **finish** command commits the default transaction. If you want to close a specific database, you must first commit or roll back the transaction.

Non-default transactions that have not been committed are effectively rolled back by a **finish** command.

on_error
Specifies the action to be performed if an error occurs during the finish operation.

statement
A host language or GDML statement. If you include more than one *statement*, you must separate them using the host language convention.

end_error
Terminate the **on_error** clause.

Examples

The following statement closes any open databases:

```
finish;
```

The following statements close the databases identified by the handle:

```
finish atlas;
finish mapper;
```

Troubleshooting

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

See Also

See the entries in this chapter for:

- **on_error**
- **database**
- **commit**
- **rollback**

For

Function

The **for** statement executes a statement or group of statements once for each record in a stream formed by a record selection expression.

You can nest **for** loops to display a hierarchy of records or to join relations across databases.

Syntax

```
for [request-option] rse
statement...
end_for [on_error]

on_error ::= on_error statement...end_error
```

Options

request-option
Specifies a transaction handle and/or request handle that determine the transaction and/or request in which the **for** loop executes.

rse
Specifies the record selection criteria used to create the record stream.

The scope of a context variable declared in the RSE is the statement in which it was declared. Therefore, you can re-use a context variable from a **for** statement when you end the for loop with **end_for** and begin a new record stream with a **for** or **start_stream** statement.

You cannot reference more than one database in a record selection expression. Therefore, use nested **for** loops to join relations across databases.

statement
Specifies GDML or host language statements to be executed within the **for** loop. The statements you include in a **for** loop are subject to the following rules:

- You can nest **for** statements within other **for** statements.

- If you include more than one statement, you must separate them using the host language convention.
- If you use other GDML statements in the for loop, those statements can use the context variables declared in the **for** statement or in an outer statement, as well as contexts declared in the current **for** statement.

`on_error`
Specifies the action to be performed if an error occurs during the statement's operation.

`statement`
A host language or GDML statement. If you include more than one *statement* in an **on_error** clause, you must separate them using the host language convention.

`end_error`
Terminates the **on_error** clause.

Examples

The following statements retrieve records through a **for** loop:

```
for c in cities with population gt 1000000
   printf ("%s\t%s\t%d\n", c.city, c.state,
   c.population);
end_for;
```

The following statements join two relations using a **for** loop:

```
for c in cities cross s in states with c.state =
s.state
   printf ("%s\t%s\t%d\n", c.city, s.state_name,
   c.population);
end_for;
```

The following statements use an outer **for** loop to create a record stream from which a **store** statement takes some values, host variables supply some values, and unreferenced fields are set to missing:

```
for oldcity in cities with oldcity.city = hostvar1
   store newcity in cities using
      strcpy(newcity.city, hostvar2);
      strcpy(newcity.state, oldcity.state);
      newcity.population =
         oldcity.population * hostvar3;
      newcity.altitude = oldcity.altitude;
```

```
   end_store;
end_for;
```

The following example lists employees by department:

```
/* program print_depts */

#include <stdio.h>

database db = filename 'emp.gdb';

main()
{
   for d in departments sorted by d.department
      printf ("%s manager %d\n", d.department,
      d.manager);
      for e in employees with
      e.department = d.department sorted by e.badge
         printf ("\t%s\t%s\n", e.last_name,
            e.first_name);
      end_for;
   end_for;
}
```

The next example demonstrates the way to join relations across databases. It uses two copies of the sample atlas databases, one of which is in your current directory and the other in the examples directory provided with InterBase. The statements display values from the STATES relation in one copy of the atlas database, and values stored in another database from CITIES in those states. The join term is the STATE field in both relations:

```
/* program mapper */

#include <stdio.h>
#include <ctype.h>

database atlas = compiletime filename 'atlas.gdb';
database gazetteer = compiletime filename
'atlas.gdb';

main()
{
ready atlas;
ready gazetteer;
```

```
for s in atlas.states sorted by s.state
   printf ("%s\n", s.state);
   for c in gazetteer.cities with c.state = s.state
      printf ("%s\t%s\t%s\n",
         c.city, c.latitude, c.longitude);
   end_for;
end_for;
finish atlas;
finish gazetteer;
}
```

The following program hires everybody's offspring and assigns them new badge numbers.

Each request (that is, each **for** and **store**) must use the same request options, even though they are nested. The **modify** statement is not a separate request and does not require a transaction handle.

The outer **for** statement is in the default transaction, so that it will not read the newly stored records and start prompting for employee grandchildren:

```
/* program nested_for */
#include <stdio.h>

database db = filename 'emp.gdb';

int update_tr;
char check [3];
int fnl, lnl;

main()
{
ready;
start_transaction update_tr consistency read_write
reserving
   badge_num, employees for protected write;

start_transaction;

for e in employees
   printf ("Should we hire %s %s's kid? ",
         e.first_name, e.last_name);
   gets (check);
```

```
      if ( (check[0] == 'y') || (check[0] == 'Y') ) {
         for (transaction_handle update_tr) b in
            badge_num
            store (transaction_handle update_tr)
               n_e in employees using
               printf ("What's the kid's first
               name? ");
               gets (n_e.first_name);
               strcpy(n_e.last_name, e.last_name);
               printf ("What's the kid's date of
               birth? ");
               gets (n_e.birth_date.char[20]);
               n_e.badge = b.badge + 1;
               strcpy(n_e.department,"NEP");
               n_e.supervisor = 13;
            end_store;
            modify b using
               b.badge = b.badge + 1;
            end_modify;
         end_for;
         }
end_for;
commit update_tr;
commit;
finish;
}
```

Troubleshooting

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

See Also

See the entries in this chapter for:

- *request-option-clause*
- **on_error**
- **for blob**

See the entry in Chapter 3 for record selection expression.

For Blob

Function

The **for blob** statement retrieves data from a field that contains blob data.

The **for blob** statement is the easiest way to access blobs. You should use it when you process whole segments of a blob field or the entire contents of the blob buffer, without calling special formatting routines.

To read or write a blob field with the **for blob** statement:

- Construct a loop with the "other" **for** statement. This *outer for loop* creates a record stream.
- Construct a loop with the **for blob** statement. This *inner loop* swings through the blob, returning a segment at a time.
- Perform whatever action(s) you want to the blob under the control of the inner loop.
- Return control to the outer loop when you are finished with the blob field.

Syntax

```
for blob-variable in dbfield-expression
[from subytpe to subtype] [on-error]
     statement
end_for
on-error::= on_error statement end_error
```

Options

blob-variable
A temporary name used for name recognition. It is associated with individual segments in the field and is used very much like a context variable.

dbfield-expression
A value expression that identifies a field containing blob data.

from *subytpe* **to** *subtype*
Specifies the pre-defined subtype a blob filter converts from and the pre-defined subtype it converts to.

statement
Any valid host language or GDML statement. Use host language punctuation to terminate each statement.

on_error
Specifies the action to be performed if an error occurs during the specified operation.

statement
A host language or GDML statement. If you include more than one *statement* in an **on_error** clause, you must separate them using the host language convention.

end_error
Terminates the **on_error** clause.

Examples

The following statements create a record stream, display several structured fields from those records, and display a blob from each of those records:

```
for tour in tourism sorted by tour.state
   printf ("%s\t%s\t%s\n",
      tour.city, tour.state, tour.zip);
   printf("\n");
   for blob in tour.guidebook
      blob.segment[blob.length] = 0;
      printf ("%s", blob.segment);
   end_for;
   printf ("\n");
end_for; {for loop}
```

The following program copies a blob to another database by retrieving it in a **for blob** statement:

```
/* program update_guide */
#include <stdio.h>

database atlas = filename 'atlas.gdb';
database guide = filename 'coastal_guide.gdb';

main()
{
start_transaction;
for t in atlas.tourism
   store new in guide.tourism using
      strcpy(new.state, t.state);
```

```
      strcpy(new.city, t.city);
      strcpy(new.zip, t.zip);
      create_blob n_guide in new.guidebook;
      for o_guide in t.guidebook
         strcpy(n_guide.segment,
o_guide.segment);
         n_guide.length = o_guide.length;
         put_segment n_guide;
      end_for;
      close_blob n_guide;
   end_store;
end_for;
commit;
finish;
}
```

Troubleshooting See the Appendix for a discussion of errors and error handling.

See Also See the chapter on using blobs in the *Programmer's Guide*.

See also the entry in this chapter for **on_error** and the entry in Chapter 3 for *value-expression*.

For_Form

Function

The **for_form** statement binds a form definition to a window and creates a context in which form fields can be referenced. This statement does not cause a form to appear on the screen. Use the **display** substatement to display the form.

For form statements can be nested. As the forms are displayed, they will overlay each other. Unless a form is specified as **tag** or **transparent**, it completely covers the previously displayed form.

Syntax

```
for_form [options] form-context-variable in
[database-handle.]form-name
form-context-variable.field-name [.state]
     statement end_form
options ::= {transparent|tag|form_handle form-
handle-variable|transaction_handle transaction-
handle-variable}
```

Options

transparent
Pushes a transparent form over the current form, covering only those portions that are actually behind text on the top form.

tag
Displays a tag form horizontally in the bottom line of the form.

form_handle
Specifies a variable by which **gpre** can refer to the form in its calls to **pyxis**. If you do not specify a form-handle, **gpre** assigns it a unique name. If you do specify a form-handle, you can use the variable to invoke the form in different routines.

form-context-variable
The context variable qualifies references to the form fields to distinguish them from database fields or program variables.

form-name
Specifies the form to bind. The form name must be the name of a form already defined in a database. If you include a database handle, the form must be in that database. Otherwise, **gpre** searches databases referenced by the program, beginning with the most recently declared database.

statement
Any host language statement or a GDML **display**, **for_item**, or **put_item** statement. See the entries in this chapter for those statements. The **for_form** statement allows free reference to form fields inside the **for form** and **end_form** structure. If your program performs a statement, such as a return from a subprogram, that would cause it not to drop through to the **end_form** terminator, it should first execute a call to **pyxis_$pop_window**. The syntax for this call follows. **Gpre** automatically provides the context of **gds_window**.

C.

pyxis_$pop_window (&gds_$window)

All other languages:

pyxis_$pop_window (gds_$window)

Example

The following code fragment displays records from the STATES relation through a form:

```
for_form x in states
   for s in states sorted by s.statehood
      strcpy (x.state_name, s.state_name);
      x.statehood = s.statehood;
      x.area = s.area;
      strcpy (x.state, s.state);
      strcpy (x.capital, s.capital);
      display x displaying statehood, area, state,
         state_name, capital;
      if (x.terminator == PYXIS_$KEY_PF1)
         break;
   end_for;
end_form;
```

Troubleshooting

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

See Also

See the entries in this chapter for:

- **case_menu**
- **display**
- **for_item**
- **put_item**

For_Item

Function

The **for_item** statement is used inside a **for_form** statement to read items from a repeating group. The **for_item** statement allows only *read* access to the fields in its substatements. It is used in a **for_menu** statement to read the entrees in a menu.

Table 2-1.

Syntax

Form format:

```
for_item subform-context-variable in
form-context-variable.subform-name
  statement...
end_item
```

Menu format:

```
for_item entree-context-variable in menu-
context-variable
  entree-assignment-statements
end_item
```

Options

subform-context-variable
Specifies a context variable for the subform. This context variable must uniquely identify the subform in the form.

form-context-variable.subform-name

Specifies the subform name qualified with the context variable associated with the form in which the subform exists.

entree-context-variable
A qualifier that references the context of the entree in the **for_item** statement.

menu-context-variable
A qualifier that references the context of the menu in the **for_menu** statement.

entree-assignment-statements
Host language statements that read the values of:

- *entree-context-variable*.**entree_text**
- *entree-context-variable*.**entree_length**
- *entree-context-variable*.**entree_value**

Examples

The following code fragment modifies database records appearing in a subform:

```
FOR_ITEM FC IN F.CITY_POP_LINE
   if (FC.POPULATION.STATE ==
PYXIS_$OPT_USER_DATA)
      FOR C IN CITIES WITH C.CITY = FC.CITY
         AND C.STATE = F.STATE
         MODIFY C USING
            C.POPULATION = FC.POPULATION;
         END_MODIFY;
      END_FOR;
END_ITEM;
```

The following C code fragment prints the name and population of each city that is in menu M:

```
FOR_ITEM E IN M
   printf ("%s has a population of %d\n",
      E.ENTREE_TEXT, E.ENTREE_VALUE);
END_ITEM
```

Troubleshooting

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

See Also

See the entries in this chapter for:

- **case_menu**
- **display**

For_Menu

Function

The **for_menu** command lets you create a *dynamic menu*. A dynamic menu obtains the specifications for its title, entries and format while the application runs. These specifications most often come from dynamic user input or from database values. This differs from the **case_menu** command which requires that you specify all of these characteristics before the application is compiled.

Syntax

```
for_menu [(menu_handle menu-handle)] menu-
context-variable
  menu-title-assignment-statements
  entree-assignment-statements
  display statement
  menu-result-statements
end_menu
```

Options

menu_handle
Specifies a variable by which **gpre** can refer to the menu in its calls to Pyxis. If you do not specify a menu-handle, **gpre** assigns it a unique name. If you do specify a menu-handle, you can use the variable to invoke the menu in different routines.

menu-context-variable
A qualifier that references the context of the menu in the **for_menu** statement.

menu-title-assignment-statements
Host language statements in which you assign values to *menu-context-variable*.**title_text** and *menu-context-variable*.**title_length**. These statements must appear between the **for_menu** statement and the **end_menu** statement, and before the **display** statement.

entree-assignment-statements
Host language statements in which, within a **put_item** statement, you assign values to:

- *entree-context-variable*.**entree_text**
- *entree-context-variable*.**entree_length**
- *entree-context-variable*.**entree_value**

These statements must appear between the **for_menu** statement and the **end_menu** statement, and before the **display** statement. For more information, see the entry in this chapter for the **put_item** statement.

display statement
The display statement displays a menu on the user's screen. This statement must appear between the **for_menu** statement and the **end_menu** statement and after all title and entree assignment statements. For more information, see the entry in this chapter for the **display** statement.

menu-result-statements
Host language statements that use the values of:

- *menu-context-variable*.**entree_text**
- *menu-context-variable*.**entree_length**
- *menu-context-variable*.**entree_value**

for the entree selected from the menu.

The *menu-result-statement* also reads the *menu-context-variable*.**terminator** to determine what key was pressed to terminate the menu selection. This statement must appear between the **for_menu** statement and the **end_menu** statement, and after the **display** statement.

Example

The following C code fragment creates a menu consisting of the first ten cities in the CITIES relation. Once the user chooses a city, the program displays the selected city name and its population:

```
FOR_MENU M
   strcopy (M.TITLE_TEXT, "Choose a City");
   M.TITLE_LENGTH = strlen(M.TITLE_TEXT)
   FOR FIRST 10 C IN CITIES SORTED BY DESCENDING
   POPULATION
      PUT_ITEM E IN M
         strcpy (E.ENTREE_TEXT, C.CITY);
         E.ENTREE_LENGTH = strlen(E.ENTREE_TEXT);
         E.ENTREE_VALUE = C.POPULATION;
      END_ITEM
   END_FOR
   for (;;)
      {
```

```
        DISPLAY M VERTICAL
        if (M.TERMINATOR == PYXIS_$KEY_PF1)
           break;
        printf ("You chose %s, population %d\n",
           M.ENTREE_TEXT, M.ENTREE_VALUE);
        }
END_MENU;
```

Troubleshooting

See the Appendix of this manual for a discussion of errors and error handling.

See Also

See the entries in this chapter for:

- **case_menu**
- **display**
- **for_item**
- **put_item**

Get_Segment

Function

The **get_segment** statement reads a portion of a blob field. Before you can read a blob, you must open it with an **open_blob** statement.

Syntax

```
get_segment blob-variable [on-error]
on-error::= on_error statement... end_error
```

Options

blob-variable
A temporary name used for name recognition. It is associated with individual segments in the field and is used like a context variable. You must have assigned the blob variable in an earlier **open_blob** statement.

on_error
Specifies the action to be performed if an error occurs during the close operation.

statement
A host language or GDML statement. If you include more than one *statement* in an **on_error** clause, you must separate them using the host language convention.

end_error
Terminates the **on_error** clause.

Example

The following example creates a record stream, opens a blob field, and reads segments from the blob field:

```
for tour in tourism cross s in states over state
      sorted by s.state
   printf ("%s\t%s\t%d\n",
      tour.zip, s.state_name, s.area);
   open_blob b in tour.guidebook;
   get_segment b;
   while ( (gds_$status [1] == 0) ||
      (gds_$status [1] == gds_$segment) ) {
      b.segment[b.length] = 0;
      printf ("%s", b.segment);
      get_segment b;
      }
```

```
    close_blob b;
    printf ("\n");
end_for;
```

Troubleshooting

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

See Also

See also the entries in this chapter for:

- **open_blob**
- **close_blob**
- **on_error**

See the chapter on using blobs in the *Programmer's Guide*.

Modify

Function

The **modify** statement updates a field or fields in a record from a record stream.

You cannot modify records through views. Rather, you must modify them through the source relations. Finally, do not update records whose record selection expression includes a **reduced to** clause.

Syntax

```
modify context-variable using statement
end_modify [on-error]

on-error ::= on_error statement...end_error
```

Options

context-variable
Specifies the record stream from which the record is to be modified. You must declare *context-variable* in a **for** or **start_stream** statement.

statement
Specifies the action to be taken in modifying the record(s). The statements are typically assignments. If you include more than one statement, you must separate them using the host language convention.

If the field you want to modify contains blob data, use the **put_segment** statement to modify it.

on_error
Specifies the action to be performed if an error occurs during the close operation.

statement
A host language or GDML statement. If you include more than one *statement* in an **on_error** clause, you must separate them using the host language convention.

end_error
Terminates the **on_error** clause.

Example

The following statements increase the value of the POPULATION field in all cities in a given state:

```
based_on states.state statecode;

printf ("State code [2 characters, uppercase]: ");
gets (statecode);
for c in cities with c.state = statecode
   modify c using
      c.population = c.population * 1.2;
   end_modify;
end_for;
```

Troubleshooting

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

See Also

See the entries in this chapter for:

- **on_error**
- **for**
- **start_stream**
- **put_segment**

On_Error

Function

The **on_error** clause specifies the action the program takes if an error occurs during the execution of the associated GDML operation.

All GDML statements can include an **on_error** clause. The **database** declaration cannot.

Syntax

```
on_error
   statement
end-error
```

Options

statement
A host language or GDML statement. If you include more than one *statement*, you must separate them using the host language convention.

Example

The following program changes the type of ski areas, using reasonable error handling. It prompts for the name of a database and reprompts if there is an error during the **ready** command. The modification takes place in a subroutine that returns the status of the change. Validation errors are handled in the routine, thus avoiding restarting either the transaction or the **for** loop. Deadlocks are handled by the main routine, which rolls back and retries. Other errors print the status, rollback and exit:

```
/* program ski_areas */

#include <stdio.h>

database db = filename 'atlas.gdb';

based on ski_areas.name area_name;
based on ski_areas.type area_type;

char more [] = "y ";
int stat = 1;

main()
{
   while (stat) {
```

```
        stat = open_database();
        if (stat == -1) {
           printf ("Toodles, kid!\n");
           return;
           }
        }
     while (more[0] == 'y') {
        printf ("Enter ski_area name: ");
        gets (area_name);
        printf ("Enter new area type: ");
        gets (area_type);
        stat = modify_type ();
        while (!stat) {
           if (gds_$status [1] == gds_$deadlock)
              stat = modify_type ();
           else {
              printf ("Farewell, cruel world...\n");
              finish;
              return;
              }
           }
        printf ("Enter 'y' to change another record:
");
        gets (more);
        }
     finish;
}
int modify_type ()
{
     start_transaction;
     for ski in ski_areas with ski.name = area_name
re_mod:
        modify ski using
           strcpy(ski.type, area_type);
        end_modify
        on_error
           if (gds_$status [1] == gds_$not_valid) {
              printf ("Type must be N, A, or B\n");
              printf ("Enter new area type: ");
              gets (area_type);
              goto re_mod;
              }
           else if (gds_$status [1] != gds_$deadlock
```

```
)
            gds_$print_status (gds_$status);
         rollback;
         return 0;
      end_error;
   end_for
   on_error
      if (gds_$status [1] != gds_$deadlock)
         gds_$print_status (gds_$status);
      rollback;
      return 0;
   end_error;
   commit;
   return 1;
}

int open_database ()
{
   char filename [40];

   printf ("Please enter pathname of database
('quit' to exit): ");
   gets (filename);
   if (!strcmp(filename, "quit"))
      return -1;
   else {
      ready filename as db
      on_error
         printf ("Error during database open.
         Status follows.");
         gds_$print_status (gds_$status);
         printf("\n");
         return 1;
      end_error;
      }
   return 0;
}
```

The following example is a Pascal version of the preceding program:

```
program ski_areas (input_output);
database db = filename 'atlas.gdb';
type
```

```
   name  = based on ski_areas.name;
   a_type= based on ski_areas.type;
var
        more: char := 'y';
   area_name : name;
   area_type: a_type;
   stat  : integer;
function modify_type (area_name : name; area_type
            : a_type) : integer;
label
   re_mod;
begin
   modify_type := gds_$true;
   start_transaction;
   for ski in ski_areas with ski.name = area_name
re_mod:
      modify ski using
         ski.type := area_type;
      end_modify
      on_error
      begin
         if gds_$status [2] = gds_$not_valid then
         begin
            writeln ('Type must be N, A, or B');
            write ('Enter new area type: ');
            readln (area_type);
            goto re_mod;
         end
         else if gds_$status [2] <> gds_$deadlock
then
            gds_$print_status (gds_$status);
         modify_type := gds_$false;
         rollback;
         return;
      end;
      end_error;
   end_for
   on_error
      if gds_$status [2] <> gds_$deadlock then
         gds_$print_status (gds_$status);
      modify_type := gds_$false;
   end_error;
   commit;
```

```
end;
function open_database : integer;
var
   filename: array [1..40] of char;
begin
   open_database := 0;
   write ('Please enter pathname of database
(''quit'' to exit): ');
   readln (filename);
   if filename = 'quit' then
      open_database := -1
   else begin
      ready filename as db
      on_error
      begin
         writeln ('Error during database open.
            Status follows.');
         gds_$print_status (gds_$status);
         writeln;
         open_database := 1;
      end;
      end_error;
   end;
end;
begin
   repeat
   begin
      stat := open_database;
      if stat = -1 then
      begin
         writeln ('Toodles, kid!');
         return;
      end;
   end;
   until (stat = 0);
   while more = 'y' do
   begin
      write ('Enter ski_area name: ');
      readln (area_name);
      write ('Enter new area type: ');
      readln (area_type);
      stat := modify_type (area_name, area_type);
      while stat = gds_$false do
```

```
            begin
               if gds_$status [2] = gds_$deadlock then
                  stat := modify_type (area_name,
                  area_type)
               else
               begin
                  writeln ('Farewell, cruel world...');
                  finish;
                  return;
               end;
            end;
            write ('Enter "y" to change another
               record:');
            readln (more);
         end;
         finish;
      end.
```

Troubleshooting See the Appendix for a discussion of errors and error handling.

See Also See the discussion of the status vector (**gds$_status**) in the chapter on getting started with GDML in the *Programmer's Guide*.

Open_Blob

Function

The **open_blob** statement opens a blob so that its data may be retrieved.

You can process blobs by using the **get_segment** and **put_segment** statements:

Syntax

```
open_blob blob-variable in dbfield-expression
[from subtype to subtype] [on-error]
on-error ::= on_error statement... end_error
```

Options

blob-variable
Declares a temporary name to be used for name recognition. It is associated with individual segments in the field and is used like a context variable.

dbfield-expression
A value expression that identifies a field containing blob data. The context variable must be assigned in an outer **for** loop or **start_stream** statement.

from *subytpe* **to** *subtype*
Specifies the pre-defined subtype a blob filter converts from and the pre-defined subtype it converts to.

on_error
Specifies the action to be performed if an error occurs during the close operation.

statement
A host language or GDML statement. If you include more than one *statement* in an **on_error** clause, you must separate them using the host language convention.

end_error
Terminates the **on_error** clause.

Example

The following example creates a record stream from two relations, opens a blob field, and reads segments from the blob field:

```
for tour in tourism cross s in states over state
      sorted by s.state
```

```
    printf ("%s\t%s\t%d\n", tour.zip, s.state_name,
             s.area);
    open_blob b in tour.guidebook;
    get_segment b;
    while (gds_$status [1] == 0) {
       b.segment[b.length] = 0;
       printf ("%s", b.segment);
       get_segment b;
       }
    close_blob b;
    printf("\n");
end_for;
```

Troubleshooting See the Appendix for a discussion of errors and error handling.

See Also See also the entries in this chapter for:

- **on_error**
- **for blob**
- **close_blob**

See the chapter on using blobs in the *Programmer's Guide*.

Prepare

Function

The **prepare** command signals your intention to commit either the default transaction (that is, a transaction you start without declaring a handle) or the transaction specified by the optional transaction handle.

The **prepare** command executes the first phase of a two-phase commit. The InterBase access method polls all participants and waits for replies from each. It checks to see that no other database activity can affect the transaction. The **prepare** command is particularly useful for transactions that access multiple databases or for transactions that involve both database and non-database activity.

If the statement completes successfully, InterBase guarantees that a **commit** command executes successfully if the disk is still intact.

Syntax

```
prepare [transaction-handle] [on-error]
  on-error: := on_error statement... end-error
```

Options

transaction-handle

Specifies which transaction to prepare to commit. If the transaction you want to commit has a transaction handle associated with it, you must use that handle on the **prepare** and subsequent **commit** commands.

If you do not specify a handle on the **prepare** command, InterBase prepares the "default" transaction. The default transaction is what gets started when you use a **start_transaction** command without a handle.

on_error

Specifies the action to be performed if an error occurs during the close operation.

statement

A host language or GDML statement. If you include more than one *statement* in an **on_error** clause, you must separate them using the host language convention.

end_error
Terminates the **on_error** clause.

Example

The following extract includes a **prepare** command with an **on_error** clause:

```
prepare zip_code_update
   on_error
      printf ("Something failed during
prepare\n");
      gds_$print_status (gds_$status);
      printf ("Starting rollback...\n");
      rollback zip_code_update;
      goto failure;
   end_error;
commit zip_code_update;
```

Troubleshooting

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

See Also

See the entries in this chapter for:

- **commit**
- **on_error**

Put_Item

Function

The **put_item** statement is used inside a **for_form** statement to write items to a repeating group. Each **put_item** statement adds one row (that is, one group) to a subform. It is used inside a **for_menu** statement to add instances of entrees to a menu. You terminate a **put_item** statement with an **end_item**.

Syntax

Form format:

```
put_item subform-context-variable in
form-context-variable.subform-name statement
end_item
```

Menu format:

```
put_item entree-context-variable in
menu-context-variable entree-assignment-statements
```

Options

subform-context-variable
Specifies a context variable for the subform. This context variable must uniquely identify the subform in the form.

`form-context-variable.subform-name`
Specifies the subform name qualified with the context variable associated with the form in which the subform exists.

entree-context-variable
A qualifier that references the context of the entree in the **for_item** statement.

`entree-assignment-statement`
Host language statements in which you assign values to:

- *entree-context-variable.***entree_text**
- *entree-context-variable.***entree_length**
- *entree-context-variable.* **entree_length**

Examples

The following program adds records to a subform's repeating groups:

```
#include <stdio.h>

database db = "atlas.gdb";
```

```
main()
{
ready;
start_transaction;
for s in states
   for form f in city_states
      strcpy (f.capital, s.capital);
      f.statehood = s.statehood;
      strcpy (f.state_name, s.state_name);
      f.area = s.area;
      for c in cities with c.state = s.state
         put_item cs in f.cities
            strcpy (cs.city, c.city);
            cs.altitude = c.altitude;
            cs.population = c.population;
         end_item;
      end_for;
      display f
         displaying *
   end_form;
end_for
}
```

The following code fragment creates a dynamic menu displaying the six New England states plus an "Exit" option:

```
FOR_MENU M
   strcpy (M.TITLE_TEXT, "Choose a state");
   M.TITLE_LENGTH = strlen (M.TITLE_TEXT);
   for (i = 0; i < 7; i++)
      {
      PUT_ITEM E IN M
         strcpy (E.ENTREE_TEXT, state_list [i]);
         E.ENTREE_LENGTH = 2;
         E.ENTREE_VALUE = i;
      END_ITEM;
      }
   DISPLAY M;
   return M.ENTREE_VALUE;
END_MENU
```

Troubleshooting See the Appendix for a discussion of errors and error handling.

See Also

See the entries in this chapter for:

- **case_menu**
- **display**
- **for_form**
- **for_item**

Put_Segment

Function

The **put_segment** statement writes a portion of a blob field.

Before you can write a blob field, you must create it with a **create_blob** statement.

Syntax

```
put_segment blob-variable [on-error]
  on-error::= on_error statement... end_error
```

Options

blob-variable
A temporary name used for name recognition. It is associated with individual segments in the field and is used much like a context variable. You must have assigned the blob variable in an earlier **open_blob** statement.

on_error
Specifies the action to be performed if an error occurs during the put_segment operation.

statement
A host language or GDML statement. If you include more than one *statement* in an **on_error** clause, you must separate them using the host language convention.

end_error
Terminates the **on_error** clause.

Examples

The following statements create a record stream, create a blob field, and write segments to the blob field:

```
store tour in tourism using
   printf ("Enter state code: ");
   gets (tour.state)
   printf ("Enter zip code: ");
   gets (tour.zip)
   printf ("Enter city: ");
   gets (tour.city)
   create_blob b in tour.guidebook;
   printf ("Enter new blurb one line at a time\n");
   printf ("A line containing '-30-' ends input");
   gets (b.segment);
   while (strcmp(b.segment, "-30-")) {
```

```
            for (i=strlen(b.segment);i>=0;i--){
               if (b.segment[i] != ' ') {
               b.segment[i+1]='\n';
               b.segment[i+2]=0;
               b.length = i+1;
               i=0;
               }
               }
            put_segment b;
            gets (b.segment);
         }
      close_blob b;
end_store;
```

The following program copies the contents of a blob field from one database to another:

```
/* program update_guide */

#include <stdio.h>

database atlas = filename 'atlas.gdb';
database guide = filename 'coastal_guide.gdb';

main()
{
start_transaction;
/* copy a blob to another database by retrieving it
in a blob for */
for t in atlas.tourism
   store new in guide.tourism using
      strcpy( new.state, t.state);
      strcpy( new.city, t.city);
      strcpy( new.zip, t.zip);
      create_blob n_guide in new.guidebook;
      for o_guide in t.guidebook
         strcpy( n_guide.segment,
o_guide.segment);
         n_guide.length = o_guide.length;
         put_segment n_guide;
      end_for;
      close_blob n_guide;
   end_store;
end_for;
```

```
commit;
finish;
}
```

Troubleshooting

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

See Also

See also the entries in this chapter for:

- **on_error**
- **open_blob**
- **close_blob**

See the chapter on using blobs in the *Programmer's Guide*.

Ready

Function

The **ready** command opens one or more databases for access. When it encounters a **ready** command, InterBase:

- Initializes itself internally. The initialization sets up data structures and allocates dynamic memory.
- Looks at the file name of the database and determines if the file is stored on the originating node (a *local database*) or on another node (a *remote database*). InterBase provides transparent access to remote databases.
- Opens the database file and looks at the header page. Assuming the header page identifies the file as containing a valid, unbroken database with the correct version of the on-disk structure, InterBase permits further access. Otherwise, it returns an error.

Depending on the options you set when preprocessing the program with **gpre**, you may not have to issue a **ready** command to access a database. By default, **gpre** generates a **ready** if one is needed so the database is automatically readied the first time your program refers to that database. However, if you specify the **manual** option when you preprocess the program, **gpre** does not generate a **ready** command (and **start_transaction** statement). The advantage to using the **manual** option is that preprocessed code is smaller and simpler.

You should close each database with a **finish** command when you are done with it. This practice saves system resources.

Syntax

```
ready (dbhandle-commalist | runtime-file)
   [on-error]
   dbhandle::= {database-handle | runtime-file
      as database-handle}
   runtime-file::= {database-filespec |
      host-variable}
   on-error ::= on_error statement... end_error
```

Options

dbhandle

References either a database assigned a handle in a **database** declaration or a database you specify with *database-filespec*

and to which you assign a database handle. The *database-filespec* must be a quoted file specification or a logical name that resolves to a quoted file specification.

In the case of a handle assigned in a previous **database** declaration, the database you ready for runtime access is the same as the one you declared for compiletime access.

database-filespec

Specifies the database from which the preprocessor reads the metadata. The *database-filespec* can be:

- A filename enclosed in single (') or double (") quotation marks, depending on your host language conventions.
- A logical name that resolves to a quoted file specification.

The file specification can contain the full pathname, including the name of the node on which the database is stored. If you are in a directory other than the one that contains the database file, the file specification *must* include the pathname. If the database is on another node, the *filespec* must include the node name and pathname. You can define a link or logical name for the database file.

File specifications for remote databases have the following form.

Table 4-2. 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

Be sure that what follows the node name and punctuation is a valid file specification on the target system; use brackets, slashes, and spaces as appropriate.

host-variable
Specifies a host language variable to accept the location of a database at runtime.

runtime-file
Readies the named database file. You can use this option if your program accesses only one database.

The file specification can contain the full pathname, including the name of the node on which the database is stored. See the earlier discussion of *database-filespec* for information about accessing remote databases.

on-error
Specifies the action to be performed if an error occurs during the ready operation.

Example

The following sequence declares a database and readies it:

```
/* program atlas */

database atlas = filename 'atlas.gdb';

main()
{
ready atlas;
start_transaction;
      ⇓
rollback;
finish atlas;
}
```

Another option is to assign the database handle in the **ready** command. For example, the following sequence declares a compiletime database and readies different databases for runtime access:

```
/* program ski_areas */

#include <stdio.h>

database atlas = filename 'atlas.gdb';
```

```
charfilename[40];
intopen_database = 0;

main()
{
   while (!open_database) {
      open_database = 1;
      printf ("Please enter db pathname ('quit' to
exit): ");
      gets (filename);
      if (!strcmp(filename,"quit")) {
         printf ("Toodles, kid!\n");
         return;
         }
      ready filename as atlas
      on_error
         printf ("Error during database open.
Status follows.\n");
         gds_$print_status (gds_$status);
         printf("\n");
         open_database = 0;
      end_error;
      }
/* do work */
finish;
}
```

Troubleshooting See the Appendix for a discussion of errors and error handling.

See Also See the entries in this chapter for:

- **on_error**
- **database**
- **finish**

Release_Requests

Function

The **release_requests** command frees the memory used by the execution tree of all compiled requests for a database and sets the request handles to null.

In most programs, the program logic involves loops that can re-use requests. Therefore, InterBase saves requests in their compiled and optimized form. However, if your program finishes and re-readies databases, requests must be re-compiled. The **finish** commit automatically marks requests from that module as obsolete and ensures they are re-compiled when, and if, they are re-used.

Large programs consisting of separately compiled modules sometimes have requests in modules that do not contain a **finish** command. In those cases, you can use the **release_requests** command to release resources and ensure re-compilation. You must include the **release_requests** command in one externally callable subroutine in each module that contains a database request. Before you execute a **finish** command, call each of the "release" subroutines to release resources allocated in its module.

Syntax

```
release_requests [[for]] database-handle]
   [[on-error]
on-error::= on_error statement... end_error
```

Options

database-handle
Specifies the database whose requests you want to release. If you do not specify a database handle, InterBase releases requests associated with all open databases.

on_error
Specifies the action to be performed if an error occurs during the release operation.

statement
A host language or GDML statement. If you include more than one *statement* in an **on_error** clause, you must separate them using the host language convention.

end_error
Terminates the **on_error** clause.

Example

The following program calls one external routine to perform an action and another to release resources associated with the request:

```
/* program driver */
/* file worker.e */
#include <stdio.h>

database atlas = filename "atlas.gdb";

worker()
{
   char quit [4];

   strcpy (quit, "no");
   while (strcmp (quit, "yes")) {
      ready atlas;
      worker();
      worker_release();
      finish;
      printf ("Done yet ('yes' to stop): ");
      gets (quit);
      }
}

/* The following module is called by the preceding
program: */

/* file worker_1.e */
database atlas = EXTERN filename 'atlas.gdb';
worker_release()
{
   release_requests;
}
worker()
{
   int i;

   i = 0;
   start_transaction;
   for s in states
```

```
        i++;
    end_for;
    commit;
    printf ("There are %d states.\n", i);
}
```

Troubleshooting See the Appendix for a discussion of errors and error handling.

See Also See the entry for **finish** in this chapter.

Request Options

Function

The *request-option* is an optional GDML clause that lets you specify the instantiation (recursion) level of a request, the transaction handle for a request, or the request handle for a request.

SQL does not support recursion. If you are using SQL statements in your program, do not involve any SQL operations in a recursive request.

Syntax

```
request-option ::= (option-commalist)
option ::= {level integer-expression |
       transaction_handle host-variable
       request_handle host-variable
```

Options

level *integer-expression*
Specifies the instantiation level of a request.

transaction_handle *host-variable*
Specifies a transaction handle for the transaction in which the statement executes.

request_handle *host-variable*
Specifies a request handle for the request in which the statement executes.

Examples

The following program produces a horizontal organization chart with the president at the top left and the rest of the company moving to the right:

```
/* program map */

database db = filename 'emp.gdb';

based on employees.badge badge_type;
int level;
char blanks[] = " ";

main()
{
   ready;
   start_transaction;
```

```
   printf ("    Employee Roster\n\n");
   for e in employees with e.supervisor missing
      printf ("%s %s\n", e.first_name,
         e.last_name);
      print_next (0, e.badge);
   end_for;
   commit;
   finish;
}
print_next (lev, super)
int lev;
based on employees.badge super;
{
   int offset;

   for (level lev) e in employees with
            e.supervisor = super
      offset = (lev) * 4;
      printf ("%*s%s %s\n", offset,
         blanks,
         e.first_name,
         e.last_name);
      print_next (lev+1, e.badge);
   end_for;
}
```

The following program starts a named **consistency** mode transaction to update the BADGE relation:

```
/* program get_badge */

database emp = filename 'emp.gdb';

int get_badge_tr;
long get_badge;

main()
{
   get_badge_tr = 0;
   start_transaction get_badge_tr
      consistency read_write reserving
      badge_num for protected write;
   for (transaction_handle get_badge_tr) b in
      badge_num
```

```
        get_badge = b.badge;
        modify b using
            b.badge++;
        end_modify;
    end_for;
    commit get_badge_tr;
}
```

The following program hires everybody's offspring and assigns them new badge numbers. Note that each request (that is, each **for** and **store**) must use the same request options, even though they are nested. The **modify** statement is not a separate request and does not require a transaction handle. The outer **for** statement is in the default transaction. It does not read the newly stored records and start prompting for employee grandchildren:

```
/* program nested_for */

database db = filename 'emp.gdb';

int gds_$handle = 0;
char check[] = "y ";
int update_tr = 0;

main()
{
ready;

start_transaction update_tr consistency read_write
    reserving badge_num, employees for
    protected write;

start_transaction;

for e in employees
    printf ("Should we hire %s %s's kid? ",
            e.first_name, e.last_name);
    gets (check);
    if ( (check[0] == 'y') || (check[0] == 'Y') ) {
        for (transaction_handle update_tr) b in
            badge_num
          store (transaction_handle update_tr) n_e
            in employees using
          printf ("What's the kid's first name?");
```

```
            gets (n_e.first_name);
            strcpy(n_e.last_name, e.last_name);
            printf ("What's the kid's birthdate?");
               gets (n_e.birth_date.char[20]);
               n_e.badge = b.badge + 1;
               strcpy(n_e.department, "NEP");
               n_e.supervisor = 13;
            end_store;
            modify b using
               b.badge++;
            end_modify;
         end_for;
         }
end_for;
commit update_tr;
commit;
finish;
}
```

Troubleshooting See the Appendix for a discussion of errors and error handling.

See Also See the entries in this chapter for:

- **for**
- **start_stream**
- **store**

Rollback

Function

The **rollback** command restores the database to its state prior to the current transaction. It affects all databases in the transaction, discarding all modified buffers and closing any open record streams.

The **rollback** command ends a transaction and undoes all changes made to the database since the most recent **start_transaction** command or since the start of the transaction specified by the transaction handle.

Syntax

```
rollback [transaction-handle] [on-error]
on-error::= on_error statement... end_error
```

Options

transaction-handle

Specifies the transaction you want to roll back. If the transaction you want to roll back has a transaction handle associated with it, you must use that handle when you roll back the transaction.

If you do not specify a transaction handle on a **rollback** command, InterBase rolls back the "default" transaction. The default transaction is what gets started when you use a **start_transaction** command without a handle.

on-error

Specifies the action to be performed if an error occurs during the rollback operation.

Example

The following statements modify the BADGE relation, but rollback the transaction if there is an error:

```
for (transaction_handle get_badge_tr) b in
badge_num
   get_badge = b.badge;
   modify b using
      b.badge++;
   end_modify
   on_error
      if (gds_$status [1] == gds_$deadlock)
         get_badge = 0;
```

```
      else
         get_badge = -1;
      rollback get_badge_tr;
      return;
   end_error;
end_for;
```

Troubleshooting

A **rollback** command cannot fail.

See Also

See the entries in this chapter for:

- **start_transaction**
- **on_error**

Save

Function

The **save** statement writes data to the database and makes the changes visible to other users while retaining the context of the current transaction. You can save changes without having to end a transaction.

The **save** statement flushes all modified buffers, but keeps open any record streams that are currently open.

Syntax

```
save [transaction-handle commalist] [on-error]
on-error::= on_error statement...end_error
```

transaction-handle
Specifies the transaction you want to save. If the transaction you want to save has a transaction handle associated with it, you must use that handle when you save the transaction. If you do not specify a transaction handle on a **save** statement, InterBase saves the default transaction, the transaction that InterBase starts when you use a **start_transaction** command without a handle.

on-error
Specifies the action to be performed if an error occurs during the save operation.

Examples

In the following C example, a record is saved as soon as it is modified:

```
database db = filename "bugs.gdb";
main ()
{
char     buffer [25]
ready;
start_transaction;
for b in bug_assignments
      printf ("%s\t%s\t%d\n",b.engineer,
      b.date_assigned.char[11], b.bug_no);
      printf ("enter a new name to reassign bug:");
      gets(buffer);
      if (*buffer)
         modify b using
```

```
            strcpy (b.engineer,buffer);
      end_modify
      save;
end_for;
commit;
finish;
```

Troubleshooting See the Appendix for the standard **gpre** error messages.

See Also See the entries in the this chapter for:

- **start_transaction**
- transaction-handle
- **on_error**

Start_Stream

Function

The **start_stream** statement declares and opens a record stream.

You can start a stream with the **for** statement or with the **start_stream** statement. The **for** statement is generally recommended. However, you may want to use a **start_stream** statement if you are processing:

- Several streams in parallel.
- A stream until some condition is met, and then exiting from the stream.

Syntax

```
start_stream [request-option] stream-name
   using rse [on-error]
   statement...
end_stream stream-name [on-error]

on-error::= on_error statement... end_error
```

Options

request-option
Specifies a transaction handle and/or request handle that determine the transaction and/or request in which the **start_stream** statement executes.

stream-name
Names the stream. The name can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_).

The context of the stream name is the whole module that contains the **start_stream** statement, so you cannot re-use a stream name in the same module.

rse
Specifies the record selection criteria used to create the record stream.

on-error
Specifies the action to be performed if an error occurs when you start the stream or when it terminates. Errors on the **end_stream** generally occur only in extreme cases, such as a network partition while the stream is still open.

`statement`
Specifies GDML or host language statements to be executed within the stream. The statements you include are subject to the following rules. If you:

- Include more than one *statement*, you must separate them using the host language convention.
- Use other GDML statements while the stream is open, those statements can use only the context variables declared in the GDML block, in outer blocks, or in inner blocks. You can re-use the context variables outside those blocks.

Example

The following program illustrates the use of the **start_stream** statement in a loop that may be terminated by user interaction:

```
/* program map */

database db = filename 'atlas.gdb';

int end_of_stream;
char genug[] = "y ";

main()
{
   ready;
   start_transaction;
   start_stream geodata using c in cities
      sorted by c.latitude, c.longitude;
   end_of_stream = 0;
   while (!end_of_stream) {
      fetch geodata
         at end end_of_stream = 1;
      end_fetch;
      if (!end_of_stream) {
         printf ("%s\t%s\t%s\t%s\t%s\n",
            c.latitude, c.longitude,
            c.altitude, c.city, c.state);
         printf ("Seen enough? (Y/N) ");
         gets (genug);
         if( (genug[0] == 'Y')||(genug[0] == 'y') )
            end_of_stream = 1;
         }
      }
```

```
    end_stream geodata;
    commit;
    finish;
}
```

Troubleshooting See the Appendix for a discussion of errors and error handling.

See Also See the entries in this chapter for:

- *request-option*
- **on_error**

See also the entry in Chapter 3 for record selection expression.

Start_Transaction

Function

The **start_transaction** statement begins a group of statements that are executed as one logical unit.

A process can start any number of independent transactions. This capability facilitates the development of server processes and allows system service routines to use databases without affecting user-level database activity.

Syntax

```
start_transaction [transaction-handle]
   [concurrency | consistency]
   [read_write | read_only ]
   [wait | nowait}
   [reserving-clause]
   [on-error]
reserving-clause::= reserving
   reserved-relation-commalist
reserved-relation::= [database-handle.]relation
         for [protected]
         {read | write}
   on-error::= on_error statement...end_error
```

Options

transaction-handle

Declares a name that you can use when you have to reference multiple transactions in a program.

If you start a transaction without specifying a transaction handle, **gpre** starts the "default transaction." There is one default transaction per process. When **gpre** encounters a subsequent statement without a transaction handle, it generates a test for the default handle. If there is no default transaction, **gpre** starts one. In any case, **gpre** applies statements without transaction handles to the default transaction.

concurrency (default)

The **concurrency** default provides high throughput and concurrency with generally satisfactory consistency. No transaction sees any data written by another active transaction.

consistency
The **consistency** option provides a high level of database consistency that guarantees that all transactions are serializable (that is, having the same effect on the database as if all transactions were run sequentially in some order) at the expense of concurrency.

To ensure a deadlock-free transaction, use the **consistency** option and reserve the relations required by the transaction for **read** or **write** depending on the mode in which they will be used. However, this option does not allow concurrent write access to the reserved relations.

read_write (default)
read_only
The default intention of a transaction is that it will read and write data. You may choose to declare a transaction **read_only** to document its behavior or as a check on program logic.

wait (default)
nowait
The default action if your program encounters a locked object is to wait until the lock goes away. The **nowait** option produces a *lock_conflict* error whenever a program encounters a locked object. The **nowait** option is not recommended because it requires more error handling in a program and can lead to unnecessary rollbacks.

reserving
Lists the relations to be used in the transaction. InterBase locks those relations for your access if you choose **consistency** mode. You must list each relation that the transaction will "touch" (that is, if it is used at all, in any capacity). List relations individually. You can specify different relation locking criteria for each. However, if you choose **read_only** for the transaction (see above), you cannot reserve a relation for **write**.

read | write
If you have a **concurrency** mode transaction, you can optionally reserve a relation for **protected write**. This mode allows other users to read the relation, but prevents them from writing to it. By default, **concurrency** mode transactions are reserved for shared access, an access mode that all users write to the relation.

The **protected write** reserving option is the default for **consistency** mode transactions.

To ensure a deadlock-free transaction, use the **consistency** option and reserve the relations required by the transaction for **read** or **write** depending on the mode in which they will be used.

dbhandle
References the handle assigned a database in a **database** declaration

relation
Specifies the relation to be used in the reserving clause

on-error
Specifies the action to be performed if an error occurs when you start the transaction.

Examples

The following statement starts a transaction that will become the default transaction because there is no transaction handle:

```
start_transaction;
```

The following statement starts a transaction and assigns a transaction handle:

```
/* program zip_update */
database db = filename 'atlas.gdb';

int zip = 0;

main()
{
start_transaction zip;
      ⇓
commit zip;
finish;
}
```

The following statement starts a transaction with a reserving clause:

```
/* program zip_update */
database db = filename 'atlas.gdb';

int zip = 0;
```

```
main()
{
   start_transaction zip
      read_write consistency
      reserving catalog.catalog_items for write;
}
```

Troubleshooting See the Appendix for a discussion of errors and error handling.

See Also See the entries in this chapter for:

- **prepare**
- **commit**
- **rollback**
- **on_error**

Store

Function

The **store** statement inserts a new record into a relation.

You cannot store records into views formed from more than a single relation. Rather, you must store them into the source relations.

Syntax

```
store [request-option] relation-clause using
   statement
end_store [on-error]
   on-errror ::= on_error statement... end_error
```

Options

request-option

Specifies a transaction handle and/or request handle that determine the transaction and/or request in which the **store** statement executes.

If you nest a **store** statement inside a **for** loop and use an explicit transaction handle on the **for** statement, you must also use the transaction handle on the **store** statement. Otherwise, the **store** statement will be executed inside the default transaction.

relation-clause

Specifies the relation into which the new record is to be stored. See the entry for RSE in the previous chapter for more information about the *relation-clause*.

on-error

Specifies the action to be performed if an error occurs during the store operation.

statement

Specifies the action to be taken in storing the record(s). The *statement*s are typically assignments. If you include more than one *statement*, you must separate them using the host language convention.

Examples

The following statement stores a new record in SKI_AREAS:

```
store ski in ski_areas using
   printf ("Name: ");
   gets (ski.name);
   printf ("City: ");
   gets (ski.city);
   printf ("State: ");
   gets (ski.state);
   printf ("Type: ");
   gets (ski.type);
end_store;
```

The following statements use an outer **for** loop to create a record stream from which a **store** statement takes some values, host variables supply some values, and unreferenced fields are set to missing:

```
for oldcity in cities with oldcity.city_name =
hostvar1
   store newcity in cities using
      strcpy (newcity.city, hostvar2);
      strcpy (newcity.state, oldcity.state);
      newcity.population =
         oldcity.population * hostvar3;
      newcity.altitude = oldcity.altitude;
   end_store;
end_for;
```

The following program hires everybody's offspring and assigns them new badge numbers. Each request (that is, each **for** and **store**) must use the same request options, even though they are nested. The **modify** statement is not a separate request and does not require a transaction handle. The outer **for** statement is in the default transaction. It does not read the newly stored records and start prompting for employee grandchildren:

```
/* program nested_for */

database db = filename 'emp.gdb';

int update_tr = 0;
char check [] ="y ";

main()
```

```
{
ready;
start_transaction update_tr consistency read_write
   reserving badge_num, employees for protected
write;

start_transaction;

for e in employees
   printf ("Should we hire %s %s's child ? ",
            e.first_name, e.last_name);
   gets (check);
   if ( (check[0] == 'y') || (check[0] == 'Y') )
      {
      for (transaction_handle update_tr) b in
            badge_num
         store (transaction_handle update_tr) n_e
            in employees using
            printf ("What's the child's first
            name? ");
            gets (n_e.first_name);
            strcpy(n_e.last_name, e.last_name);
            printf("What's the child's date of
            birth? ");
            gets (n_e.birth_date.char[20]);
            n_e.badge = b.badge + 1;
            strcpy (n_e.department, "NEP");
            n_e.supervisor = 13;
         end_store;
         modify b using
            b.badge = b.badge + 1;
         end_modify;
      end_for;
      }
end_for;
commit update_tr;
commit;
finish;
}
```

Troubleshooting See the Appendix for a discussion of errors and error handling.

See Also

See the entries in this chapter for:

- **request option**
- **on_error**

See the entry in Chapter 3 for record selection expression.

Store Blob

Function

The **store blob** statement stores data into a blob field.

The storage of blob fields must occur in the larger context of whole record storage. To store a record that contains a blob field:

- Use the "other" **store** statement to store non-blob fields with host language assignments.
- When you get to the blob field, construct a loop to solicit and accept blob data or to retrieve data from another blob field or text file.
- Use the **store blob** statement to assign data to the blob field.

Note

When you store or modify a blob, you cannot directly assign data to that blob. You must use the following syntax:

<blob field name> = edit

Syntax

```
store blob-variable in dbfield-expression
[on-error]

on-error ::= on_error statement...end_error
```

For some guidance on the best approach to processing blobs, see the entries in this chapter for **for blob** and **open_blob**.

Options

blob-variable
A temporary name used for name recognition. It is associated with individual segments in the field and is used much like a context variable.

dbfield-expression
Identifies a field that contains blob data.

on-error
Specifies the action to be performed if an error occurs during the store operation.

statement
Any valid host language or GDML statement. Use host language punctuation to terminate each statement.

Example

The following statements store a record in CATALOG_ITEMS:

```
store cb in catalog_blurb using
   strcpy(cb.item_number, "PRO10");
   strcpy(cb.date_modified.char[11], "today");
   printf ("Enter new blurb one line at a time
      with ^Z to end input");
   store blob in cb.blurb
   while (gets (blob.segment) != 0) {
      blob.length = strlen(blob.segment);
      put_segment blob;
      }
   end_store;
end_store;
```

Troubleshooting

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

See Also

See the entries in this chapter for:

- **on_error**
- **store**
- **modify**
- **for**
- **for blob**

See also the entry in Chapter 3 for value expression.

Transaction Handle

Function

The *transaction-handle* clause specifies the name of a transaction in several GDML statements.

If you do not start a transaction with the **start_transaction** statement, choosing instead to let **gpre** start transactions as needed, you can still specify the transaction under which you want a statement to be executed by declaring a transaction handle in the *request-option* clause of the **for**, **store**, and **start_stream** statements. If that transaction does not exist, **gpre** starts it.

Syntax

```
host-variable
```

Options

host-variable
A host language program variable that serves as the transaction handle.

- For Ada, the transaction handle must be declared as INTERBASE.TRANSACTION_HANDLE and initialized to zero.
- For BASIC, the transaction handle must be declared as LONG and set to 0.
- For C programs, the transaction handle must be declared as a long integer initialized to **null** (0).
- For COBOL, the transaction handle must be declared as PIC S(9) COMP.
- For FORTRAN programs, the transaction handle must be declared as INTEGER*4 set to 0.
- For Pascal programs, the transaction handle must be explicitly declared in the program as a pointer to any type and initialized to **nil** before use. The variable **gds_$handle** is pre-declared as a type for Pascal.
- For PL/I, the transaction handle must be declared as a pointer and initialized to **NULL()**.

Example

The following Pascal example starts two named transactions, performs some unspecified data manipulation in each, then writes the changes for only the specified transaction to the database. Then it commits the other transaction:

```
start_transaction store_resort;
start_transaction drop_resort;
      ⇓
for (transaction_handle store_resort)
      ⇓
for (transaction_handle drop_resort)
      ⇓
commit store_resort;
      ⇓
commit drop_resort;
```

Troubleshooting

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

See Also

See the entries in this chapter for:

- **commit**
- **prepare**
- **rollback**
- **start_transaction**
- **for**
- **store**
- **start_stream**
- **request-option**

Chapter 5
SQL Expressions

This chapter contains entries for SQL expressions.

Overview

InterBase uses the following SQL expressions

- Predicate, which specifies a condition that is applied to a record or records in a table that evaluates to true, false or unknown
- Scalar expression, which is a symbol or string of symbols used in predicates to calculate a value
- Select expression, which specifies the search and delivery conditions for record retrieval

Predicate

Function

The **predicate** specifies a condition that is applied to a record or records in a table that evaluates to true, false or unknown. It follows the **where** clause in the **delete** and **update** statements and the *select expression*.

Syntax

```
predicate::= {condition | condition and predicate
|condition or predicate | not predicate}

condition::= {compare-condition|
between-condition|like-condition|in-condition|
exists-condition|(predicate)}
```

The following sections describe the five options of the **predicate**:

- Compare condition
- Between condition
- Like condition
- In condition
- Exists condition

Compare Condition

Function

The **compare** condition describes the characteristics of a single scalar expression (for example, a missing or null value) or the relationship between two scalar expressions (for example, *x* is greater than *y*).

Syntax

```
{scalar-expression comparison-operator
   scalar-expression|
scalar-expression comparison-operator
   (column-select-expression)
|scalar-expression is [not]null}

comparison-operator::= {=|^=|<|^<|<=|>|^>|>=}

column-select-expression ::= select [distinct]
```

Options

scalar-expression
A **scalar** expression is a field reference, an alphanumeric literal, a numeric literal or an arithmetic expression.

from-clause
Specifies the table name or table name's alias from which records are selected.

where-clause
Specifies search conditions or a combination of conditions.

Example

The following cursor retrieves all fields from CITIES records for which the POPULATION field is not missing:

```
exec sql
   declare inhabited cursor for
      select city, state, population
      from cities
      where population is not null;
```

Between Condition

Function

The **between** condition specifies an inclusive range of values to match.

Syntax

```
[database-field | scalar-expression] [not]
between scalar-expression-1 and scalar-
expression-2
```

Options

database-field
Specifies the field containing the values matching the inclusive range.

scalar-expression
A **scalar** expression is a field reference, an alphanumeric literal, a numeric literal or an arithmetic expression.

Example

The following cursor retrieves the CITY and STATE fields from cities with populations between 100000 and 125000:

```
exec sql
   declare midsized_cities cursor for
      select city, state from cities
      where population between 100000 and 125000;
```

Like Condition

Function

The **like** condition matches a string with the whole or part of a field value. The test is case-sensitive.

Syntax

```
database-field [not] like scalar-expression
[escape scalar expression-2]
```

Options

database-field
Specifies the field containing the values that are matched to the scalar expression.

scalar-expression
The **scalar** expression represents an alphanumeric value. It may be a literal, a host language variable or a dataase field. It can contain wildcard characters. Wildcard characters are:

Character	Matches
Underscore (_)	A single character
Percent sign (%)	Any sequence of characters, including none.

escape
Allows you to search for the predefined wildcard characters by instructing InterBase to treat the character following **escape**as itself.

scalar expression-2
Represents a single character. It can be a literal, a host language variable, or a database field.

Examples

The following cursor retrieves the CAPITAL and STATE from STATES records in which the CAPITAL field contains the string "ville" preceded or followed by any number of characters:

```
exec sql
   declare ville cursor for
      select capital, state
      from states
      where capital like '%ville%';
```

The following example uses the optional escape clause:

```
exec sql
   declare percent.cities cursor for
      select city, state from cities
      where city like "%c%%" escape "c"
```

In Condition

Function

The **in** condition lists a set of scalar expressions as possible values.

Syntax

```
scalar-expression [not] in (set-of-scalars)
set-of-scalars::= {scalar-expression-commalist|
   column-select-expression}

column-select-expression ::= select [distinct]
scalar-expression from-clause[where-clause]
```

Options

scalar expression
A database field, host language variable, quoted string or numeric literal.

from-clause
Specifies the table name or table name's alias from which records are selected.

where-clause
Specifies serach conditions or combination of conditions.

Example

The following cursor selects records from the CITIES table with city names that are in the specified set:

```
exec sql
   declare favorite_cities cursor for
      select city, state, population
      from cities
      where city in ('Boston', 'Providence',
'Albany');
```

Exists Condition

Function

The **exists** condition tests for the existence of at least one qualifying record identified by the select subquery.

Because the **exists** condition uses the parenthesized **select** statement to retrieve a record for comparison purposes, it requires only wildcard (*) field selection.

A predicate containing an **exists** condition is true if the set of records specified by *select-expression* includes at least one record. If you add **not**, the predicate is true if there are *no* records that satisfy the subquery.

Syntax

```
[not] exists (select * where-clause)
```

Options

where-clause
Specifies search conditions or a combination of conditions.

Example

The following cursor tests to see if at least one record that satisfies the condition exists:

```
exec sql declare exist_test cursor for
   select s.capitol, s.state from states s where
exists (select * from ski_areas sk where
      sk.state = s.state);
```

Troubleshooting

See Appendix A for a discussion of error handling.

See Also

See the entries in this chapter for:

- select expression
- scalar expression

See also the entries in Chapter 4 for:

- **delete**
- **update**

Scalar

Function

The **scalar** expression is a symbol or string of symbols used in predicates to calculate a value. InterBase uses the result of the expression when executing the statement in which the expression appears.

You can add (+), subtract (-), multiply (*), and divide (/) scalar expressions. Arithmetic operations are evaluated in the following order: addition, subtraction, multiplication, division. You can use parentheses to change the order of evaluation.

Syntax

```
scalar-expression ::= [-]scalar-term
      arithmetic-operator scalar-expression]

scalar-term ::= [-]scalar value

scalar-value ::= {field-expression|host-language
variable|constant-expression|
statistical function|(scalar-expression)

arithmetic-operator::= {+|-|*|/}
```

Options

The following sections describe the four options of the **scalar** expression:

- Field expression
- Constant expression
- Host language variable
- Statistical function

Field Expression

Function

The **field** expression references a database field.

Syntax

```
[table-name.|alias.]database-field]

table-name ::= [authorization_id.]
[database-handle.]{table-name|view-name}
```

Options

database-handle
Specifies the name associated with a database. You establsih the handle in a GDML **database** delcaration. You can also assign the handle to a specific database in a GDML **ready** statement.

The optional database handle is useful in multi-database applications in which databases are declared with the GDML **database** declaration.

table-name.
view-name.
alias.
Specifies the table, view, or alias (synonym for a table or view) in which the field is located. The alias is assigned to a table or a view in a *select-expression*.

database-field
Specifies the field.

authorization_id
The user name of the owner of the table or view.

Examples

The following cursor retrieves fields from the CITIES record that represents the city of Boston:

```
exec sql
   declare legume_village cursor for
      select city, state, altitude, latitude,
            longitude
      from cities
      where city = 'Boston';
```

The following cursor retrieves selected fields from CITIES with a population greater than 1,000,000:

```
exec sql
   declare big_cities cursor for
      select city, state, population
      from cities
      where population > 1000000;
```

The following cursor joins records from the CITIES and STATES tables:

```
exec sql
   declare city_states cursor for
      select c.city, s.state_name
      from states s, cities c
      where s.state = c.state;
```

Constant Expression

Function

The **constant** expression specifies a string of ASCII digits interpreted as a number or a quoted string of ASCII characters.

Syntax

```
{integer-string|decimal-string|float-string|
ascii-string}
```

Options

integer-string

Integer numeric strings are written as signed or unsigned decimal integers without decimal points. For example, the following are integers: *-14, 0, 9*, and *+47*.

decimal-string

Decimal numeric strings are written as signed or unsigned decimal integers with decimal points. For example, the following are decimal strings: *-14.3, 0.021, 9.0*, and *+47.9*.

float-string

Floating numeric strings are written in scientific notation (that is, *E-format*). A number in scientific notation consists of a decimal string mantissa, the letter *E*, and a signed integer exponent such as *7.12E+7* or *7.12E-7*.

ascii-string

Character strings are written using ASCII printing characters enclosed in single (') or double (") quotation marks. ASCII printing characters are shown in the following table:

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

Host Language Variable

Function

You use host language variables whenever you:

- Retrieve data from a database. SQL moves the values of database fields into host variables when it returns data.
- Solicit data from a user of your application. You need a host variable to hold the value until you can pass it to InterBase.
- Specify search conditions. When you specify the conditions for selecting records, you can either hard code a value or use a host variable. For example, both *where state = 'NH'* or *where state = :state* are valid search conditions.

Syntax

```
:host-language-variable
```

Example

The following code fragment uses host language variables:

```
exec sql select state, state_name, capital into
   :statecode, :name, :cap_city from states
   where state = :input_variable;
```

Statistical Function Expression

Function

A **statistical** function is an expression that calculates a single value from the values of a field in a table, view, or join.

Syntax

```
{count(*)|count (distinct field-expression)|
sum ([distinct] field-expression)|
avg ([distinct] field-expression)|
min (field-expression)|max (field expression)}
```

Note

If you are programming Pascal, put a space between the open parenthisis and the asterisk. Because Pascal uses the sequence "(*" for comments, failure to leave a space results in a compilation error.

Options

`count(*)`
Returns the number of selected rows.

`count [distinct]`
Returns the number of unique values for the field. You must specify distinct.

`sum [distinct]`
Returns the sum of values for a numeric field in all qualifying records. InterBase ignores null values.

`avg`
Returns the average value for a numeric field in all qualifying records. InterBase ignores null values.

`max`
Returns the largest value for the field.

`min`
Returns the smallest value for the field.

Example

The following program returns a count of records in the CITIES table, the maximum population, and the minimum population of cities in that table:

```
exec sql
   include sqlca;

main()
```

```
{
int counter;
int minpop,maxpop;

exec sql
select count ( * ), max (population), min
(population)
      into :counter, :maxpop, :minpop
      from cities;
printf("Count: %d\n", counter);
printf("Max Population: %d\n", maxpop);
printf("Min Population: %d\n", minpop);
}
```

Troubleshooting See Appendix A for a discussion of error handling.

See Also See the entries in this chapter for:

- Select expression
- Predicate

Select

Function

The **select** expression specifies the search and delivery conditions for record retrieval.

Syntax

```
select-clause [where-clause][grouping-
clause][having-clause]
```

The following sections describe the four options of the **select** expression:

- Select clause
- Where clause
- Group By clause
- Having clause

Select Clause

Function

The **select** clause lists the fields to be returned and the source table or view.

Syntax

```
select [distinct]{scalar-expression-commalist|*}
from from-item-commalist
from-item ::=  table-name [alias]
```

Options

distinct
Specifies only unique values are to be returned. InterBase considers the values in the *scalar-expression* list and returns only one set value for each group of records that meets the selection criteria. It does not return duplicate values.

scalar-expression
Lists the fields to be selected. Scalar expressions can also be host variables or constants.

*
The asterisk wildcard signifies all. In a select clause, using * selects all fields from the source table. You can use an asterisk in place of the full selection list. Although it is the preferred

form for the existential qualifier, **exists**, the wildcard is discouraged for all other uses. Changes to the database (for example, adding or reordering fields) cause the program to fail after its next precompilation.

`table-name`
Specifies the source table.

alias
Used for name recognition, and is associated with a table. An alias can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

The following considerations apply to case sensitivity in programming languages:

- For all languages except C, **gpre** is not sensitive to the case of the alias. For example, it treats "B" and "b" as the same character.
- For C programs, you can control the case sensitivity of the alias with the **either_case** switch when you preprocess your program.

Examples

The following cursor projects the SKI_AREAS table on the STATE field:

```
exec sql
   declare ski_states cursor for
      select distinct state from ski_areas;
```

The following example uses a join:

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


main ()
{
char    name [25], city [26], state [26];
long  population, pop_ind;
char  pop [12];
exec sql
    declare capitol_mayors cursor for
```

```
        select m.mayor_name, s.capitol,
       s.state_name, c.population from
   mayors m, states s, cities c
            where m.city = c.city and
            m.state = c.state and
            c.city = s.capitol and
            c.state = s.state
        order by s.state_name

exec sql open capitol_mayors;

exec sql fetch capitol_mayors into :name, :city,
    :state, :population :pop_ind;

while (!SQLCODE)
    {
    if (pop_ind < 0)
   sprintf (pop, "unknown");
    else
   sprintf (pop, "%d", population);
    printf ("%s is mayor of %s\n\t capitol of\
 %s population %s\n",
        name, city, state, pop);
   exec sql fetch capitol_mayors into :name, :city,
        :state, :population :pop_ind;
    }
if (SQLCODE != 100)
    {
    printf ("Unexpected SQLCODE %d\n", SQLCODE);
    gds_$print_status (gds_$status);
    }
}
```

The following example uses a wildcard in place of the selection list:

```
exec sql select city from cities c
   where exists c
      select * from ski_areas
      where city = c.city;
```

Where Clause

Function

The **where** clause specifies search conditions or combinations of search conditions.

Often you want only a subset of the records in a table. When you can describe the records you want by comparing values in the records to values you specify, InterBase selects and returns only those records you have described.

A statement, in which the choice is between the truth or falsity of a proposition, is called a "Boolean test" and is expressed by a predicate. See the entry for predicate in this chapter.

Syntax

```
where predicate
```

Example

The following cursor selects CITIES records for which the POPULATION field is not missing:

```
exec sql
   declare inhabited cursor for
      select city, state, population from cities
      where population is not null;
```

The following cursor joins two tables on the STATE field for cities whose population is not missing:

```
exec sql
   declare inhabited_join cursor for

      select c.city, s.state, c.population
      from cities c, states s
      where c.state = s.state
      and c.population is not null;
```

The following program selects the smallest city in each state that has at least two other cities with recorded population. A city qualifies as largest and smallest if it is the only city. If there are two cities, a city qualifies as the larger or smaller of the two:

```
exec sql
   include sqlca;

main ()

{
```

```
int pop;
char city [16];
char state_code [3];

exec sql
   declare small_cities cursor for
   select city, state, population
      from cities c1
      where c1.population = (
         select min (population)
            from cities c2
            where c2.state = c1.state)
      and 2 <= (
         select count ( * )
            from cities c3
            where c1.state = c3.state
            and c1.city <> c3.city
            and c3.population is not null)
   order by c1.state;
exec sql
   open small_cities;
exec sql
   fetch small_cities into :city, :state_code,
:pop;

while (SQLCODE == 0)
  {
  printf("The smallest city in %s is %s (pop:\
         %d)\n",
    state_code, city, pop);
  exec sql
   fetch small_cities into :city, :state_code,
:pop;
  }

exec sql
   close small_cities;
exec sql
   commit release;
}
```

Group By Clause

Function

The **group by** clause partitions the results of the **from** clause or **where** clause into control groups, each group containing all rows with identical values for the fields in the grouping clause's field list. Aggregates in the select clause and having clause are computed over each group. The select clause returns one row for each group.

The aggregate operations are count (**count**), sum (**sum**), average (**avg**), maximum (**max**), and minimum (**min**). See the entry for *scalar-expression* in this chapter.

You can compute an aggregate value in the *select-clause* and the *having-clause* of the select expression.

Syntax

```
group by database-field-commalist
```

Option

database-field
Specifies the field whose values you want to group. Each set of values for these fields identifies a group.

Example

The following example uses the **group by** clause. The cursor provides a total population by state of municipalities stored in the CITIES table. It includes only those cities for which the latitude and longitude information has been stored, which are located in states whose names include the word "New", and where the average population of cities in the state exceeds 200,000 people:

```
exec sql
declare total_pop cursor for
   select sum (c.population), s.state_name
   from cities c, states s
   where s.state_name like '%New%' and
      c.latitude is not null and
      c.longitude is not null and
      c.state = s.state
   group by s.state
   having avg (population) > 200000;
```

Having Clause

Function

The **having** clause specifies search conditions for groups of records. If you use the **having** clause, you must first specify a *group by-clause*.

The **having** clause eliminates groups of records, while the *where-clause* eliminates individual records. Generally speaking, you can use subqueries to obtain the same results. The main advantage to the use of this clause is brevity. However, some users may find that a more verbose query with a subquery is easier to understand.

Syntax

```
having predicate
```

Example

The following example uses the **having** clause. The cursor provides a total population by state of municipalities stored in the CITIES table. It includes only those cities for which the latitude and longitude information has been stored, which are located in states whose names include the word "New", and where the average population of cities in the state exceeds 200,000 people:

```
exec sql
   declare total_pop cursor for
      select sum (c.population), s.state_name
      from cities c, states s
      where s.state_name like '%New%' and
         c.latitude is not null and
         c.longitude is not null and
         c.state = s.state
      group by s.state
      having avg (population) > 200000;
```

Troubleshooting

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

See Also

See the entries in this chapter for:

- Predicate
- Scalar expression
- Select expression

Chapter 6
SQL Statements and Commands

This chapter contains entries for SQL statements.

Overview

InterBase supports the followng SQL statements for data definition and manipulation:

alter table
close
commit
create database
create index
create table
create view
declare cursor
declare table
declare statement
delete
describe
drop database
drop index
drop table
drop view
execute
execute immediate
fetch
grant
insert
open
prepare
revoke
rollback
select
update
whenever

Alter Table

Function

The **alter table** command drops a field from a table or adds a field to a table. Unlike the "standard" SQL **alter table** command, **gpre** lets you perform multiple drops and adds in one statement.

The **alter table** command is also supported in Dynamic SQL.

Syntax

```
alter table table-name operation-commalist
operation::= {add field-name datatype[not null]|
drop field-name}
```

Options

table-name
Specifies the table you want to change.

add
Adds a field to the specified table.

drop
Drops a field from the specified table.

field-name
Names the field you want to add or drop. If you add a field to a table, the field name must be unique among all field names in the table.

datatype
For a list of datatypes, see the entry in this chapter for **create table**.

not null
Disallows the null or missing value as a valid value for this field. You cannot disallow nulls when you add fields to a table containing records.

Example

The following statements alter tables by adding and dropping fields:

```
exec sql
   alter table states
      add type_of_govt char(3),
      add capital varchar(25);
```

```
exec sql
   alter table cities
      drop population;
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.
- SQLCODE of 100 indicates that no qualifying records were found.

Of these codes, the most likely to occur are -551 and -607:

- -551 indicates that a privilege was denied by an access control list. Check the access control lists for the database to make sure you have the right to manipulate database entities.
- -607 indicates that an attempt to update metadata failed. Check secondary messages to find out why the attempt failed.

See Also

See the discussion on defining metadata in the *Programmer's Guide*.

Close

Function

The **close** statement terminates the specified cursor. InterBase automatically releases resources associated with the closed cursor. The SQL **commit** and **rollback** statements, and GDML's **prepare** statement automatically close all cursors.

Once you have closed a cursor, you cannot issue any more **fetch** statements against that cursor unless you explicitly re-open it with another open statement. If you close a cursor, records selected for that cursor's active set are no longer available to your program. The active set of the cursor is said to be "undefined."

Syntax

```
close cursor-name
```

Option

cursor-name
Specifies the cursor to close.

Example

The following example declares a cursor, opens it, accesses records in its active set, and then closes the cursor:

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

main ()

{
char statecode[5];
char cityname[16];

exec sql
   declare bigcities cursor for
      select city, state from cities
      where population > 1000000;

exec sql
   open bigcities;
exec sql
   fetch bigcities into :cityname, :statecode;
```

```
printf ("\n");
while (!SQLCODE)
   {  printf ("%s is in %s\n", cityname,
      statecode);
      exec sql
      fetch bigcities into :cityname, :statecode;
   }
exec sql
   close bigcities;
exec sql
   rollback release;
}
```

Troubleshooting

InterBase returns errors if you:

- Fetch beyond the last record of an active set. InterBase automatically closes the cursor and returns an end-of-file error.
- Try to close a cursor that has not been opened. InterBase returns an error.

When you use the **close** statement with Dynamic SQL statements, you may also receive SQLCODE -504.

The Appendix describes these and other Dynamic SQL errors.

See Also

See the entries in this chapter for:

- **open**
- **commit**
- **rollback**

Commit

Function

The **commit** command:

- Ends the current transaction
- Makes the transaction's changes visible to other users
- Closes open cursors
- Does not affect the contents of host variables

The **commit** command is supported in Dynamic SQL.

Syntax

```
commit [work] [release]
```

Options

work
An optional word.

release
Breaks your program's connection to the attached database, thus making system resources available to other users. Do not release a database until you are finished with it. The cost of reopening the database is considerable.

Example

The following program illustrates the use of multiple cursors in a single transaction, terminated by a single commit that makes all changes permanent:

```
exec sql
   include sqlca;

#define TRUE  1
#define FALSE 0

main()
{
char newcity[26];
char oldcity[26];
char state[5];
int  first;
char option[4];

printf("Enter the city name that's changing: ");
```

```
gets(oldcity);
printf("Enter the new city name: ");
gets(newcity);
printf("Changing %s to %s in all tables\n",
oldcity, newcity);
exec sql
   declare cities_cursor cursor for
   select state from cities
   where city = :oldcity
   for update of city;
exec sql
   declare tourism_cursor cursor for
   select state from tourism
   where city = :oldcity
   for update of city;
exec sql
   declare ski_areas_cursor cursor for
   select state from ski_areas
   where city = :oldcity
   for update of city;
exec sql
   open ski_areas_cursor;
exec sql
   open tourism_cursor;
exec sql
   open cities_cursor;
first = TRUE;

while (SQLCODE == 0)
   {
    if (!first)
      {
   printf("Change %s, %s in cities? ",
         oldcity, state);
   gets(option);
      if (option[0] == 'y')
        exec sql update cities
      set city = :newcity
      where current of cities_cursor;
      }
      exec sql
      fetch cities_cursor into :state;
      first = FALSE;
```

```
    }
SQLCODE = 0;
first = TRUE;
while (SQLCODE == 0)
    {
     if (!first)
       {
       printf("Change %s, %s in tourism? ",
          oldcity, state);
       gets(option);
       if (option[0] == 'y')
       exec sql
          update tourism
          set city = :newcity
          where current of tourism_cursor;
       }
    exec sql
    fetch tourism_cursor into :state;
    first = FALSE;
    }
SQLCODE = 0;
first = TRUE;
while (SQLCODE == 0)
    {
    if (!first)
      {
       printf("Change %s, %s in ski areas? ",
          oldcity, state);
       gets(option);
       if (option[0] == 'y')
       exec sql
          update ski_areas
          set city = :newcity
          where current of ski_areas_cursor;
      }
        exec sql
       fetch ski_areas_cursor into :state;
       first = FALSE;
    }
exec sql
    close ski_areas_cursor;
exec sql
    close tourism_cursor;
```

```
exec sql
   close cities_cursor;
exec sql
   commit release
}
```

Troubleshooting

See the the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.
- SQLCODE = 100 indicates that no qualifying records were found.

See Also

See the entry in this chapter for **rollback**.

Create Database

Function

The **create database** command creates a database and its system tables.

Syntax

```
create database quoted-filespec
[pagesize=integer]
```

Options

quoted-filespec

Specifies the database 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 out in the **create database** command.

The file specification can contain the full pathname to another node in the network. File specifications for remote databases have the following form:

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

pagesize=*integer*

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. The advantage of a larger page size is that it allows a more shallow "tree" structure in the index. Each index bucket is one page long, so longer pages mean larger buckets and fewer levels in the index hierarchy. If you will have more than 50,000 records in any one table, you should use a page size of 2048 rather than the default.

Example

The following statement creates a database in the current directory:

```
exec sql
   create database 'employees.gdb';
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.

Of these codes, the most likely to occur is -607:

- – 607 indicates that an attempt to update metadata failed. Check secondary messages to find out why the attempt failed.

See Also

For more information about creating a database and for other database file options, see the chapter on creating a database in the *Data Definition Guide*.

For more information about SQL metadata operations, see the *Programmer's Guide*.

See also the entries in this chapter for:

- **create table**
- **create index**
- **create view**

Create Index

Function

The **create index** command defines an index for a table.

This command is also supported in Dynamic SQL.

Syntax

```
create [unique] [ascending|descending] index
index-name on table-name(field-name-commalist)
```

Options

`unique`
Disallows duplicate values in the index. The values for the indexed fields must be unique. If you try to store a value that already exists, the assignment operation fails.

`ascending|descending`
Specifies the order in which an index is built. If neither qualifier is specified, the default order is ascending.

`index-name`
Names the index. The index name must be unique among all index names in the database.

`table-name`
Identifies the table for which the index is defined.

`field-name`
Specifies a column name or list of field names, separated by commas, that comprise the index.

Note

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 the order by clause in the **select** statement.

Example

The following statements create a non-unique and unique index, respectively:

```
exec sql
   create index xxx on states (capital);
exec sql
   create unique index xyz on states (state);
```

The following example creates a descending index on the LENGTH field in the RIVERS table. :

```
exec sql
   create descending index longriv on rivers
   (length);
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.

Of these codes, the most likely to occur are -551 and -607:

- -551 indicates that a privilege was denied by an access control list. Check the access control lists for the database to make sure you have the right to manipulate database entities.
- -607 indicates that an attempt to update metadata failed. Check secondary messages to find out why the attempt failed.

See Also

See the discussion of defining metadata with SQL in the *Programmer's Guide*.

Create Table

Function

The **create table** command defines a table and its constituent fields.

This command is also supported in Dynamic SQL.

Syntax

```
create table table-name(field-definition-
 commalist)
 field-definition::= field-name datatype[not
null]
datatype::= {smallint|integer|date|
char(integer)| varchar(integer)|
decimal[(scale)]|float|long float}
```

Options

table-name
Names the table you want to create. A table name can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character and be unique among table names in the database.

field-name
Specifies the name you want for the field in the table. The field name must be unique among all field names in the table.

datatype
The following table lists the SQL datatype and what InterBase gives you.

SQL Datatype	InterBase Datatype
smallint	short
integer	long
date	date
char	char
varchar	varying
decimal	long

SQL Datatype	InterBase Datatype
float	float
longfloat	double

not null
Disallows the null or missing value as a valid value for this field.

unique
Creates unique indexes on a table You use the unique option to create unique indexes on a table. Unique can be used in two ways: as part of the field defintion clauses or as a separate clause. The following example shows both methods of using unique. It creates a unique index on the f1 field and another index on the combination of the f2 and f3 fields.

```
create table t1
(f1 smallint not null unique,
 f2 char(10),
 f3 integer,
 unique (f2, f3));
```

Usage

Using the **create table** command automatically invokes the SQL security scheme for that table. If you create a table, you are that table's owner and accordingly have all privileges for that table. You also have **grant** option for those privileges for that table. See the entries in this chapter for **grant** and **revoke** for further information on SQL security.

Note

You cannot assign a security class to tables created with the SQL **create table** command. Instead, you control access to these tables by using SQL **grant** and **revoke** commands.

Example

The following statements define tables:

```
exec sql
   create table states (
      state char(2) not null,
      state_name varchar(25),
      area integer,
```

```
        statehood date,
        capital varchar(25));

exec sql
   create table populations (
        state char(2) not null,
        census_1950 integer,
        census_1960 integer,
        census_1970 integer,
        census_1980 integer));
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.

Of these codes, the most likely to occur are -551 and -607:

- -551 indicates that a privilege was denied by an access control list. Check the access control lists for the database to make sure you have the right to manipulate database entities.
- -607 indicates that an attempt to update metadata failed. Check secondary messages to find out why the attempt failed.

See Also

See the discussion on defining metadata in the SQL section of the *Programmer's Guide*.

Create View

Function

The **create view** command creates a temporary view of data. When you create a view by using embedded SQL, the view definition is *not* stored on the database. As a result, you cannot access this definition through **qli** or **gdef**. A view definition should include a field name list and a specific list of fields to be selected from the source tables. This precaution protects view definitions from changes to the underlying tables. You can use any option of the record selection expression except the **first** and **sorted** clauses when using the **create view** command.

The **create view** command is also supported in Dynamic SQL.

Syntax

```
create view view-name[(field-name-commalist)]
as select-statement
```

Options

view-name
Names the view you want to create. The view name must be unique among all view names in the database.

field-name
Optionally names the fields for the view. If you choose not to supply a field name, gpre uses the field name as specified in the **select** statement that follows. Because the field names map chronologically to the list of selected fields in the select statement, you must specify all view field names or none.

If you supply the field name, it must be unique among all field names in the view.

select-statement
A **select** statement that specifies the selection criteria for records to be included in the view. Instead of the into clause used in queries, the list of selected fields maps to the list of field names for the view. If you use *select* * rather than a field list, the order is based on the value of the RDB$FIELD_POSITION field in the RDB$RELATION _FIELDS system table.

Example

The following statements define views:

```
exec sql
   create view half_mile_cities as
```

```
        select city, state, altitude from cities
           where altitude > 2500;

exec sql
   create view capital_cities as
      select c.city, s.state_name, c.altitude
      from cities c, states s where
      c.state = s.state and c.city = s.capital;
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.

Of these codes, the most likely to occur are -551 and -607:

- -551 indicates that a privilege was denied by an access control list. Check the access control lists for the database to make sure you have the right to manipulate database entities.
- -607 indicates that an attempt to update metadata failed. Check secondary messages to find out why the attempt failed.

See Also

See the discussion on defining metadata in the SQL section of the *Programmer's Guide*.

Declare Cursor

Function

The **declare cursor** declaration defines a cursor by associating a name with the active set of records determined by a select statement.

Syntax

```
declare cursor-name cursor for select statement
[for update of database-field-commalist]
[order by sort-key-commalist]
sort-key::= field-reference[asc|desc]
field-reference::= {database-field|integer}
```

Options

cursor-name
Provides a name for the cursor you are declaring.

select-statement
An SQL **select** statement that specifies search conditions to determine the active set of the cursor.

for update of
Indicates that your program may update one or more fields of records in the active set. Standard SQL restricts you to updating only the listed fields; however, InterBase does not enforce this restriction.

order by
Specifies the order in which the retrieved records are to be delivered to the program.

database-field
Specifies the fields in the source table(s) to sort by.

integer
References the field position of one of the fields in the select statement. Specifies the field to sort by.

Examples

The following example declares a cursor, a search condition, and a sorting clause:

```
exec sql
   include sqlca;
```

```
main ()
{
char statecode[5];
char cityname[16];
int min_pop;
char option[4];

min_pop = 100;

/* the crude way */

exec sql
   delete from cities
   where population < :min_pop;

exec sql
   rollback;

/* with finesse */

exec sql
   declare small_cities cursor for
      select city, state
      from cities
      where population < :min_pop;
exec sql
   open small_cities;
exec sql
   fetch small_cities into :cityname, :statecode;

while (!SQLCODE)
{
printf ("Eliminate %s, %s? ", cityname, statecode);
   gets(option);
   if (option[0] == 'Y') or (option[0] == 'y')
   exec sql
      delete from cities
      where current of small_cities;
   exec sql
      fetch small_cities into :cityname,
:statecode;
}

exec sql
   close small_cities;
```

```
exec sql
   rollback release;

}
```

The following Pascal example declares a cursor for two tables:

```
exec sql
   include sqlca;

main()
{

char city[16];
char lat[16];
char long[16]
char state[21]

exec sql
   declare city_state_join cursor for
      select c.city, s.state_name, c.latitude,
c.longitude
      from cities c, states s where c.state =
s.state
      order by s.state, c.city;

exec sql
   open city_state_join;
exec sql
   fetch city_state_join into :city, :state, :lat,
:long;

while (SQLCODE == 0)
{
   printf ("%s, %s, %s, %s\n", city, state,
      lat, long);
   exec sql
      fetch city_state_join into :city, :state,
:lat, :long;
}

exec sql
   rollback release;
}
```

The following program declares a cursor with the union of three tables:

```
   include sqlca;

main()
{
char city[26];
char state[26];

exec sql
   declare all_cities cursor for
      select city, state from cities
      union
      select city, state from ski_areas
      union
      select capital, state from states
      order by 2, 1;
exec sql
   open all_cities;
exec sql
   fetch all_cities into :city, :state;

while (SQLCODE == 0)
{
   printf ("%s, %s\n", city, state);
   exec sql
      fetch all_cities into :city, :state;
}
exec sql
   rollback release;
}
```

Troubleshooting

The declare cursor statement is not executed, so it does not produce runtime errors. If you have syntax errors in your statement declaration, gpre tries to diagnose it and provide you with an explanatory error message.

See Also

See the entry for **select** in this chapter.

Declare Statement

Function

The **declare statement** statement identifies names of Dynamic SQL statements that will be prepared, or prepared and executed in the program. This statement is not required, but provides internal documentation.

Syntax

```
declare operation-name-commalist statement
```

Options

operation-name
Specifies the Dynamic SQL statements that will be prepared, or prepared and executed in a program.

Example

The following statement declares Q1 to be the name of an SQL operation that will be prepared or executed in the program:

```
exec sql declare q1 statement;
```

Troubleshooting

The declare statement statement is not executed, so it does not produce runtime errors. If you have syntax errors in your cursor declaration, gpre tries to diagnose it and provide you with an explanatory error message.

See Also

See also the entry in this chapter for:

- **execute**
- **execute immediate**
- **prepare**

Declare Table

Function

The **declare table** statement establishes the structure of a table you have not yet created. If you are creating a new database, or adding tables to an existing database, use the **declare table** statement. The **declare table** statement gives the preprocessor a description of the new tables. Ordinarily, **gpre** gets metadata descriptions from the existing database. When you are creating or adding tables to a database, **gpre** requires you to declare new tables, so it can validate field usage and datatypes.

Syntax

```
declare table-name table (field-definition-
  commalist)
  field-definition::= field-name datatype[not
    null]
  datatype::= {smallint|integer|date|
char(integer)| varchar(integer)|
decimal[(scale)]|float|long float}
```

Options

table-name

Names the table you want to create. A table name can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character and be unique among table names in the database.

field-name

Specifies the name you want for the field in the table. The field name must be unique among all field names in the table.

datatype

The following table lists the SQL datatype and what InterBase gives you.

SQL Datatype	InterBase Datatype
smallint	short
integer	long
date	date
char	char

SQL Datatype	InterBase Datatype
varchar	varying
decimal	long
float	float
longfloat	double

Example

The following program uses the declare table statement:

```
exec sql include sqlca;

main()
{

exec sql
    create database "foo.gdb";

exec sql
    declare recordings table (
   number varchar (10) not null,
   name varchar (40),
   performer_last_name varchar (20),
   performer_first_name varchar (10),
   composer_last_name varchar (20),
   compose_first_name varchar(10),
   type char(2));

exec sql
    declare performers table (
   last_name varchar(20),
   first_name varchar(10),
   nationality varchar(10));

exec sql
    create table recordings (
   number varchar (10) not null,
   name varchar (40),
   performer_last_name varchar (20),
   performer_first_name varchar (10),
   composer_last_name varchar (20),
   compose_first_name varchar(10),
```

```
   type char(2));

if (SQLCODE) gds_$print_status (gds_$status);

exec sql
    create table performers (
   last_name varchar(20),
   first_name varchar(10),
   nationality varchar(10));

if (SQLCODE) gds_$print_status (gds_$status);

exec sql declare c cursor for
   select*
   fromrecordings;

if (SQLCODE) gds_$print_status (gds_$status);

exec sql
    create unique index i1 on recordings (number);

if (SQLCODE) gds_$print_status (gds_$status);

exec sql
    create unique index i2 on recordings (number,
type);

if (SQLCODE) gds_$print_status (gds_$status);

exec sql
    create view performances as
   select r.name, r.number, p.last_name,
      p.first_name
   from recordings r, performers p where
       r.performer_last_name = p.last_name and
       r.performer_first_name = p.first_name;

if (SQLCODE)
    gds_$print_status (gds_$status);

exec sql
    alter table performers
   add group char (8),
```

```
    add money decimal (6, 2),
    drop nationality;

if (SQLCODE) gds_$print_status (gds_$status);
}
```

Troubleshooting

The declare table statement is not executed, so it does not produce runtime errors. If you have syntax errors in your table declaration, gpre tries to diagnose it and provide you with an explanatory error message.

See Also

See the entries in this chapter for:

- **create table**
- **declare cursor**
- **declare statement**

Delete

Function

The **delete** statement erases records in a table or in the active set of a cursor.

If you do not provide a search condition (where...), all records in the specified table are deleted. Be careful with this option.

Syntax

```
delete from table-name
[where predicate|where current of cursor-name]
```

Options

table-name
Specifies the table from which a record is to be deleted.

where predicate
Determines the record to be deleted.

where current of *cursor-name*
Specifies that the current record of the active set is to be deleted. This form of **delete** must follow:

- The declaration of the cursor with a **declare cursor** statement
- The opening of that cursor with an **open** statement
- The retrieval of a record from the active set of that cursor with a **fetch** statement

Examples

The following statement erases the entire table named VILLAGES (which does not exist in the sample database):

```
exec sql delete from villages;
```

The following program deletes all records from CITIES with a population less than that of the host variable MIN_POP:

```
exec sql
   include sqlca;

main()
{
char statecode[5];
char cityname [16]
int min_pop;
```

```
char option[4];
begin

min_pop = 100;

/* the crude way */

exec sql
   delete from cities
   where population < :min_pop;

exec sql
   rollback;

/* with finesse */

exec sql
   declare small_cities cursor for
      select city, state
      from cities
      where population < :min_pop;
exec sql
   open small_cities;
exec sql
   fetch small_cities into :cityname, :statecode;

while (!SQLCODE)
{
   printf ("Eliminate %s, %s ?",
       cityname, statecode?);
   gets(option);
   if (option == 'Y') or (option == 'y')
   exec sql
      delete from cities
      where current of small_cities;
   exec sql
      fetch small_cities into
:cityname,:statecode;
}

exec sql
   close small_cities;
exec sql
   rollback release;
}
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.
- SQLCODE = 100 indicates that no qualifying records were found.

See Also

See also the entries in this chapter for:

- **declare cursor**
- **open**
- **fetch**
- **select**

See also the entry in Chapter 5 for predicate.

Describe

Function

The **describe** statement retrieves the contents of the SQLDA (that is, SQLD and for each value to be returned, the SQLTYPE, SQLLEN, and SQLNAME) for use in allocating buffers with Dynamic SQL statements. If you have used the **into sqlda** option on the prepare statement, you do not have to use the **describe** statement.

If the value returned for SQLD is larger than SQLN, you must allocate a larger SQLD and re-issue the **describe** statement.

Syntax

```
describe operation-name into sqlda-structure
```

Options

operation-name
A Dynamic SQL statement that has been processed with the prepare statement.

sqlda-structure
Specifies the SQLDA into which the output of the describe statement is placed.

Example

The following statement retrieves information about the output of a prepared select statement:

```
exec sql describe q1 into sqlda;
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.

The **describe** statement may result in SQLCODE -518 being returned.

The Appendix describes this and other Dynamic SQL errors.

See Also

See also the entries in this chapter for:

- **execute**
- **prepare**

Drop Database

Function

The **drop database** command deletes an entire database.

Syntax

```
drop database quoted-filespec
```

Option

quoted-filespec
Specifies the database you want to drop.

Example

The following example deletes the entire database:

```
exec sql drop database 'phones.gdb';
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.

Of these codes, the most likely to occur are -551 and -607:

- -551 indicates that a privilege was denied by an access control list. Check the access control lists for the database to make sure you have the right to manipulate database entities.
- -607 indicates that an attempt to update metadata failed. Check secondary messages to find out why the attempt failed.

See Also

See the discussion on defining metadata with SQL in the *Programmer's Guide*.

Drop Index

Function

The **drop index** command deletes an index. If the index you are attempting to delete is in use, your program waits until the index is free before deleting it.

This command is supported in Dynamic SQL.

Syntax

```
drop index index-name
```

Option

index-name
Specifies the index you want to delete.

Example

The following example deletes an index:

```
exec sql drop index statesnames;
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.
- SQLCODE of 100 indicates that no qualifying records were found.

Of these codes, the most likely to occur are -551 and -607:

- -551 indicates that a privilege was denied by an access control list. Check the access control lists for the database to make sure you have the right to manipulate database entities.
- -607 indicates that an attempt to update metadata failed. Check secondary messages to find out why the attempt failed.

See Also

See the discussion of defining metadata in the SQL part of the *Programmer's Guide*.

Drop Table

Function

The **drop table** command deletes a table. If you try to drop a table used in a view, computed field, or trigger, InterBase returns an error. You must delete the view, field or trigger before you delete the table. If the table is in use, your program waits until the table is free before deleting it. The **drop table** command also deletes all indexes on the deleted table.

This command is supported in Dynamic SQL.

Syntax

```
drop table table-name
```

Option

table-name
Specifies the table to drop.

Example

The following examples deletes a table:

```
exec sql drop table tourism;
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.

Of these codes, the most likely to occur are -551 and -607:

- 551 indicates that a privilege was denied by an access control list. Check the access control lists for the database to make sure you have the right to manipulate database entities.
- -607 indicates that an attempt to update metadata failed. Check secondary messages to find out why the attempt failed.

See Also

See the discussion of defining metadata in the SQL part of the *Programmer's Guide*.

Drop View

Function

The **drop view** command deletes a view. If you try to drop a view used in another view, a computed field, or a trigger, InterBase returns an error. You must delete the view, field or trigger before you delete the view. The tables that comprise the view are not affected.

This command is supported in Dynamic SQL.

Syntax

```
drop view view-name
```

Option

view-name
Specifies the view to drop.

Example

The following example deletes a view:

```
exec sql drop view colonies;
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.

Of these codes, the most likely to occur are -551 and -607:

- -551 indicates that a privilege was denied by an access control list. Check the access control lists for the database to make sure you have the right to manipulate database entities.
- -607 indicates that an attempt to update metadata failed. Check secondary messages to find out why the attempt failed.

See Also

For a discussion of the SQLDA, see the chapter on setting up an SQLDA in the part on DSQL in the *Programmer's Guide.*

Execute

Function

The **execute** statement runs a Dynamic SQL string that has been compiled with the **prepare** statement.

Syntax

```
execute operation-name [using descriptor
descriptor-name]
```

Options

operation-name
Specifies the name of the prepared string to run.

descriptor-name
Specifies that the values corresponding to the prepared string's parameters are passed through the SQL descriptor area. The InterBase implementation of Dynamic SQL does not support the use of host variables to pass values.

Example

The following statement executes a Dynamic SQL string:

```
EXEC SQL EXECUTE Q1;
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.
- SQLCODE = 100 indicates that no qualifying records were found.

The **execute** statement may result in SQLCODEs 303, 510, 518, and 804 being returned.

The Appendix describes these and other Dynamic SQL errors.

See Also

See also the entry in this chapter for:

- **declare statement**
- **describe**
- **prepare**

For a discussion of the SQLDA, see the chapter on setting up an SQLDA in the part on DSQL in the *Programmer's Guide.*

Execute Immediate

Function

The **execute immediate** statement prepares and executes a Dynamic SQL string. By using this statement, you eliminate the need to issue a **prepare** statement first.

Syntax

```
execute operation-name[using descriptor
descriptor-name]
```

Options

operation-name
Specifies the name of the prepared string to run.

descriptor-name
Specifies that the value corresponding to the prepared string's parameters are passed through the SQL descriptor area. The InterBase implementation of Dynamic SQL does not support the use of host variables to pass values.

Example

The following statement executes a Dynamic SQL string:

```
EXEC SQL EXECUTE IMMEDIATE Q1;
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.
- SQLCODE of 100 indicates that no qualifying records were found.

The **execute immediate** statement may result in SQLCODEs -103, -104, -204, -206, and -510 being returned.

The Appendix describes these and other Dynamic SQL errors.

See Also

See the entries in this chapter for:

- **declare statement**
- **describe**
- **execute**
- **prepare**

For a discussion of the SQLDA, see the chapter on setting up an SQLDA in the part on DSQL in the *Programmer's Guide.*

Fetch

Function

The **fetch** statement advances the position of the cursor to the next record of the active set.

Syntax

The **fetch** statement is also used in Dynamic SQL. The syntax diagram shows the two forms of the statement.

Embedded SQL form:

```
fetch cursor-name[into host-item-commalist]
host-item ::= host-variable[indicator]
indicator ::= [indicator]:indicator-variable
indicator-variable ::= integer
```

Dynamic SQL form:

```
fetch cursor-name using descriptor descriptor-
name
```

`cursor-name`
Specifies the open cursor from which you want to fetch records.

`host-item`
Specifies a host language variable into which fields from records in the active set of the cursor will be fetched. The *into* list is not required if the fetch gets records to be deleted or updated; however, if you display the record before you delete or update it, you need the *into* list.

`descriptor-name`
Specifies the SQL descriptor area (SQLDA) associated with the cursor. The SQLDA is used in DSQL to communicate information between a program and InterBase.

`indicator-variable`
An integer that gets the missing value for the field immediately preceding it. For example,

```
fetch c into :builder, :model
```

retrieves two columns into host-variables build and model.

```
fetch c into :builder :model
```

retrieves one column into host-variable builder and sets indicator-variable model to the value of the missing flag for column builder in this row.

Note

InterBase enforces SQL's requirement that the number of columns in a **fetch** must equal the number of columns in the **declare cursor**.

Usage

Keep the following points in mind when you use the fetch statement:

- If the **fetch** statement immediately follows an **open** statement, the cursor is set before the first record in that cursor. The **fetch** statement advances the cursor to the first record.
- If you try to fetch beyond the last record in the active set, InterBase automatically closes the cursor and returns an end-of-file message.
- Once the **fetch** statement has advanced the cursor, it writes the fields of that record into the listed host variables or the buffers pointed to by the SQLDA descriptor.
- If you want to update or delete a record in a cursor's active set, you must first fetch it. You can then use the **update** statement to modify one or more of its field values, or use the **delete** statement to erase it.

To loop through the records selected by the cursor, enclose the **fetch** statement in a host language looping construct. Terminating the loop when the SQLCODE is non-zero is usually good practice, because the loop ends when the active set of the cursor is finished.

Because the **select** substatement of the **declare cursor** statement explicitly lists database field names, you must be sure the fields you provide to receive the data match the database fields in order, size, and datatype.

For example, in an embedded SQL program, the third host variable you list in the **into** clause of the **fetch** statement must be of the same datatype and length as the third field listed in the **select** substatement of the cursor declaration. In a Dynamic SQL program, the third parameter in the output SQLDA must match the third field listed in the **select** substatement of the cursor declaration. Thus, if the third field listed in the select substatement

is PHONE (a ten character fixed length text field in the CROSS_-COUNTRY relation) then one of the following must be true:

- In an embedded SQL **fetch** statement, the third host language variable in the **into** clause must be a character datatype, and at least 10 characters long.
- In a Dynamic SQL **fetch** statement, the third parameter in the SQLDA must provide storage for at least 10 characters.

Fields in the **select** substatements are matched to fields listed in the **into** clause of the SQLDA named in the **using** descriptor clause by the order of the listing. If the order of the two lists is different, the wrong values are assigned. Using the **select * from** *table* syntax is discouraged in cursors because adding or dropping fields in that table causes the **select** list to change. This in turn causes errors at runtime because the **fetch into** list or the SQLDA is not changed to match.

Examples

The following example declares a cursor, opens it, accesses records in its active set, and then closes the cursor:

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

main()
{

char statecode[3];
char cityname[26];

exec sql
   declare bigcities cursor for
      select city, state from cities
      where population > 1000000;

exec sql
   open bigcities;
exec sql
   fetch bigcities into :cityname, :statecode;

printf ("\n");
while (SQLCODE == 0)
   {
```

```
    begin
       printf ("%s is in %s\n", cityname,
          statecode);
       exec sql
          fetch bigcities into :cityname,
          :statecode;
}

exec sql
   close bigcities;
exec sql
   commit release;
}
```

The following statements are from a Dynamic SQL program that retrieves records through a cursor:

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

setup_buffer (buffer, sqlda);

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

Troubleshooting

See the Appendix for a discussion of error handling.

The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.
- SQLCODE = 100 indicates the end of the input stream.

When you use the **fetch** statement with Dynamic SQL strings, you may also receive SQLCODEs -303, -504, and -804.

The Appendix describes these and other Dynamic SQL errors.

See Also

See the entries in this chapter for:

- **open**
- **declare cursor**
- **select**
- **update**
- **delete**
- **whenever**

See the section on DSQL in the *Programmer's Guide*.

For a discussion of the SQLDA, see the chapter on setting up an SQLDA in the part on DSQL in the *Programmer's Guide*.

Grant

Function

The **grant** command defines privileges for users for designated tables and views. It can also grant a user the ability to pass along privileges. A table's owner is the only user to have automatic grant authority for that table. To pass the ability to grant privileges to a user, the **grant** statement must contain the **with grant option** clause.

The **grant** command is supported in Dynamic SQL.

Syntax

```
grant privilege-comma-list on table-name|view-
name to user [with grant option]
privilege::= {all [privileges]|select|delete|
insert|update (column-list)}
user::=public|userid-comma-list
```

Options

table-name
Specifies the table to which you assign privileges.

view-name
Specifies the view to which you assign privileges.

user
Specifies the user assigned privileges.

with grant option
Passes grant authority along to the user(s) specified in the **grant** command. This is valid for only those privileges authorized in the grant statement.

privilege
Specifies the operations for which a user has authority.

Privilege	Allows User to
All	Select, delete, insert, update
Select	Retrieve records from a table or view
Delete	Eliminate records from a table or view

Privilege	Allows User to
Insert	Store new records in a table or view
Update	Change the value of one or more fields in the existing records in a table or view

If a view is a subset of a table, it is updated directly. If a view is either a join of two or more tables, or a join of table to itself, triggers must be in place in order for the records in the tables to change.

public | *userid*
Specifies which authorized users have access to privileges for a table or view. **Public** incorporates all authorized user ids.

Usage

Once you have secured a table using SQL, you should use only SQL to further secure it. Do not use the InterBase security class system in combination with SQL security.

Examples

The following example grants select and delete privileges to a user and gives that user the authority to grant other users select and delete privileges:

```
exec sql
   grant select, delete on cities to julie with
   grant option;
```

The following example grants update privileges to a user for specific fields in a table:

```
exec sql
   grant update state_name, capital on states to
   john;
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.

Of these codes, the most likely to occur are -551 and -607:

- -551 indicates that a privilege was denied by an access control list. Check the access control lists for the database to make sure you have the right to manipulate database entities.
- -607 indicates that an attempt to update metadata failed. Check secondary messages to find out why the attempt failed.

See Also

See the entry for **revoke** in this chapter.

For more information on triggers, see the discussion of preserving data integrity in the *Data Definition Guide*.

See the chapter on securing data and metadata in the *Data Definition Guide*.

Insert

Function

The **insert** statement stores a new record into the specified table.

You can assign field values by inserting values, by picking up values from an existing record, or by a combination of both.

Syntax

```
insert into table-name[database-field-commalist]
{values insert-item-commalist|select-statement}
insert-item::=
   {constant|host-variable-expression|null}
```

Options

table-name
Specifies the table into which you want to store a new record.

database-field
Lists the field in *table-name* for which you are providing a value.

SQL by itself does not support manipulation of the blob datatype. You can store a null value for a blob field, but you must use GDML or **gds** calls to create the blob, then use the SQL assignment to include the blob in the new record.

If the field you are assigning is a date, you cannot handle the field directly with SQL. Instead, you must use date handling functions such as GDML's **gds_$encode_date** to convert your external date representation to a host variable in the InterBase date format (that is, an array of two 32-bit integers). Then use the SQL assignment to assign the host variable to the database field.

Note

The database field list is optional. If it is omitted, values are assigned to all the fields in the table in their normal order. Leaving out the field list is *not* recommended because changes to the table, such as adding or reordering fields, will cause the assignment list to change without warning when the program is next precompiled with **gpre**.

insert-item
Provides a value for *database-field*. The value can be a constant, host variable, or null.

select-statement
Specifies that the values for the new record are to come from the record identified by a select statement.

constant
Specifies a string of ASCII digits interpreted as a number or as a quoted string of ASCII characters.

expression
An arithmetic computation using host variables, constants, or, in the **select** statement, selected fields.

host-variable
Variable used for data transfer between a host language and InterBase.

null
Inserts a null value in a field.

Examples

The following program stores a record, assigning quoted constants for field values:

```
exec sql
   include sqlca;

main()
{

exec sql
   insert into river_states
   (river, state)
   values ('Croton', 'NY');

exec sql
   commit release;
}
```

The following program stores a new record into STATES using host variables and **null** as sources for values:

```
exec sql
   include sqlca;
```

```
main()
{
char state[3];
char state_name[21];
char capital[16];
date  :  gds_$quad;
date_array:gds_$tm;

date_array.tm_sec := 0;
date_array.tm_min := 0;
date_array.tm_hour := 0;
date_array.tm_mday := 1;
date_array.tm_mon := 1;
date_array.tm_year := 90;
date_array.tm_wday := 0;
date_array.tm_yday := 0;
date_array.tm_isdst := 0;

gds_$encode_date (date_array, date);
state := 'GU';
state_name := 'Guam';
capital := 'Agana';
exec sql
   insert into states
   (state, state_name, area, capital, statehood)
   values (:state, :state_name, null, :capital,
:date);

exec sql
   commit release;
}
```

The following program stores a new record using values from an existing record and the value of a host variable for assignments:

```
exec sql
   include sqlca;
main()
{

char villeancienne[26];
char villenouvelle[26];

printf ("Enter city to clone: ");
gets (villeancienne);
```

```
printf ("Enter new name for city: ");
gets (villenouvelle);

exec sql insert into cities (city, state,
population,
      altitude, latitude_degrees,
         latitude_minutes,
      latitude_compass, longitude_degrees,
         longitude_minutes,
      longitude_compass)
   select :villenouvelle, state, population,
      altitude, latitude_degrees,
         latitude_minutes,
      latitude_compass, longitude_degrees,
         longitude_minutes,
      longitude_compass
   from cities where city = :villeancienne;
exec sql
   commit
   release
}
```

The following program uses the non-recommended form of the **insert** statement, in which the database field list is omitted:

```
exec sql
   include sqlca;

main()
{
char state[3];
char state_name[21];
char capital[26];

state = 'GU';
state_name = 'Guam';
capital = 'Agana';
exec sql
   insert into states
   values (:state, :state_name, null, null,
         :capital);
exec sql
   commit release;
}
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.
- SQLCODE = 100 indicates that no qualifying records were found. This code occurs when the source of values is a **select** subquery that returned no records.

See Also

See the entry for **select** in this chapter.

Open

Function

The **open** statement activates a cursor. This statement causes InterBase to evaluate the search conditions associated with the specified cursor. Once the access method has determined the set of records that satisfies the query, it activates the cursor and makes the selected records the "active set" of that cursor.

The access method then places the cursor itself before the first record in the active set. If you want to retrieve or update records in that set, use the fetch statement. Once you open the cursor, the first fetch statement operates on the very first record in the active set. Subsequent fetch statements advance the cursor through the results table associated with that cursor.

The access method does not re-examine the host variables or values passed to the search conditions until you close the cursor and re-open it. Changes you make to their values are not reflected in the active set until you close and re-open the cursor.

If someone else accesses the database after you open a cursor, makes changes, and commits them, the active set may be different the next time you open that cursor if you commit your transaction.

Syntax

```
open cursor-name [using descriptor
descriptor-name]
```

Options

cursor-name
Specifies the declared cursor you want to access.

descriptor-name
Specifies that the values corresponding the prepared statement's parameters are passed through the SQLDA descriptor. This clause is used only in Dynamic SQL.

Example

The following example declares a cursor, opens it, accesses records in its active set, and then closes the cursor:

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

```
main()
{
char statecode[3];
char cityname[26];

exec sql
   declare bigcities cursor for
      select city, state from cities
      where population > 1000000;

exec sql
   open bigcities;
exec sql
   fetch bigcities into :cityname, :statecode;

printf (" ");
while (!SQLCODE)
{
   printf ("%s is in %s\n", cityname, statecode);
   exec sql
      fetch bigcities into :cityname, :statecode;
}

exec sql
   close bigcities;
exec sql
   rollback release;
}
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.
- SQLCODE = 100 indicates that no qualifying records were found.

When you use the **open** statement with Dynamic SQL strings, you may also receive SQLCODEs -303, -504, and -804.

The Appendix describes these and other Dynamic SQL errors.

See Also

See the entries in this chapter for:

- **declare cursor**
- **fetch**
- **close**
- **commit**
- **rollback**
- **whenever**

Prepare

Function

The **prepare** statement processes a Dynamic SQL statement. It:

- Accepts an SQL statement via a host language variable
- Checks the statement for errors
- Compiles the statement into a structure that can be executed by an SQL **execute** statement

Syntax

```
prepare operation-name[into sqlda_variable]from
:variable
```

Options

operation-name
Provides a name for the SQL operation you are describing. The name can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character (A—Z, a—z).

Except for C programs, **gpre** is not sensitive to the case of the name string. For example, it treats "B" and "b" as the same character. For C programs, you can control the case sensitivity of the name with the **either_case** switch when you preprocess your program.

into *sqlda_variable*
Instructs the preprocessor to place information about output fields into the SQL descriptor area. This step is equivalent to using the describe statement.

:variable
A host language variable declared to contain varying string data.

Example

The following statement prepares a Dynamic SQL statement from a host variable statement:

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

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.

- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.

The **prepare** statement may result in SQLCODEs -103, -104, -204, and -206 being returned.

The Appendix describes these and other Dynamic SQL errors.

See Also

See also the entries in this chapter for:

- **declare statement**
- **describe**
- **execute**

Revoke

Function

The **revoke** command takes privileges away from a user for a designated table or view. Only the user who grants a privilege can revoke that privilege. A revoke command does not effect privileges a user may have received from other grant command. The revoke command has a cascading effect on any privileges that were passed on through the with grant option clause in the grant command.

The revoke command is supported in Dynamic SQL.

Syntax

```
revoke privilege-comma-list on table-name|view-
name from userid-comma-list
```

Options

privilege
Specifies the operations for which a user has authority.

Privilege	Allows User to
All	Select, delete, insert, update
Select	Retrieve records from a table or view
Delete	Eliminate records from a table or view
Insert	Store new records in a table or view
Update	Change the value of one or more fields in the existing records in a table or view

table-name
Specifies the table from which you take away privileges.

view-name
Specifies the view from which you take away privileges.

userid
Specifies the user whose privileges you remove.

Example

The following example takes the select privilege away from a user for the CITIES table:

```
exec sql
   revoke select on cities from julie;
```

In the following example, John grants Julie select and delete privileges on a table that he created, and he gives her the ability to pass the grant privilege to other users:

```
exec sql
   grant select, delete on rivers to julie with
   grant option;
```

Julie can now pass the select privilege for the RIVERS table on to Dana:

```
exec sql
   grant select on rivers to dana;
```

If John decides to revoke Julie's select privilege for the RIVERS table, the revoke cascades through Julie's grant statement and also takes away Dana's select privilege:

```
exec sql
   revoke select on rivers from julie;
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.

Of these codes, the most likely to occur are -551 and -607:

- -551 indicates that a privilege was denied by an access control list. Check the access control lists for the database to make sure you have the right to manipulate database entities.
- -607 indicates that an attempt to update metadata failed. Check secondary messages to find out why the attempt failed.

See Also

See the entry in this chapter for grant.

Rollback

Function

The **rollback** command restores the database to its state prior to the current transaction. It also closes open cursors.

The **rollback** command is supported in Dynamic SQL.

Syntax

```
rollback [work][release]
```

Options

work
An optional word.

release
Breaks your program's connection to the attached database, thus making system resources available to other users.

Example

The following code segment shows the use of rollback in an error condition:

```
exec sql begin declare section;
exec sql end declare section;
main ()
{

exec sql
    insert into rivers (river, source, outflow,
   length) select "Assabet", state, "Charles", 35
   from cities where city = "Maynard";

if (SQLCODE)
    {
    printf ("Encountered SQLCODE %d\n", SQLCODE);
    if (SQLCODE != 100)
   {
   printf ("    expanded error message -\n");
   gds_$print_status (gds_$status);
   exec sql
        rollback release;
   }
    }
else
```

```
    exec sql commit release;
}
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.

See Also

See the entries in this chapter for:

- **commit**
- **whenever**

Select

Function

The **select** statement finds the record(s) of the tables specified in the **from** clause that satisfy the given search condition.

You can use the **select** statement by itself or within a **declare cursor** statement.

If you use the **select** statement by itself:

- The search conditions you specify should return at most one record.

  For example, the search condition references a field for which duplicate values have been disallowed.
- InterBase sets SQLCODE to -1 if there is more than one qualifying record.
- The **select** statement requires the **into** clause.

If you use the **select** statement within a **declare cursor** statement:

- The search condition can identify an arbitrary number of records.
- Remember that **declare cursor** is only declarative. Before you can retrieve records via the cursor, you must **open** it and **fetch** records sequentially.
- You cannot use the **into** clause.

Syntax

```
select-statement ::=  union-expression
[ordering-clause]
union-expression::= select-expression
[into-clause][union union-expression]
ordering-clause::= order by sort-key-commalist
sort-key::= {database-field|integer}[asc|desc]
into-clause::= into host-variable-commalist
```

Options

union-expression
Creates dynamic tables by appending tables. The source tables should have identical structures or at least share some common fields.

select-expression
Creates a stream of records which is input to the union, if a union is present. In the absence of a union, it produces the active set of the **select** statement.

ordering-clause
Returns the record stream sorted by the values of one or more *database-field*s. You can sort a record stream alphabetically, numerically, by date, or by any combination.

The *database-field* is called the *sort key*. You can construct an *ordering-clause* that includes as many sort keys as you want.

For each sort key, you can specify whether the sorting order is **asc** (ascending, the default order for the first sort key) or **desc** (descending). Unlike GDML's sort clause, the SQL sorting order is *not* "sticky."

into-clause
Specifies the host variables into which you will retrieve database field values. You must preface each host variable with a colon (:). The colon is an SQL convention that indicates the following variable is not a database field. You cannot use the *into-clause* in a **select** statement that appears inside a cursor declaration.

Example

The following **select** statement includes an *ordering-clause* with two sort keys:

```
exec sql
   declare urban_population_centers cursor for
      select city, state from cities
      order by state, population desc;
```

The following **select** statement includes an *into-clause* that specifies which database fields are put into which host variables:

```
exec sql
   select population, altitude, latitude,
         longitude
   into :pop, :alt, :lat, :long
   from cities
   where city = 'Boston';
```

This example assumes you declared the variables POP, ALT, LAT, and LONG to correspond to the database fields

POPULATION, ALTITUDE, LATITUDE, and LONGITUDE from the CITIES table.

The following cursor declaration joins records from two t-ables:

```
exec sql
   declare city_state cursor for
      select c.city, s.state_name, c.altitude,
            c.population
      from cities c, states s where
            c.state = s.state
      order by s.state_name, c.city;
```

The following cursor declaration retrieves the union of two tables:

```
exec sql
   declare all_cities cursor for
      select distinct city, state from cities
      union
      select distinct city, state from ski_areas
      union
      select distinct capital, state from states
      order by 2, 1;
```

The following example retrieves a record from STATES using STATE, a field with unique values:

```
exec sql
   select state_name, capital
      into :statename, :capital
      from states
      where state = :st;
```

The following example declares a cursor for all items that meet the specified criteria:

```
exec sql
   declare middle_america cursor for
      select city, state, population from cities
      where latitude_degrees between 33 and 42
         and longitude_degrees between 79 and 104;
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.

- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.
- SQLCODE of 100 indicates that no qualifying records were found.
- SQLCODE = -1 indicates a **select** statement has returned more than one record.

See Also

See the entries in this chapter for:

- **open**
- **fetch**
- **close**
- **whenever**

See also the entry in Chapter 5 for select expression.

Update

Function

The **update** statement changes the values of one or more fields in a record in a table or in the active set of a cursor.

If you do not provide a search condition (where...), InterBase updates all records in *table-name*. Be careful with this option.

Do not update records whose selection expression includes a distinct clause or a group clause.

Syntax

```
update table-name
set assignment-commalist
[where predicate|where current of cursor-name]
assignment::= database-field=scalar-expression
```

Options

table-name
Specifies the table that contains the record you want to update.

assignment
Assigns the *scalar-expression* to *database-field*. This assignment statement belongs to SQL and not to the host language. Do not use a host language assignment or equality operator inside an SQL update statement.

If the field you are assigning is a date, you cannot handle the field directly with SQL. Instead, you must use date functions such as GDML's **gds_$encode_date** to convert your external date representation to a host variable in the InterBase date format (that is, an array of two 32-bit integers), and then use the SQL assignment to assign the value of the host variable to the database field.

where *predicate*
Selects the record to modify.

where current of *cursor-name*
Specifies that the current record of the active set is to be modified. If you use the **where current of** clause, InterBase updates only the record at which the cursor is pointing. This form of update must follow:

- Declaring the cursor with a declare cursor statement

- Opening the cursor with an **open** statement
- Retrieving a record from the active set of that cursor with a **fetch** statement

Examples

The following statement updates the POPULATION field of all records from CITIES that are located in New York:

```
exec sql update cities
   set population = population * 1.03
   where state = 'NY';
```

The following statement modifies the POPULATION field of all records in the CITIES table:

```
exec sql update cities
   set population = population * 1.03;
```

The following example declares a cursor, opens it, fetches a record, and then alters that record:

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

main()
{

char statecode[3;
char st[3];
char cityname[16];
int  multiplier, pop, new_pop;
print ("Enter state with population needing
adjustment: ");
gets (statecode);
printf ("Percent change (eg 5 => 5% increase; -5 =>
5% decrease): ");
scanf (%d, multiplier);
multiplier = multiplier + 100;

exec sql
   declare pop_mod cursor for
      select city, state, population from cities
      where state = :statecode
      for update of population;
exec sql
```

```
   open pop_mod;
exec sql
   fetch pop_mod into :cityname, :st, :pop;
printf (" ");
while (SQLCODE == 0)
{
   new_pop := trunc ((pop * multiplier) / 100);
   printf ("%s, %s, old population: %, new
population: %d\n", cityname, st, pop, new_pop);
   exec sql
      update cities
         set population = :new_pop
         where current of pop_mod;
   exec sql
      fetch pop_mod into :cityname, :st, :pop;
}

exec sql
   close pop_mod;
exec sql
   rollback release;
}
```

Troubleshooting

See the Appendix for a discussion of error handling. The following values may be returned to SQLCODE:

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.
- SQLCODE = 100 indicates that no qualifying records were found.

See Also

See the entries in this chapter for:

- **declare cursor**
- **open**
- **fetch**
- **select**
- whenever

See the entry in Chapter 5 for predicate.

Whenever

Function

The **whenever** statement tests the SQLCODE value returned with each execution of an SQL statement. If the listed condition occurs, the **whenever** statement performs the **goto** statement.

A **whenever** statement must precede any statements that might result in an error. This way, InterBase knows what action to take in case of error.

Syntax

```
whenever {not found | sqlerror | sqlwarning}
goto-statement
```

Options

not found
Indicates the end of the input stream. This condition corresponds to the SQLCODE value of 100. This option is useful when you are looping through the active set of a cursor.

sqlerror
Indicates that the statement did not complete. This condition corresponds to a negative SQLCODE.

sqlwarning
Indicates a general system warning or informational message. This condition corresponds to SQLCODE values between 1 and 99, inclusive.

goto-statement

Example

The following example demonstrates the **sqlerror** option of the **whenever** statement:

```
database db = filename "bar.gdb";
exec sql begin declare section;
exec sql end declare section;
main ()
{
exec sql whenever sqlwarning
   go to warn;
exec sql whenever sqlerror
   go to error;
exec sql whenever not found
   go to no_data;
```

```
exec sql
    insert into rivers (river, source, length,
outflow)
   select "Assabet", state, 35, "Charles"
           from cities c where c.city = "Maynard";

exec sql
    commit release;
    exit ();

no_data:
    printf ("No record matched selection
criteria\n");
    exec sql
   rollback release;
    exit ();

warn:
error:
    printf ("Encountered SQLCODE %d\n", SQLCODE);
    printf ("    expanded error message -\n");
    gds_$print_status (gds_$status);
    exec sql
    rollback release;
   }
```

Troubleshooting

The following values may be returned to SQLCODE.

- SQLCODE < 0 indicates that the statement did not complete.
- SQLCODE = 0 indicates success.
- SQLCODE > 0 and < 100 indicates an informational message or warning.
- SQLCODE = 100 indicates the end of the input stream.

See Also

See the discussion of errors and error handling in the Appendix.

Appendix
Reporting and Handling Errors

This appendix discusses error reproting and handling in GDML and SQL programs. It also lists InterBase's major error codes and SQL error codes.

Overview

InterBase returns GDML error messages through the *status vector*, and returns SQL error messages through the SQLCODE. The following sections discuss error handling and list the error messages associated with GDML and SQL.

Reporting Errors to Programs

When you use **gpre** to preprocess programs, you may receive parsing errors. These are errors that **gpre** encounters when parsing a command, such as an unrecognized word, invalid syntax, and so on. The messages are generally self-explanatory.

When you run a program, InterBase returns the following types of errors:

- A database error. Database errors can be any one of many problems, such as conversion errors, arithmetic exceptions, and validation errors. If you encounter one of these messages, check any secondary messages.
- A bugcheck or internal error. Bugchecks reflect a problem that you should report. If you encounter a bugcheck, save the output and send it to InterBase at the following address for analysis:

  Borland International Inc.
  InterBase Customer Support
  P. O. Box 660001
  1800 Green Hills Road
  Scotts Valley, CA 85067

The InterBase access method returns error messages through the *status vector*, a list of twenty 32-bit integers. When InterBase writes to the status vector, it uses the first longword to pass the count (up to 19) of returned messages. Messages are divided into two classes:

- *Major codes* comprise a limited set of error codes that InterBase returns to the second longword slot of the status vector.

  For the sake of transportability to other DSRI-compatible systems, your program should test only for the major codes.
- *Minor codes* provide additional information about the problems identified by the major codes.

Figure A-1 represents a conceptual view of the status vector and what it might include:

Figure A-1. Status Vector

```
01 [count] 5
02 [major] bad_db_format
03 string pointer with name of database
04 [minor] object not a database
05 [minor] object not a file
```

The first longword returned by InterBase is the number of status vector slots. The second longword is the major code that describes the failure in its most general terms. Subsequent longwords provide additional information about the failure.

The GDML library includes a routine that lets you conveniently display the contents of the status vector in conjunction with the **on_error** clause you can use with all GDML statements. The format of this routine follows:

Syntax

```
gds_$print_status (gds_$status)
```

Gpre declares **gds_$status**, so you do not have to define it in your program.

Major Codes

This section lists the major code messages for the major codes. It contains a complete set of **gds** status codes. Many of the **gds** status codes occur in programs that use only SQL statements. Errors that do not occur in a purely SQL program are marked with a dagger (†).

Error	***gds_$arith_except***
Message	`arithmetic exception, numeric overflow, or string truncation`
Explanation	There was an error during the computation or data conversion of a numeric data item. For example, you changed the datatype of a field from short to long, but did not preprocess and compile the program again. As a result, you received an overflow error, or there may have been an overflow during the calculation of a computed field.
Action	Check for conflicts that could cause conversion errors or for string truncation. If necessary, preprocess and compile the program again.

Error	***gds_$bad_db_format***
Message	`file <filename> is not a valid database`
Explanation	You tried to open a file that is not a database.
Action	Check the file name of the database you want to access, correct any errors, and try again.

Error	***gds_$bad_db_handle*** †
Message	`invalid database handle (missing READY?)`
Explanation	If you are using a variable as an explicit database handle, you did not set the handle to zero or NIL before attaching the database, you modified the variable after its value was set by InterBase, or, if you used the **manual** option on **gpre**, you did not ready the database before using it.
Action	Set the database handle to zero or NIL before attaching the database, make sure that your program is not writing to this variable, or ready the database before using it.

Error ***gds_$bad_dbkey***†

Message `invalid database key`

Explanation You have probably modified the variable holding the database key (dbkey) value or used an uninitialized variable.

Action Check and correct your program logic.

Error ***gds_$bad_dpb_content***†

Message `bad parameters on attach or create database`

Explanation One or more of the items in the database parameter block (DPB) does not belong there. For example, a value may be out of range, or there may be conflicting items.

Action Check and correct the parameters.

Error ***gds_$bad_dpb_form***†

Message `unrecognized database parameter block`

Explanation One or more of the parameters in the database parameter block (dpb) is undefined.

Action Check and correct the parameters.

Error ***gds_$bad_req_handle***†

Message `invalid request handle`

Explanation You did not set the request handle to zero or NIL before compiling the request, you modified the variable after its value was set by InterBase, or, if you are using the call interface, you tried to start a request before you compiled it.

Action Set the handle to zero or NIL before compiling the request, make sure that your program does not write to this variable, or compile the request before starting it.

Error ***gds_$bad_segstr_handle***†

Message `invalid blob handle`

Explanation You did not set the blob handle to zero or NIL before opening the blob field, you modified the blob variable after it was set by Inter-

Base, or, if you are using the call interface, you tried to read or write the blob before opening it.

Action — Set the handle to zero or NIL before opening the blob field. Make sure that your program does not write to this variable, open the blob for read (**gds_$open_blob**), or write (**gds_$create_blob**) before accessing it.

Error — ***gds_$bad_segstr_id***†

Message — `invalid blob identifier`

Explanation — You referenced an invalid identifier for a blob field.

Action — Make sure that you correctly copied the blob id from the source relation and try again.

Error — ***gds_$bad_tpb_content***†

Message — `invalid parameter in transaction parameter block`

Explanation — One or more of the parameters in the transaction parameter block (TPB) is out of range, or tpb items conflict.

Action — Check and correct the tpb parameters.

Error — ***gds_$bad_tpb_form***†

Message — `invalid format for transaction parameter block`

Explanation — The format of the transaction parameter block (tpb) is incorrect.

Action — Be sure you are using the correct tpb version and the qualifiers are grouped correctly.

Error — ***gds_$bad_trans_handle***†

Message — `invalid transaction handle (missing START_TRANSACTION?)`

Explanation — You did not set the transaction handle to zero or NIL before starting the transaction, you modified the variable after it was set by InterBase, or if you used the **manual** option on **gpre**, you did not start the transaction before reading or writing a record.

Action	Set the handle to zero or NIL, make sure that your program is not writing to this variable, or start the transaction before you perform any data manipulation operations.
Error	***gds_$bug_check***
Message	`internal gds software consistency check (<value>)`
Explanation	An internal system failure occurred.
Action	Please report this error to Interbase Software Customer Support at 800-437-7367 or you can fax a copy of your program to 617-271-0221.
Error	***gds_$convert_error***
Message	`conversion error from string "<string>"`
Explanation	Your program attempted an illegal or unsupported data conversion. For example, you may have tried to convert an alphabetic string to a floating point number or a blob to any other datatype.
Action	Check your program for illegal or unsupported data conversions. If the problem is caused by bad data entry, provide some mechanism to handle invalid input.
Error	***gds_$db_corrupt***
Message	`database file appears corrupt (<filename>)`
Explanation	Data structures in the database have been corrupted.
Action	Read the discussion of **gfix** in the *Database Operations* guide and follow directions for dealing with a corrupted database.
Error	***gds_$deadlock***
Message	`deadlock`
Explanation	Your program cannot continue because it has encountered a deadlock with another process.
Action	Roll back your transaction and start a new one.

Error *gds_$excess_trans*†

Explanation Your program, while running with a foreign DBMS, attempted to start a second concurrent transaction and the foreign DBMS allows only one transaction per process.

Action Do not use more than one transaction per process with Rdb/VMS.

Error *gds_$fatal_conflict*

Message `unrecoverable conflict with limbo transaction <transaction id>`

Explanation Your transaction attempted to update a record whose most recent copy was created by a transaction in limbo. Until that transaction has completed, you cannot change the record.

Action Use the **gfix-two-phase-list** command to print the current state of the transactions and their partners. This command does an automatic commit or rollback.

Error *gds_$from_no_match*†

Message `no match for first value expression`

Explanation You used a request language **blr_from** value expression or a GDML first-value-expression for which there were no matches in the database.

Action In the request language, use a **blr_via** value expression instead of **blr_from**. **Blr_via** lets you supply a default value in case there is not a match for the primary value expression. If you are using GDML you should test for this condition.

Error *gds_$imp_exc*

Message `Implementation limit exceeded`

Explanation Your request exceeded some defined internal limit.

Action Simplify the request. Please report this error to Interbase Software Customer Support at 800-437-7367. You can send a copy of the original request to Interbase for analysis at:

Borland International
InterBase Customer Support

1800 Green Hills Road
P. O. Box 660001
Scotts Valley, CA 95067-0001 USA

You can also fax a copy of your original request to 408-439-7808.

Error ***gds_$infinap***†

Message `information type inappropriate for object specified`

Explanation A call to **gds** information routines referenced an item that does not exist. For example, you asked for the total length of a blob in a call to **gds_$database_info**.

Action Check and correct the information items.

Error ***gds_$infona***†

Message `no information of this type available for object specified`

Explanation A call to a **gds** information routine requested information that is not currently available for that object.

Action Check and correct the parameters. This error is sometimes appropriate depending on the status of the object at the time of the call. For example, an inactive request does not have a current message.

Error ***gds_$infunk*** †

Message `unknown information item`

Explanation A call to a **gds** information routine referenced an undefined item.

Action Check and correct the information items.

Error ***gds_$invalid_blr***

Message `invalid request blr at offset <integer>`

Explanation InterBase found an error in the binary language representation (**blr**) of a request at the position indicated by the offset.

Action

The offset returned in the message is the number of the offending byte in the **blr** string. Check and correct the syntax of the request language statement containing the unrecognized byte.

Note

Report this error to Interbase Software if you encounter it in an SQL program or a preprocessed GDML program.

Error *gds_$io_error*

Message `I/O error during "<operation>" operation for file "<filename>"`

Explanation Your program encountered an input or output error.

Action Check secondary messages for more information. The problem may be an obvious one, such as incorrect file name or a file protection problem. If that does not eliminate the problem, check your program logic. To avoid errors when the user enters a database name interactively, add an error handler to the statement that causes this message to appear.

Error *gds_$lock_conflict*

Message `lock conflict on no wait transaction`

Explanation A **start_transaction** statement specified the **nowait** option, and a resource needed by the program was not available.

Action The **nowait** option is generally not recommended. If your program requires it, include an error handler to trap for this error, and then wait and retry the statement. Your program may have to loop while waiting for resources.

Error *gds_$metadata_corrupt*

Message `corrupt system relation`

Explanation The system relations for the named database have been corrupted.

Action Check any secondary messages. Correct the problem. If the system relations were not corrupted by something you did, copy the broken database and send the copy to the following address for analysis:

Borland International
InterBase Customer Support
1800 Green Hills Road
P. O. Box 660001
Scotts Valley, CA 95067-0001 USA

Error ***gds_$no_cur_rec***

Message `no current record for fetch operation`

Explanation Your program executed a fetch before opening the associated stream or cursor, or after reaching the end of stream. This error does not occur through the call interface.

Action Open a stream before fetching records from it. Do not fetch records after the stream has been exhausted. You may want to include an error handler to trap for the end of stream.

Error ***gds_$no_dup***

Message `attempt to store a duplicate value in a unique index`

Explanation A store or modify violated the uniqueness of the index.

Action Provide a non-duplicate value for the indexed field. If the problem is at the data entry level, you may want to add an rror handler to trap for this error and re-prompt for another value.

Error ***gds_$no_finish***†

Message `program attempted to exit without finishing database`

Explanation Your program exited without detaching the database.

Action Explicitly end database access with a GDML **finish** statement or a call to **gds_$detach_database**.

Error ***gds_$no_priv***

Message `no permission for <access-type> access to <entity>`

Explanation Your program attempted an operation on an object for which you do not have the appropriate access rights.

Action

Check the access control list (ACL) for the object you referenced. You may have inadvertently locked yourself out. However, if you do not own the database you are accessing, someone else may have prevented you from accessing that object. If you think that you should have access to that object, contact the person who created the ACL for that object.

For more information about securety schemes, see the chapter on securing data and metadata in the *Data Definition Guide.*

Error ***gds_$no_recon***†

Message `transaction is not in limbo`

Explanation Your attempt to reconnect a transaction to a database failed because the specified transaction is not in limbo.

Action Check that the transaction identifier is incorrect. If it is, correct it and try again.

Error ***gds_$no_record***†

Message `invalid database key`

Explanation Your program referenced an invalid database key (dbkey).

Action You used a **blr_dbkey** value expression in your program which references a non-record. You may have erased the record and then tried to access it by its dbkey, or you may have altered the dbkey. Correct your program.

Error ***gds_$no_segstr_close***†

Message `blob was not closed`

Explanation Your storage of a blob field did not terminate by closing the blob field.

Action Terminate the blob storage with a call to **gds_$close_blob** or GDML's **close_blob** statement so that InterBase can complete the field.

Error ***gds_$not_valid***

Message `validation error for field <field-name>, value "<string>"`

Explanation — A validation check failed on a store or modify.

Action — If the value you tried to write seems valid, you may want to update the validation clause for that field. Otherwise, include an error handler to re-prompt if the validation check fails.

Error — ***gds_$obsolete_metadata***†

Message — `metadata is obsolete`

Explanation — Your program probably referenced an object that does not exist in the database.

Action — Check object names or identifiers to make sure that they still exist in the database. Correct your program if they don't.

Error — ***gds_$open_trans***†

Message — `a transaction has not been terminated`

Explanation — Your program attempted to detach a database without committing or rolling back one or more transactions.

Action — Commit or roll back those transactions before you detach the database.

Error — ***gds_$port_len***

Message — `message length error (encountered <integer>, expected <integer>)`

Explanation — The actual length of a buffer does not correspond to what the request language says it should be.

Action — Make sure that the *blr_string_length* parameter on the call to **gds_$compile_request** matches the BLR string. If you receive this error while using GDML or SQL, please submit a bug report.

Error — ***gds_$random***

Explanation — An unexpected error occurred.

Action — Check secondary messages.

Error	***gds_$read_only_field***
Message	`attempted update of read-only field`
Explanation	Your program tried to change the value of a read-only field in a system relation, a computed field, or a field used in a view.
Action	If the read-only field is in a system relation, change your program. If the field is a computed field, you have to change the source fields to change its value. If the field takes part in a view, update it in its source relations.

Error	***gds_$read_only_rel***
Message	`attempted update of read-only relation`
Explanation	Your program tried to update a relation that it earlier reserved for read access.
Action	If you want to write to the relation, reserve it for write.

Error	***gds_$read_only_trans***†
Message	`attempted update during read-only transaction`
Explanation	Your program tried to update during a read-only translation.
Action	If you want to update the database, use a read-write transaction.

Error	***gds_$req_no_trans***
Message	`no transaction for request`
Explanation	Your program tried to continue a request after the enveloping transaction had been committed or rolled back.
Action	Check and correct your program logic. Commit or roll back the transaction only after you have completed all operations that you want in the transaction. In SQL programs, you must re-open cursors after you commit a transaction.

Error	***gds_$read_only_view***
Message	`can't update read only view <view-name>`
Explanation	Your program tried to update a view that contains a record select, join, or project operation.

Action	Views that include a record select, join, or project cannot be updated. If you want to perform updates, you must do so through the source relations. If you are updating join terms, make sure that you change them in all relations. In any case, update the source relations in a single transaction so that you make the changes consistently.
Error	***gds_$req_sync***
Message	`request synchronization error`
Explanation	Your program issued a send or receive for a message type that did not match the logic of the BLR request.
Action	For call interface programs, locate and correct the program error. If you received this error while using GDML or SQL. Please submit a bug report.
Error	***gds_$req_wrong_db***
Message	`request referenced an unavailable database`
Explanation	Your program referenced a relation from a database that is not available within the current transaction.
Action	Change your program so that the required database is within the scope of the transaction.
Error	***gds_$segment***†
Message	`segment buffer length shorter than expected`
Explanation	The length of the segment_buffer on a blob call was shorter than the segment returned by InterBase. Therefore, InterBase could return only part of the segment.
Action	Check the segment_buffer_length parameter on the blob calls and make sure that it is long enough for handling the segments of the blob field you are accessing. Alternately, you could trap for this error and accept truncated values.
Error	***gds_$segstr_eof***†
Message	`attempted retrieval of more segments than exist`

Explanation	Your program tried to retrieve more segments from a blob field than were stored.
Action	Change your program so that it tests for this condition and stops retrieving segments when there aren't any more.

Error	***gds_$segstr_no_op***
Message	`attempted invalid operation on a blob`
Explanation	Your program tried to do something that cannot be done with blob fields.
Action	Check your program to make sure that it does not reference a blob field in a Boolean expression or in a statement not intended for use with blobs. Both GDML and the call interface have statements or routines that perform blob storage, retrieval, and update.

Error	***gds_$segstr_no_reads***†
Message	`attempted read of a new, open blob`
Explanation	Your program tried to read from a blob field that it is creating.
Action	Check and correct your program logic. Close the blob field before you try to read from it.

Error	***gds_$segstr_no_trans***†
Message	`attempted action on blob outside of a transaction`
Explanation	Your program referenced a blob field after it committed or rolled back the transaction that had been processing the field.
Action	Change your program so that you perform whatever data manipulation is required in a transaction before you end that transaction.

Error	***gds_$segstr_no_write***
Message	`attempted write to read-only blob`
Explanation	Your program tried to write to a blob field that had been opened for read access.

Action	If you are using the call interface, open the blob by calling **gds_$create_blob**. If you are using GDML, open the blob with the **create_blob** statement.
Error	***gds_$segstr_wrong_db***
Message	`attempted reference to blob in unavailable database`
Explanation	Your program referenced a blob field from a relation in a database that is not available to the current transaction.
Action	Change your program so that the required database is available to the current transaction.
Error	***gds_$sys_request***
Message	`operating system directive failed`
Explanation	The operating system returned an error.
Action	Check secondary messages for more information. When you isolate the problem, you may want to include an error handler to trap for this condition.
Error	***gds_$unavailable***
Message	`unavailable database`
Explanation	Your program referenced a database that InterBase cannot access.
Action	Before you copy a database to another system, make sure that a version of InterBase or a compatible access method is available on that system.
Error	***gds_$unres_rel***†
Message	`relation <relation-name> was omitted from the transaction reserving list`
Explanation	Your program tried to access a relation that was not in the lock list for that transaction.
Action	Change your program so that all required relations are in the reserved list when you start a transaction with the reserving option.

Error	***gds_$uns_ext***
Message	`request includes an extension not supported in this implementation`
Explanation	Your program tried to do something that cannot be done with this version of InterBase.
Action	Avoid using that feature.

Error	***gds_$wish_list***
Message	`feature is not supported`
Explanation	You tried to do something that is not possible in this release of InterBase.
Action	Avoid using that feature.

Error	***gds_$wrong_ods***
Message	`unsupported on disk structure for file <filename>; found <structure-id>, support <structure-id>`
Explanation	The on-disk structure of the database you referenced does not match the structure expected by the version of InterBase that you are using.
Action	Check any secondary messages for both the actual and expected versions of the on-disk structure. The database may have been created using a field test version of InterBase that used a different on-disk structure than the production version. Use the old version of **gbak** to back up the database and then restore it using the current version of **gbak**.

Minor Codes

The minor codes supply additional information about the generic error indicated by the major codes. When the InterBase access method returns an error, read the explanation for the major code, do whatever is recommended as a user action, and then try again.

If the recommended action is to check secondary messages, read the text returned with the secondary message and correct the problem.

Preserving SQL Program Portability

If you intend to move your SQL programs between InterBase and other database management systems, in your error handling routines you should limit yourself to using the **whenever** statement and to checking the value of SQLCODE. The **whenever** statement lets you perform an action depending on SQLCODE's value.

For error handling in applications that you plan to use on InterBase or other DSRI database management systems, you can check both the SQLCODE message and the actual message returned by InterBase. There is a large overlap of SQLCODE errors, such that the code -901 signals any one of more than a dozen InterBase database errors.

SQLCODE Correspondence

The following list of SQLCODE errors equates the general SQL error with the InterBase error symbol discussed earlier in this appendix and provides a brief explanation. Errors that should not occur in a purely SQL program are marked with a dagger (†).

SQLCODE	GDS Error Symbol	Brief Error Message
-1	no corresponding msg	a singleton select returned too many records
-150	read_only_rel	read only relation
-151	read_only_field	read only field
-151	read_only_view	can't update read only view <view-name>
-413	convert_error	conversion error from string "<string>"
-501	bad_req_handle	invalid request handle
-508	no_cur_rec	no current record
-551	no_priv	no permission for <access-type> access to <entity>
-607	no_meta_update	no metadata update
-625	not_valid	validation error for field <field-name>, value "<string>"
-802	arith_except	arithmetic exception, numeric overflow, or string truncation
-803	no_dup	attempt to store a duplicate value in a unique index
-804	wronumarg	wrong number of arguments on call
-817	read_only_trans	attempted update during read-only transaction
-901	bad_dpb_content†	bad parameters on attach or create database
-901	bad_dpb_form†	unrecognized database parameter block

SQLCODE	GDS Error Symbol	Brief Error Message
-901	bad_dbkey†	bad database key
-901	bad_segstr_handle†	invalid blob handle
-901	bad_segstr_id†	bad segmented string id
-901	bad_tpb_content†	bad tpb content
-901	bad_tpb_form†	bad tpb format
-901	bad_trans_handle†	invalid transaction handle (missing START TRANSACTION?)
-901	excess_trans	too many transactions (Rdb access only)
-901	fatal_conflict	unrecoverable conflict with limbo transaction <transaction-id>
-901	from_no_match†	no matches in the database
-901	imp_exc	Implementation limit exceeded
-901	infinap†	information type inappropriate for object specified
-901	infona†	no information of this type available for object specified
-901	infunk†	information item unknown
-901	invalid_blr	invalid request blr at offset <integer>
-901	lock_conflict	lock conflict
-901	no_finish†	program exited without finish
-901	no_recon†	reconnect failed
-901	no_record†	invalid database key
-901	no_segstr_close†	blob field not closed
-901	open_trans†	attempt to detach without ending transaction
-901	port_len	message length error (encountered <integer>, expected <integer>)

SQLCODE	GDS Error Symbol	Brief Error Message
-901	req_no_trans	attempt to continue request after transaction ended
-901	req_wrong_db	wrong database referenced in request
-901	random	unexpected error
-901	req_sync	request synchronization error
-901	segment†	segment buffer was too short
-901	segstr_eof†	attempt to retrieve more segments than were there
-901	segstr_no_op	attempt to do something with a blob that you can't do
-901	segstr_no_read†	attempt to read from a blob being created
-901	segstr_no_trans†	attempt to reference a blob after transaction ended
-901	segstr_no_write†	attempt to write a blob field opened for read
-901	segstr_wrong_db†	wrong database for referenced blob
-901	unres_rel†	relation <relation-name> was omitted from the transaction reserving list
-901	uns_ext	request includes an extension not supported in this implementation
-901	wish_list	feature is not supported
-902	bug_check	internal gds software consistency check (<string>)
-902	db_corrupt	database file appears corrupt (<string>)
-902	io_error	I/O error during "<operation>" operation for file "<filename>"
-902	metadata_corrupt	metadata is corrupt
-902	obsolete_metadata	metadata is obsolete

SQLCODE	GDS Error Symbol	Brief Error Message
-902	sys_request	operating system directive failed
-902	wrong_ods	unsupported on disk structure for file <filename>; found <structure-id>, support <structure-id>
-904	bad_db_handle†	invalid database handle (missing READY?)
-904	unavailable	unavailable
-912	deadlock	deadlock
-922	bad_db_format	file <filename> is not a valid database

Dynamic SQL Error Codes

The following list of SQLCODEs can occur with the dynamic SQL statements discussed in the SQL section of the *Programmer's Guide*:

SQLCODE	Brief Error Message
100	No more records
103	Constant datatype unknown
104	Parsing error
204	Relation unknown
206	Field name unresolvable
303	Conversion error
504	Could not find cursor
510	Cursor not updatable
518	Could not find request
804	Counts of field and values don't match (INSERT)
804	Datatype not recognized
804	SQLDA does not exists
804	Wrong number of variables

As with many SQLCODEs, the codes for dynamic SQL can mean any of several things. The Troubleshooting sections for the dynamic SQL entries in the SQL section of the *Programmer's Reference* list which errors can occur.

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W