InterBase
Programmer's Guide

Disclaimer

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

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

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

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

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

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

PART III: Programming with SQL

13 Getting Started with SQL

14 Retrieving Data with SQL

Preface

This manual describes how to program with InterBase.

Who Should Read this Book

You should read this book if you are an applications programmer who uses an InterBase database. The book assumes programming knowledge, but does not assume specific knowledge of InterBase.

Using this Book

This book contains the following chapters and appendixes:

Chapter 1	Introduces you to the fundamentals of programming with InterBase.
Chapter 2	Contains information about InterBase transaction handling.
Chapter 3	Describes host language considerations.
Chapter 4	Introduces programming with GDML, the InterBase data manipulation language.
Chapter 5	Describes how to retrieve data from an InterBase database by using embedded GDML.
Chapter 6	Describes how to store, modify, and delete data by using embedded GDML.
Chapter 7	Discusses the array datatype, array concepts and how to use arrays with InterBase.
Chapter 8	Describes the blob datatype and how to program with blobs
Chapter 9	Describes blob filters and discusses how to use blob filters
Chapter 10	Describes how to program using date fields.
Chapter 11	Contains information on programming with events, a mechanism for detecting and reporting changes in a database.
Chapter 12	Describes how to use **gds** call interface routines.
Chapter 13	Introduces programming with embedded SQL.
Chapter 14	Describes how to retrieve data by using embedded SQL.
Chapter 15	Describes how to store, modify, and delete data by using embedded SQL.
Chapter 16	Describes how to define, modify, and delete metadata by using embedded SQL.
Chapter 17	Describes how to mix SQL with GDML statements and **gds** calls.
Chapter 18	Describes how to preprocess your program by using **gpre**.

Chapter 19	Describes how to compile and link your preprocessed program.
Appendix A	Shows a sample database definition.
Appendix B	Includes information about programming with Ada.

Text Conventions

This book uses the following text conventions.

boldface

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

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

italic

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

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

`fixed width font`

Indicates user-supplied values and example code:

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

UPPER CASE

Indicates relation names and field names:

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

Syntax Conventions

This book uses the following syntax conventions.

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

InterBase Documentation

The InterBase Version 3.0 documentation set contains the following books:

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

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

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

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

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

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

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

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

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

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

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

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

Part I
InterBase Programming Fundamentals

Chapter 1
Introduction

This chapter introduces the GDML and SQL relational data manipulation languages. It also introduces **gpre**, the InterBase preprocessor.

Overview

GDML, SQL, and **gpre** are described in this section.

GDML

GDML is a data manipulation language used to access relation databases maintained by InterBase and other DSRI-compatible database management systems. GDML provides full read-write access to user data and to the data dictionary, comprehensive control over its transaction environment, and full support of the basic large object (*blob*), date, and array datatypes.

InterBase supports two variants of GDML:

- Embedded GDML consists of GDML statements that you put in a host-language program. This variant of GDML is described in Part II of this book.
- Interactive GDML consists of GDML statements you use interactively through **qli**. This variant of GDML is described in the *Qli Guide*.

SQL

SQL is an industry-standard data manipulation language used to access relational databases. SQL provides full read-write access to both user data and the data dictionary, and also provides a transaction environment. You can enhance the power of an SQL program by intermixing GDML statements.

InterBase supports three variants of SQL:

- Embedded SQL consists of statements you put in a host-language program. This variant of SQL is described in Part III of this book.
- Dynamic SQL is a facility that lets you accept or generate SQL statements at runtime. This facility consists of statements you put in a host-language program. It is described in the *DSQL Programmer's Guide*.
- Interactive SQL consists of SQL statements you use interactively through **qli**. This variant of SQL is described in the *Qli Guide*.

The InterBase implementation of SQL matches the ANSI SQL standard.

Gpre

Gpre is a system utility that reads data manipulation statements and converts them into calls to InterBase. Once you embed GDML and SQL statements in a host-language program, you must preprocess that program using gpre.

Table 1-1 lists the languages and operating systems **gpre** supports. Supported langauges are marked with an x.

Table 1-1. Supported Languages

Operating System	Ada	BASIC	C	COBOL	FORTRAN	Pascal	PL/1
Apollo	x		x		x	x	
Sun	x		x		x	x	
Other UNIX			x				
VMS	x	x	x	x	x	x	x

Statements and Commands

Requests made to **gpre** fall into two categories:

- *Statements* retrieve and alter data, or affect the order of execution for other statements.

  For example, the GDML **for**, **store**, **modify** and **erase** statements and the SQL **select**, **update**, **insert** and **delete** statements retrieve and alter data. Statements that affect the order of execution include the interactive GDML **if/else**, **repeat** and **begin/end** statements.

- *Commands* affect the operating environment. Commands include the GDML **ready** and **finish** commands, and metadata change requests such as the SQL **alter table** command.

The distinction between commands and statements is important when you work with compound statements. Only statements can be used in a GDML **if/else** statement, between a GDML **begin-end** block, in a GDML **repeat** loop, or in the loop of a GDML **for** statement. A comand like the SQL **commit** command closes opened SQL cursors.

For More Information

For more information on:

- The InterBase transaction environment and host-language considerations, refer to the remainder of Part I, *InterBase Programming Fundamentals*.
- GDML statements, refer to Part II, *Programming with GDML*.
- SQL statements, refer to Part III, *Programming with SQL*.
- **Gpre**, refer to Part IV, *Preparing Your Program*.
- DSQL statements, refer to the *DSQL Programmers Guide*.

Chapter 2
InterBase Transaction Management

Overview

In any programming environment, a related set of changes should be made in their entirety or not at all. If your program terminates before it finishes performing the complete set of related changes to a database, you would want the database restored to the state it was in before the set of changes started.

InterBase provides this capability through transactions. A transaction is a bounded set of statements that:

- Succeed or fail as a group
- See no changes made by simultaneous transactions
- Maintain a constant image of metadata

Transaction Models

The goals of InterBase transaction management are to prevent other users from interfering with data being accessed in active transactions and, if possible, to optimize database performance.

To ensure accuracy in updates and provide consistent results until a transaction is complete, transactions should be locked from the effects of other transactions. Simultaneous requests should not collide and users should never see or operate on data that has not been committed.

To optimize database performance, deadlocks should be prevented and the time one transaction waits for another to complete should be minimized.

To provide users with a consistent view of a database and still optimize database performance, InterBase provides both concurrency and consistency modes for transaction management. The user selects the mode which provides the appropriate locks on a relation and can specify options to modify a designated mode.

Consistency Model

Consistency mode provides a lock on a relation whenever it is accessed for read or write. In the traditional consistency mode, all transactions must wait for others to complete before accessing the locked relation. In InterBase, other readers will never wait for access and the user will not read the version of the record that is being updated.

The consistency model is useful in environments where there are many implicit relationships between records and a high risk that silmultaneous updates might damage these relationships in a way that cannot be controlled by triggers or unique constraints.

The Concurrency Model

Concurrency mode, the default InterBase mode, provides read and write sharing at the record level. This enables multiple users simultaneous access to the same database relation. Because of the InterBase multi-generational approach to transaction management, a transaction always uses a stable view of the database, even though changes may be occurring. Concurrency provides complete consistency for read-only transactions without waiting for updates or risking deadlocks. It also provides high consistency for read-write transactions.

Using the multi-generational record structure, InterBase creates a new version of a record, when a transaction modifies or erases the record. It tags this record version with the identifier of the transaction that created it and chains together all versions of

a record to form the multi-generational record. Each transaction sees exactly the state of the database when it began, plus its own changes, because it reads the record version that belongs to its generation.

In concurrency mode, a transaction always reads a stable view of the database, even though changes may be occurring while the read takes place. If your transaction starts after all versions of the record are committed, InterBase delivers the newest version of the record. Otherwise, it looks backwards through the versions until it finds the right one.

Establishing the Transaction Environment with GDML

In GDML and SQL programs, the default transaction model is the concurrency model. When you use GDML, you can choose whether to use the default transaction environment, or to modify it.

Starting Transactions

There are two ways transactions are started in a GDML program:

- **Gpre** starts a transaction when it encounters the first GDML statement. The transaction **gpre** starts is called the default transaction. There is only one default transaction per process. It has the transaction handle **gds_$trans** (on systems where the compiler accepts dollar signs in identifiers) or **gds__trans** (on systems where the compiler doesn't accept dollar signs in identifiers).

  You can override the automatic starting of transactions by using the **manual** option when you preprocess your program. For information on using this option, refer to Chapter 18, *Preprocessing Your Program*.

- You can use a **start_transaction** command to explicitly start a transaction. When you use this statement, you can include a transaction handle. The transaction handle declares a name you can use when you have to reference multiple transactions in a program. Multiple transactions are discussed later in this chapter.

Using **start_transaction**, you can run a transaction in the default concurrency mode or specify consistency mode. In InterBase, a transaction is started for all the databases that you specify in your program.

The syntax you use in your program to start a transaction depends on the type of access you need for the specified transaction and how you want locking to be performed.You can use the **start_transaction** command to explicitly start a transaction and assign the appropriate locks with the following syntax:

```
start_transaction [<transaction handle>]
[concurrency | consistancy]
[read_write | read_only]
[wait | no_wait]
[reserving <relation_list> for [read | write] ];
```

start_ transaction-- The command used to start a transaction providing the following default characteristcs:

```
gds_$trans as the transaction handle
concurrency
read_write
wait
no reserving of relations
```

The default characteristics are reasonable choices for many applications; however, you may add one or more of the following options to this command:

transaction handle —The assigned name of the transaction. You can assign a unique name to a transaction. If no name is assigned, **gds_$trans** is used as the default. If you use more than one transaction in your program, you can use the default transaction name for one transaction but must assign unique names to additional transactions. The default transaction is automatically committed without naming it in a **commit** statement; however, all named transactions must be committed explicitly by name. In SQL you do not use transaction names, since multiple transactions are not permitted.

concurrency — The default transaction mode that provides relation sharing.

consistency — The optional transaction mode that provides relation locking.

read_write — Provides both read and write access to a relation. This is the default for both concurrency and consistency modes.

read_only — Provides read_only access to a relation. The user can select either consistency read_only or concurrency read_only options. Choosing either of these options rather than the default read_write access prevents write access to a relation by the locking process.

wait — Applies to lock conflicts. In consistency mode, an update to a transaction waits until a previous update completes and releases the lock on a relation. In concurrency mode, an update waits when you attempt to update a record that was updated by a simultaneous transaction. The wait option is the default for both consistency and concurrency modes and is the recommended transaction option.

no_wait — Applies to lock conflicts. In consistency mode, an error is returned when you try to update a relation that is already locked. In concurrency mode, an update returns an error immediately when you attempt to update a record that was updated by a concurrent transaction.

Note

The **no_wait** transaction option should be used only for applications that cannot wait. Caution is recommended, since deadlocks can result. If your application requires the no_wait option, code should be added to prevent most deadlocks.

reserving — Acquires a read or write lock on a specified relation or relation list. This option is used in conjunction with the transaction mode to prevent deadlocks and update conflicts by acquiring a lock on the reserved relation at the beginning of the transaction and holding the lock until the transaction completes.[1] If you reserve a relation, the specified lock on the reserved relation overrides the transaction mode.

You can reserve a specified relation for read or write. If **read** or **write** is not specified in the reserving clause, the lock on the reserved relation defaults to the transaction mode.

Any number of relations can be reserved within the list, each relation separated by a comma. If you elect to specify mixed-mode reserving in a reserving clause; that is, reserve some relations for read and others for write, you must separate the different modes by a comma. In the following example, the relations cities and states are reserved for read and the relation mayors is reserved for write:

```
start transaction consistency
reserving cities, states for read, mayors for write;
```

The reserving clause is also used to start a transaction for a specified database. To do so, you must specify the appropriate database handle associated with a relation. For example, to start a transaction against the employee database:

```
start transaction promote consistency
reserving emp.job_title for read;
```

Lock Compatibilities

InterBase locking enables you to acquire a lock on a relation as long as that relation has not been previously locked by a more stringent locking mode. For example, if a consistency mode transaction is updated in a relation, a concurrency mode transaction must wait until the update is complete before acquiring a lock on the relation and performing its update.

Locks on relations in InterBase can be upgraded, but can never be downgraded. For example, if you have a read lock on the relation and you decide to update the relation, the lock is updated to a write lock.

---

1. If you are running in concurrency mode, you do not get deadlock protection by reserving relations.

The table below indicates lock compatibilities:

yes — The locks are compatible. If two transactions attempt to access the same relation at the same time, both transactions are allowed access to the relation.

blank — The locks are incompatible. The first transaction to access the relation must complete before the lock is released and the second transaction is allowed access

Table 2-1. Lock Compatibilities for Transaction Modes

Transactions	Concurrency read_only	Concurrency read_write	Consistency read_only	Consistency read_write
Concurrency read_only	yes	yes	yes	yes
Concurrency read_write	yes	yes		
Consistency read_only	yes		yes	
Consistency read_write	yes			

Transaction Examples

The following examples using the employee database indicate the most effective and common uses of transaction management:

Table 2-2. Transactions EXamples

Syntax	Attribute	Lock compatibility
start_transaction	-concurrency -shared read_write -wait	You can read or write. Other concurrency transactions can read or write. Other consistency transactions can only read.

Table 2-2. Transactions EXamples

Syntax	Attribute	Lock compatibility
start_transaction sales consistency;	-consistency -protected read_write -wait	You can read or write. Other concurrency transactions can read. Other consistency transactions can only read.
start_transaction promotion consistency reserving emp.job for write;	-consistency -protected read_write on promotion -protected read_write on job -wait	You can read or write. Only you can write to job in employee database. Other concurrency transactions can read.
start_transaction add_emp consistency reserving job for read, salary for write;	-Consistency -protected read_write on add_emp -read_only on job -protected read_write on salary -wait	You can read add_emp; read and write salary. Other concurrency and consistency transactions can read job. Other concurrency transactions can read salary.

Accessing the Database

GDML supports multiple database access from a single transaction and multiple transactions per program.

In a GDML program, you can use:

- One transaction to access a single database
- Multiple transactions to access a single database
- One transaction to access multiple databases
- Multiple transactions to access multiple databases

Using One Transaction to Access a Single Database

The simplest type of transaction accesses one or more relations from a single database. The database can be located either on your local node or on a remote node.

For example, the transaction below reads the CITIES relation, asks whether you want to delete a record, and deletes the record if you say yes:

```
start_transaction;
for c in cities
  printf ("%s %s\n", c.city, c.state);
  printf ("Delete this one? ");
  gets (response);
  if (response == 'Y')
      erase c;
end_for;
commit;
```

This transaction uses all of the default values discussed earlier in this chapter.

Using Multiple Transactions to Access a Single Database

You may find that a single transaction is not adequate for your applications. For this reason, GDML lets you have more than one transaction in a program. When you use multiple transactions, you can group statements into autonomous units and commit or roll them back individually. You can update related records with the certainty that all changes are made simultaneously.

Whenever you use multiple transactions, you must use *transaction handles* to identify the transactions on which a statement should operate. A transaction handle is a host variable, initialized to zero or null, that you declare and use to name a transaction when you start it. When you commit or rollback the transaction, you can specify which transaction you want to terminate.

If you don't declare a transaction handle when you start a transaction, **gpre** starts the *default transaction*. All data manipulation takes place within the default transaction unless you:

1. Start another transaction.
2. Declare a handle for the transaction.
3. Execute GDML statements under the control of the named transaction.

When **gpre** encounters statements without a transaction handle, **gpre** executes those statements under the default transaction.

For example, when you prepare, commit, or roll back an unnamed transaction, only the default transaction is affected.

The following program starts two separate transactions, one to get a badge number and the other to store a new employee. The retrieval code is in a separate function that can rollback and restart if it encounters a deadlock:

```
database db = filename 'emp.gdb';

#include <stdio.h>
static based on badge_num.badge    get_badge();
main ()
/*
 *  simple routine to store one employee from the terminal
 */
{
ready;
start_transaction;
store e in employees
    e.badge = get_badge ();
    printf ("enter first name: ");
    gets (e.first_name);
    printf ("enter last name· ");
    gets (e.last_name);
    printf ("enter birth date: ");
    gets (e.birth_date.char[12]);
    printf ("enter supervisor id: ");
    gets (e.supervisor.char[3]);
    printf ("enter department code: ");
    gets (e.department);
end_store;
commit;
finish;
}

static based on badge_num.badge get_badge ()
/*
 *  get a badge number and update the next available
 *  number
 */
{
long  get_badge_tr;
based on badge_num.badgebadge;
```

```
get_badge_tr = 0;

start_transaction get_badge_tr read_write consistency
  reserving badge_num for protected write;
for (transaction_handle get_badge_tr)
  first 1 b in badge_num sorted by descending b.badge
     badge = b.badge;
     modify b using
        b.badge += 1;
     end_modify;
end_for;
commit get_badge_tr;
return badge;
}
```

Using One Transaction to Access Multiple Databases

A major feature of InterBase is its ability to access more than one database in a single transaction, whether the databases are on your node or elsewhere on the network.

Whenever you use multiple databases, you must declare a *database handle* for each database. The database handle lets you refer to databases individually. It also lets you specify references, if there are overlapping relation names in the two databases.

As you can see in the example below, the database handle is a parameter on the **ready** and **finish** commands and the **for** statement.

InterBase opens multiple transactions by default in a program that accesses multiple databases. If you want to restrict a transaction to a single database in a multi-database program, you must use the **reserving** clause. This clause is described earlier in this section.

Retrieving Information from Multiple Databases

The following program opens two databases and starts a transaction that reads records from both. This example joins relations across databases by creating a STATES-CITIES hierarchy.

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

database atlas = compiletime filename 'atlas.gdb';
database gazetteer = compiletime filename
'pariah::dua1:[gds.demo]atlas.gdb';
```

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

   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 %s %s\n", c.city, c.latitude,
c.longitude);
       end_for;
   end_for;

   commit;
   finish atlas;
   finish gazetteer;
}
```

Writing Information to Multiple Databases

InterBase also lets you write to multiple local and remote databases in a transaction, while guaranteeing the integrity of all involved databases.

For example, suppose you update information in both the atlas and gazetteer databases used in the preceding example. Because the databases are on different computers in a local area network, there is always the chance the write operation on the remote node could fail. If this happens, you want the local node's write operations to roll back so that the databases are consistent.

InterBase automatically uses a *two-phase commit* when a transaction involves multiple databases:

- The first phase of the commit polls all participating nodes to see if anything stands in the way of actually committing the transaction. For example, there may have been a network failure that has disrupted inter-node communication.

  If any node reports a failure or fails to report success during the first phase, all other subtransactions roll back.
- When all participating nodes check in, the copy of InterBase that is coordinating the transaction and its subtransactions on remote nodes initiates the second phase of the two-phase commit. This phase makes the changes to the database permanent. The final commit is guaranteed to be successful, as long as the disk is still recognizable as a disk.

However, if a participating node reports back that it can't commit or fails to report at all, none of the participating nodes commits that transaction.

On very rare occasions, you may encounter a *limbo* situation. This can occur if there's a failure during the second phase of a two-phase commit, or between the first phase and the beginning of the second phase. This is a very rare situation, but it can happen if your communication lines go down during the two-phase commit process. A limbo transaction is neither committed nor rolled back.

Gfix automatically accesses this information and tells you whether to rollback or commit your remaining transactions. For information on using **gfix**, refer to the chapter on maintaining databases in *Database Operations*.

You also have the option of doing an explicit two-phase commit to coordinate database activity with non-database operations. For example, suppose you want to generate a paycheck by making some calculations in the database. Then you want to print the check. Before flagging the employee as paid and committing the transaction, you can check to see if the check printed. If it didn't, you can roll back the transaction and try again, once the printer is back on line.

You do an explicit two-phase commit by using the **prepare**, **commit**, and **rollback** commands. For information on these commands, refer to the GDML part of the *Programmer's Reference*.

Using Multiple Transactions to Access Multiple Databases

The process of using multiple transactions to access multiple databases is similar to using one transaction to access multiple databases.

The major difference between the processes is that you must keep track of your transactions as well as your databases when you use multiple transactions to access multiple databases.

For example, the following program uses two transactions. One accesses the emp1.gdb database; the other accesses the emp2.gdb database:

```
database db1 = filename "emp1.gdb";
database db2 = filename "emp2.gdb";

main ()
long  tr1, tr2;

tr1 = tr2 = 0;

ready;
```

```
start_transaction tr1 reserving db1.employees for read;

start_transaction tr2 reserving db2.badge_num for read;

for (transaction_handle tr2) first b in db2.badge_num sorted by
  descending b.badge
    for (transaction_handle tr1) e in db1.employees with
      e.badge > b.badge
      printf ("%s %s has a bad badge number\n", e.first_name,
          e.last_name)
   end_for;
end_for;

commit tr1, tr2;
finish;

}
```

Establishing the Transaction Environment in SQL

In accordance with the ANSI SQL standard, the transaction environment in standard InterBase SQL programs has the following characteristics:

- Transactions operate only in consistency mode.
- You can use only one transaction in your program. InterBase starts this default transaction for you automatically, when it encounters the first SQL statement.

You have the option of including GDML statements and commands in your SQL programs in order to gain more control over the transaction environment. For example, by mixing SQL and GDML statements and commands, you can start transactions explicitly and use all available GDML transaction options.

For information on using GDML statements and commands in an SQL program, refer to Chapter 17, *Mixing SQL with Other Interfaces*.

Error Recovery

You should always include error handling logic to deal with rolled back transactions in a graceful, predictable manner. A transaction should be able to deal with problems such as:

- A condition that triggers a rollback. For example, you may want to stop a transaction if there is an insufficient quantity of something on hand, a certain value shows up in data input, or a user hits an *undo* key.
- An error code from InterBase. For example, you may encounter a deadlock requiring the transaction to be rolled back. You can trap for specific errors by checking the contents of the status vector.

  The following **prepare** command includes an **on_error** clause that prints a message, rolls back a specific transaction, and then goes off to an error handling routine:

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

There are many other problems beyond the control of your program, like power failures and the loss of communication lines. For these reasons, InterBase provides backup and journaling utilities. These utilities are described in *Database Operations*.

More on the Concurrency Model

A detailed description of how the concurrency model handles concurrent transactions and how it cleans up old record versions is presented below. This is followed by a comparison of the concurrency model with the consistency model.

Handling Concurrent Transactions

Within the concurrency model, read operations are able to slide under concurrent write operations by reading older record versions. As an example of this approach to concurrency and consistency, consider Table 2-5 below. In this table:

- The *Time* column provides a relative chronology for two concurrent transactions.
- The *Program 1* and *Program 2* columns describe the activity of the two transactions.
- The *Data Value* column provides the value of the record in question.

Table 2-3. Concurrency Interactions

Time	Program 1	DataValue	Program 2
t0		version x: Smith's salary = $50,000	
t1	start transaction		
t2			start transaction
t3	update: Smith's salary = $75,000	version y: Smith's salary = $75,000	
t4			read: Smith's salary = $50,000
t5	commit transaction		
t6			commit transaction
t7			start transaction
t8			read: Smith's salary = $75,000

This table shows the interaction of two programs: Program 1 updates data, and Program 2 reads data. Before the transactions start, at time t0, Smith's salary is $50,000. Program 1 starts a transaction at point t1, and Program 2 starts a transaction at t2.

Both programs are restricted to a view of the database as of the time their transactions started.

At t3, Program 1 updates the value of Smith's salary record, such that he now makes $75,000. At t4, Program 2 gets around to reading Smith's record, and reads $50,000 for his salary. At t5, Program 1 commits, followed soon after by Program 2 's commit at t6. When Program 2 reads Smith's salary record again at t7, it sees he now makes $75,000.

Within the concurrency model, updates happen instantaneously, unless you try to write a record that's been changed while your transaction was active. However, concurrent write operations are more complicated than concurrent reads. A transaction is not allowed to change a record, if it can't see the most recent version. In this case, your program can read a record, but it gets a *deadlock* error if it tries to modify or erase it. Because the current state (or non-existence) of the record should influence your changes, InterBase denies the update request.

The correct response to a *deadlock* error is to roll back your transaction and restart it. Because your new transaction starts after the newest version of the record was committed, your changes are accepted.

Table 2-6 shows what happens when two transactions try to update the same record simultaneously. Program 1 wants to give Smith a $25,000 raise. Program 2 wants to give him a $5000 bonus:

Table 2-4. Conflict Mediation

Time	Program 1	Data Value	Program2
t0		version x: Smith's salary = $50,000	
t1	start transaction		
t2			start transaction
t3	update Smith's salary = $75,000		read: Smith's salary = $50,000
t4			write: Smith's salary = $55,000
t5			waiting for Program 1's lock
t6	commit transaction		

Table 2-4. Conflict Mediation

Time	Program 1	Data Value	Program2
t7			deadlock: update conflicts with concurrent update
t8			roll back transaction
t9		version y: Smith's salary = $75,000	read: Smith's salary = $75,000
t9			update: Smith's salary = $80,000
t10		version z: Smith's salary = $80,000	
t11			commit transaction

In Table 2-4, both Program 1 and Program 2 want to increase Smith's salary. In this example, Program 1 updates Smith's salary, as before. Program 2 also tries to update the Smith's salary, but it requests the update before Program 1 has committed it's changes.

Because it does not have access to the most recently written version of the salary record, Program 2 now receives a deadlock error and must roll back. Once Program 1 commits its changes, Program 2 can re-access and update the record.

Note

As a good programming practice, you should always incorporate the commit, rollback, restart sequence shown in steps t6 through t9 in a loop as part of your error recovery. Error recovery is discussed earlier in this chapter.

Cleaning Up Old Record Versions

To avoid filling a database with old versions of records, every transaction participates in cooperative clean-up. When a transaction accesses a record, it removes both versions too old to be useful and versions created by transactions that ended without committing.

Because of this automatic clean-up, databases don't fill up with old record versions that can't be used. For example, if you load a database, check the size of the database, and then modify all of the records, you'll notice that the size of the database significantly increases. This increase results from the creation of a record version for each record in the database. However, subsequent database activity, including read activity, automatically initiates the garbage collection process. The database does not shrink, but space is made available for new records or additional record versions.

If you ever need to force the collection of old record versions, you can use the **gfix** utility. This utility is described in *Database Operations*.

Comparing the Concurrency and Consistency Models

There are six classic problems a transaction management system must address:

- Lost updates
- Dirty reads
- Cursor instability
- Non-reproducible reads
- Phantom records
- Update side effects

A discussion of how the two transaction models handle these problems follows.

Lost Updates

Lost updates occur when an update is overwritten by a simultaneous transaction that is unaware of the last change. Consider the previous example. If Program 2 had ignored Program 1's change and given Smith a $5,000 bonus but lost his $25,000 raise, Smith would be the victim of a lost update.

Both the consistency model and the concurrency model handle this problem. Once a transaction has updated a record, no concurrent transaction can update that same record until the first transaction either commits its update or rolls it back:

- A consistency transaction prevents lost updates by locking relations that are updated.
- A concurrency transaction marks new versions with an identifier. Concurrent transactions recognize this identifier as an indication that they must wait for the record to be committed or rolled back.

Thus, a transaction can't lose its update because of concurrent transactions.

Dirty Reads

Dirty reads occur when the database system allows a transaction to see data that has been updated, but not yet committed, by another transaction. The data is referred to as *dirty* in this context, because the transaction that updated it could roll back its changes. Any transaction that saw the change may have made incorrect decisions based on data that never entered the database.

The consistency model solves this problem by locking updated relations with a lock that excludes consistency mode readers. This lock prevents any other transaction from reading the records. Unfortunately, this causes consistency-mode readers to wait for updates to finish, and reduces throughput.

The concurrency model solves this problem by using its multi-generational approach. Because updates create new versions of records, the older versions can always be made available to concurrent readers. Readers always see the record versions that are consistent with the state of the database as it was at the start of their transaction. They never have to wait for updates to commit or roll back.

Non-Reproducible Reads

Non-reproducible reads occur when a transaction reads the same records repeatedly and gets different values for some or all fields. This can occur if a transaction is allowed to update or erase records that have been read by an active transaction. When the updating transaction commits, its changes are visible to the reading transaction.

For example, suppose your transaction calculates the total salary of employees in branch 12 and prints the result. Then your transaction prints the employee name and salary of each employee in branch 12. If a concurrent transaction changes the salary of any employee in branch 12 after the total salary is computed, the report will be incorrect.

To guard against this phenomenon, the consistency model locks relations that are read to prevent concurrent transactions from updating them. This makes writers wait for readers to finish their work.

The concurrency model uses its multi-generational approach to guarantee reproducible reads within a transaction. As in the case of dirty reads, concurrent transactions can always see the version of records that is consistent with their view of the database.

Phantom Records

The problem of phantom records is similar to the problem of non-reproducible reads. In fact, it's a form of non-reproducible reads which manifests itself when locking is used.

The difference between phantoms and non-reproducible reads is that phantoms are new records that the active reader would have read.

Consider the previous situation once again. Your transaction computes the total salary of employees in branch 12 and prints it. A concurrent transaction then inserts a new employee in branch 12. Your transaction then proceeds to print the employees in branch 12, and prints the new employee.

Once again, the report is wrong. Even if the system locked records that your transaction had read, this problem would not have been prevented. It could not have locked the phantom record, since it didn't exist at that time.

The consistency model solves this problem by locking the entire relation to prevent concurrent inserts. This reduces concurrency dramatically.

The multi-generational aproach of InterBase's concurrency model solves the phantom problem without locking. The system simply skips over any record versions that a transaction shouldn't see. In this case, it skips over phantom records altogether.

Update Side Effects

There is one final class of problems, which, in practice, is very rare. With all the protection offered by concurrency mode, it is still possible to have concurrent transactions execute in interleaved fashion and produce results that are not equivalent to a serial execution of the same transactions. Update side effects occur when record values are interdependent and their dependencies are not protected with triggers or unique constraints.

For example, suppose your application assigns employee id numbers by looking up the highest number currently stored in the database and incrementing it by 1. Now consider the case where two transactions, X and Y, each go off to store a new employee. The highest employee number when they both start is 32910. Transaction X reads the maximum value and adds 1. Transaction Y does the same. Assuming that duplicate values are allowed, the database now has two different employees with the same number, 32911. There is no way that two transactions run in a series could issue duplicate badge numbers.

Another case where updates have side effects occurs when transactions count the number of records in a table and then store the count. For example, if in the previous case, the program derived new badge numbers by counting the current employees and adding one to the count, duplicate badge numbers could still be assigned.

Table 2-7 shows how this can happen:

Table 2-5. Update Side Effects

Time	Badges	Program 1	Program2
t0	1, 2, 3, 4, 5	count = 5 employees	count = 5 employees
t1	1, 2, 3, 4, 5	new badge = 6	new badge = 6
t2	1, 2, 3, 4, 5	store	store
t3	1, 2, 3, 4, 5, 6, 6		

If you have to count records in a relation before storing a new one, you should either:

- Use metadata to avoid the problem. For example, you can define a unique index for the relation. By including the source field in the index, InterBase automatically disallows duplicate values. In this case, one transaction succeeds in storing the record, while the other transaction receives an error and doesn't store the record.

  For more information on defining a unique index, refer to the chapter on defining views and indexes in the *Data Definition Guide*.
- Define a relation that holds the next available badge number. This approach is also faster than the approach shown above.
- Use consistency mode in these limited cases.

A third example of a non-serializable transaction is the exchange of values situation. An exchange of values occurs when two transactions change values in two records. Transaction X reads a field in R1 and uses that value to update a field in R2. Meanwhile, transaction Y reads the same field in R2, and stores that value in the same field in R1.

For example, consider the classic problem of sexual discrimination in salary practices. A company discovers that it is paying its female employees fifty-nine cents for every dollar it pays its male employees. The company sets up a task force to study the problem. Two data processing subcommittees go off to solve the problem:

- One subcommittee sees that women make less than men, so they increase the salary of all women to match that of men.
- The other subcommittee looks at the company's bottom line and decides to bring the salary of men in line with women.

The net result is that men now make fifty-nine cents for every dollar made by women.

One solution to the exchange problem is to define another relation that holds the contentious salary data. Another solution is to use a consistency mode transaction.

For More Information

For information about transaction commands and error-handling clauses, refer to these entries in the GDML section of the *Programmer's Reference*:

- **commit**
- **finish**
- **on_error**
- **prepare**
- **rollback**
- **start_transaction**
- **transaction_handle**

For information about using GDML statements and commands in an SQL program, refer to Chapter 17, *Mixing SQL with Other Interfaces.*

Chapter 3
Host Language Considerations

This chapter discusses the special considerations involved with using GDML and SQL with supported host languages.

Overview

InterBase supports the following host languages:

- Ada
- BASIC
- C
- COBOL
- FORTRAN
- Pascal
- PL/1

Ada Considerations

This section contains usage information that pertains to all supported implementations of Ada. Considerations for both GDML and SQL and for SQL only are described below.

Considerations for Both GDML and SQL

The following considerations apply to using either GDML or SQL in Ada programs:

- Ada names can't contain a dollar sign ($). Therefore, for generic InterBase names shown in this document, substitute a name without the dollar sign. For example, **gds_$trans** becomes **gds_trans**.

  Because of the dollar sign prohibition, you must be very careful if you want to mix modules written in Ada with modules written in other languages. For the non-Ada modules, edit the output of **gpre** or use your own handles. If one of the non-Ada modules uses the status vector, make sure that you either edit the module to remove the dollar sign from the status vector name or reference the status vector only within that module.

- Preprocessed modules use two Ada packages:

  — You must compile and link the package *interbase* to your Ada library before you can compile any preprocessed Ada modules.

  — If you want to use the InterBase examples directory, you must also use the file *basic_io*. This file must be compiled and linked to your Ada library before the file *interbase*.

  To use these files, you must include them in your source programs.

  The files *interbase* and *basic_io* use vendor-specific extensions:

  — On Apollo systems, the extension for VERDIX Ada files is *.a*. The file *interbase.a* resides in the InterBase include directory, */interbase/include*. The file *basic_io.a* resides in the InterBase examples directory, */interbase/examples*.

  The extension for Alsys Ada files is *.ada*. The file *interbase.ada* resides in the InterBase include directory, */interbase/include*. The file *basic_io.ada* resides in the InterBase examples directory, */interbase/examples*.

  — On UNIX systems, the extension for Telesoft Ada is *.ada*. The file *interbase.ada* resides in the InterBase include directory, */usr/interbase/include*. The file *basic_io.ada* resides in the InterBase examples directory, */usr/interbase/examples*.

— On VMS systems, the extension for VMS Ada files is *.ada*. The file *interbase.ada* resides in the InterBase include directory, *sys$library*. The file *basic_io.ada* resides in the InterBase examples directory, *inerbase$ivp*.

Considerations for SQL Only

The following considerations apply to using SQL in an Ada program:

- Terminate SQL statements with a semicolon.
- Include the SQL communications area (SQLCA) where you declare module-wide data.
- Use double quotes (") instead of single quotes (') for quoted strings that are more than one character long.

BASIC Considerations

This section contains usage information that pertains to BASIC. Considerations for both GDML only and SQL only are described below.

Considerations for GDML Only

When you use GDML in a BASIC program, the GDML **database** declaration must follow a "labeled" BASIC line. It must precede any executable statements and **based_on** variable declarations.

Considerations for SQL Only

The following considerations apply to using SQL in a BASIC program:

- A "labeled" BASIC line must precede the inclusion of the SQLCA. Include the SQLCA after each subprogram or function declaration.
- Like BASIC statements, SQL statements that cross line boundaries must use an ampersand (&) to indicate the continuation. For example:

```
exec sql fetch cursor c &
   into :host1, :host2, :host3
```

C Considerations

This section contains usage information that pertains to C. Considerations for both GDML and SQL, for GDML only, and for SQL only are described below.

Considerations for Both GDML and SQL

The following considerations apply to using either GDML or SQL in a C program:

- **Gpre** provides the **either_case** option for use with C programs. If you mix cases in your C programs, make sure that you preprocess them with the **either_case** option. Otherwise, **gpre** prefixes a header to your input file and ignores all lowercase GDML and SQL characters.
- Terminate GDML and SQL statements with a semicolon.
- One GDML or SQL statement translates into one host language statement, although the host language statements are often compound statements.
- The relational operators in a record selection expression can match the operators used in native language comparisions.

  For example, the following is a legitimate GDML **for** loop:

```
for c in cities with c.state != "NY"
   printf ("%s, %s is not in New York\n", c.city, c.state);
end_for;
```

  The following is a legitimate SQL cursor declaration:

```
exec sql
   declare non_tulips cursor for
      select * from plants where species != "TULIP";
```

- Fields declared to be **char** are returned as null-terminated strings, unless the field has a subtype of **fixed** or the program was preprocessed with the **string** option.
- By default, **gpre** generates debug line numbers so you can debug a program without looking at the expanded intermediate C program. You can suppress the line numbers with the **no_lines** option.
- C does not have native string handling. The GDML library provides several string conversion routines that are useful only in C programs. These routines are:
  - **gds_$ftof** to move a fixed-length string to a fixed-length field
  - **gds_$vtof** to move a null-terminated string to a fixed-length field
  - **gds_$vtov** to move a null-terminated string to a variable length field

The syntax of the routines follows:

```
gds_$ftof (string, length1, field, length2)
gds_$vtof (string, field, length)
gds_$vtov (string, field, length)
```

These routines differ from other C library routines in that they enforce the length of the string. Therefore, if you copy a string that is too long, the string is truncated.

For character and varying string fields, **gpre** allocates one byte more than the length specified in the database allocation. This extra byte allows for a terminating null. By default, character fields are null-terminated in C programs, but the null is not stored in the database.

If the field will contain binary data, you should either specify a subtype of **fixed** when you define the field with **gdef** or use the **string** option when you preprocess the program. For information about defining fields, refer to the chapter on defining fields in the *Data Definition Guide*. For information about preprocessing a program, refer to Chapter 18, *Preprocessing Your Program*.

Considerations for GDML Only

The following considerations apply when you ise GDML in a C program:

- Place the **database** declaration before any executable statements and **based_on** variable declarations.
- On HP machines, you can't use a dollar sign ($) in **gds** calls. Use a double underscore instead. For example, to use the **gds_$ftof** routine, specify **gds_ _$ftof**.

Considerations for SQL Only

When you use SQL in a C program, include the SQLCA where you declare module-wide data.

COBOL Considerations

This section contains usage information that pertains to COBOL. Considerations for both GDML and SQL, for GDML only, and for SQL only are presented below.

Considerations for Both GDML and SQL

Gdef, the data definition language compiler, lets you specify a scale factor on the binary datatypes for use with COBOL. The scale factor is the degree of precision with which InterBase stores data.

For detailed information on the scale factor, refer to the chapter on defining fields in the *Data Definition Guide*.

Considerations for GDML Only

The following considerations apply when you use GDML in a COBOL program:

- Place the **database** declaration in the working storage section of the program.
- Don't use a period (.) inside a **for** loop or a **store**, **end store** loop.

Considerations for SQL Only

The following considerations apply when you use SQL in a COBOL program:

- Include the SQLCA either in the working storage section or the declare section of the program.
- End all SQL statements with the words **end-exec**.
- Don't use a period as a terminator inside an **exec sql**, **end-exec** block.

FORTRAN Considerations

This section contains usage information that pertains to FORTRAN. Considerations for both GDML and SQL, for GDML only, and for SQL only are presented below.

Considerations for Both GDML and SQL

The following considerations apply to using either GDML or SQL in a FORTRAN program:

- Use **block if** statements (if-then-end if) rather than relying on a one-to-one translation of GDML or SQL statements to host language statements. In FORTRAN, one data manipulation statement may translate into several FORTRAN statements.

  For example, the following GDML construct generates code that starts a transaction named MY_TRANS, if it has not already been started:

```
 IF (MY_TRANS .EQ. 0) THEN
+    start_transaction MY_TRANS
 END IF
```

  This GDML construct generates incorrect FORTRAN:

```
IF (MY_TRANS .EQ. 0) start_transaction MY_TRANS
```

  The following SQL construct generates code that rolls back a transaction, if the specified SQLCODE value shows up:

```
IF (SQLCODE .EQ. -16) THEN
   EXEC SQL ROLLBACK
END IF
```

  This SQL construct generates incorrect FORTRAN:

```
IF (SQLCODE .EQ. -16) EXEC SQL ROLLBACK
```

- Use GDML or SQL operators rather than FORTRAN operators in record selection expressions.

  You can't use the FORTRAN operators .EQ., .LE., and so on. **Gpre** accepts the Boolean operators | |, &&, AND, OR, but not .AND. or .OR. **Gpre** accepts EQ and NE (as well as LT, LE, and so on), but not the "dotted" form used in FORTRAN.

- You must treat a space as a token separator. Spaces are not significant in FORTRAN, but they are in both GDML and SQL. You can use multiple spaces where one would do.

For example, the following GDML commands are correct:

```
START_TRANSACTION MY_TRANS
start_transaction MY_TRANS
```

These GDML commands are incorrect:

```
S T A R T _ T R A N S A C T I O N M Y _ T R A N S
start_transactionMY_TRANS
```

The following SQL command is correct:

```
EXEC SQL OPEN BIG_CITIES
```

These SQL commands are incorrect:

```
E X E C  S Q L  O P E N  B I G _ C I T I E S
EXECSQLOPENBIG_CITIES
```

Considerations for GDML Only

The following considerations apply to using GDML in a FORTRAN program:

- Place the **database** declaration between the data declarations and the GDML data statements. The **database** declaration must precede any executable statements, but can follow the **based_on** declaration.
- Be sensitive to the position of carriage returns. In FORTRAN, GDML statements terminate with a carriage return. However, unlike FORTRAN statements, they do not require continuation lines when a statement crosses line boundaries.

  The absence of explicit terminators affects several simple statements. For example, the following **ready** commands are correct:

  ```
  ready
  ready DB
  ready 'fleurs.gdb' AS DB,
     'veggies.gdb' AS DB2
  ```

  Gpre does *not* preprocess the following statement correctly:

  ```
  ready
     DB
  ```

- Place matching parentheses around database field expressions that are followed immediately by a FORTRAN operator. Or, reverse the order of the expression so the database field expression comes after the FORTRAN operator.

For example, **gpre** correctly parses the following expression:

```
(S.STATE) .EQ. 'MA'

'MA' .EQ. S.STATE
```

Gpre does not process this expression correctly:

```
S.STATE .EQ. 'MA'
```

- By default, VAX FORTRAN passes character data by descriptor instead of by reference. Because InterBase is implemented in C, all **gds** calls and blob function calls expect character data to be passed by reference. Therefore, you must override the default behavior of the FORTRAN compiler when you pass character data to one of these functions. Use the FORTRAN function %REF to perform this override.

  For example, to override the default behavior of the FORTRAN compiler when you use a literal filename, type:

```
CALL BLOB_$LOAD(blob-field, db-handle, tx-handle,
      %REF('filename'), LEN('filename'))
```

  To override the default behavior of the FORTRAN compiler when you use a variable filename, type:

```
CALL BLOB_$LOAD(blob-field, db-handle, tx-handle,
      %REF(filename), LEN(filename))
```

Considerations for SQL Only

The following considerations apply to using SQL in a FORTRAN program:

- Include the SQLCA after data declarations and before SQL data statements at the head of each subroutine and function. You must include the SQLCA in every routine that contains database commands or references SQLCODE.
- Like FORTRAN, SQL requires continuation lines when a statement crosses line boundaries. For example:

```
 EXEC SQL FETCH CURSOR C
+  INTO :HOST1, :HOST2, :HOST3
```

- A **whenever** statement is altered either by another **whenever** statement or by the end of the routine.

Pascal Considerations

This section contains usage information that pertains to Pascal. Considerations for both GDML and SQL, for GDML only, and for SQL only are presented below.

Considerations for Both GDML and SQL

The following considerations apply to using either GDML or SQL in a Pascal program:

- Terminate all GDML and SQL statements with a semicolon, as you would terminate the host language statements.
- A single GDML or SQL statement translates into a single host language statement, although host language statements are often compound statements.
- The relational operators in a record selection expression can match the operators used in native language comparisions.

  For example, this is a correct GDML **for** loop:

```
for p in plants with p.species <> 'TULIPA'
   writeln (p.common_name, ' is not a tulip');
end_for;
```

  This is acorrect SQL cursor declaration:

```
exec sql declare non_tulips cursor for
   select plants with species <> 'TULIPA';
```

- The GDML library provides several string conversion routines that may be useful with some Pascal compilers. For example, your Pascal compiler may not support assignment between character strings of different lengths.

  These routines are:

  — **gds_$ftof** to move a fixed-length string to a fixed-length field

  — **gds_$vtof** to move a null-terminated string to a fixed-length field

  — **gds_$vtov** to move a null-terminated string to a variable-length field

  The syntax of the routines follows:

```
gds_$ftof (string, length1, field, length2)
gds_$vtof (string, field, length)
gds_$vtov (string, field, length)
```

  For a VAX Pascal program, use the Pascal function %REF (described below) to pass the strings by reference. For example:

```
gds_$ftof (%REF filename, %REF length (filename),
   field, %REF length (field))
```

Considerations for GDML Only

The following considerations apply to using GDML in a Pascal program:

- The **database** declaration should follow the Pascal **program** statement. It must precede any executable statements and **based_on** variable declarations.
- By default, VAX Pascal passes character data by descriptor, instead of by reference. Because InterBase is implemented in C, all **gds** calls and blob function calls expect character data to be passed by reference. Therefore, you must override the default behavior of the Pascal compiler when you pass character data to one of these functions. Use the Pascal function %REF to perform this override.

  For example, to override the default behavior of the Pascal compiler when you use a literal for the filename, type:

```
blob_$load (blob-field, db-handle, tx-handle,
      %REF 'filename', %REF length ('filename'))
```

  To override the default behavior of the Pascal compiler when you use a variable for the filename, type:

```
blob_$load (blob-field, db-handle, tx-handle,
      %REF filename, %REF length (filename))
```

Considerations for SQL Only

When you use SQL in a Pascal program, include the SQLCA after the Pascal **program** statement and before the first statement in the module or routine that does database access. This usually is in the area where module-wide variables are declared.

PL/1 Considerations

This section contains usage information that pertains to PL/1. Considerations for both GDML and SQL, for GDML only, and for SQL only are presented below.

Considerations for Both GDML and SQL

The following considerations apply to using either GDML or SQL in a PL/1 program:

- Terminate all GDML and SQL statements with a semicolon, as you would terminate the host language statements.
- A single GDML or SQL statement translates into a single host language statement, although host language statements are often compound statements.
- The relational operators in a record selection expression can match the operators used in native language comparisions.

  For example, this is a correct GDML **for** loop:

```
for p in plants with p.species ~= 'TULIPA'
    put list (p.common_name, ' is not a tulip')
skip;
end_for;
```

  This is a correct SQL cursor declaration:

```
exec sql declare non_tulips cursor for
   select plants with species ~= 'TULIPA';
```

- **Gdef**, the data definition language compiler, lets you specify a scale factor on the binary datatypes for use with PL/1. The scale factor is the degree of precision with which InterBase stores data.

  For detailed information on the scale factor, refer to the chapter on defining fields in the *Data Definition Guide*.

Considerations for GDML Only

When you use GDML in a PL/1 program, the **database** declaration should follow the PL/1 procedure. It must precede any executable statements and **based_on** variable declarations.

Considerations for SQL Only

When you use SQL in a PL/1 program, include the SQLCA after the PL/1 procedure and before the first statement in the module or routine that does database acess. This is ordinarily in the area where module-wide variables are declared.

For More Information

For more information on host-language specific preprocessing options, refer to Chapter 18, *Preprocessing Your Program*.

Part II
Programming with GDML

Chapter 4
Getting Started with GDML

This chapter describes how to program with GDML, the InterBase data manipulation language.

Overview

You can freely intermix GDML with host language statements. However, language compilers do not support GDML. Therefore, when you write a program that mixes GDML statements with host language statements, you must preprocess the program with **gpre** before compiling it.

A complete GDML program does the following:

1. Optionally declares a database.
2. Opens a database.
3. Declares local variables if necessary.
4. Starts a transaction.

5. Reads or writes data according to embedded GDML statements.
6. Handles errors.
7. Saves or rolls back changes and ends the transaction.
8. Closes any databases opened within the program.

Once you have a program that meets these requirements, you preprocess the program with **gpre**.

Actually, you can let **gpre** handle the opening of the database, and the start and finish of the transaction. Or, you can explicitly open databases, and start and end transactions. However, you must declare which database you want to access, and tell InterBase what you want to do with the data in the database. Additionally, you must always provide your own error handling.

The following sections discuss each of the steps required for a complete GDML program.

Declaring a Database

The first step in a program that includes GDML statements is to make a **database** declaration. For example:

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

The database declaration identifies the database file atlas.gdb as the source of database metadata for **gpre**. You can also use more than one database in a program. However, you must declare every database you use.

Note

When you declare multiple databases, the **start_transaction** statement tries to use all the declared databases. If you do not ready each database you declare, **gpre** gives you an error, unless you use a **reserving** clause that restricts the databases used. For information on the **reserving** clause, see Chapter 2, *InterBase Transaction Management.*

The declaration also identifies atlas as the *database handle* of the database file atlas.gdb. The database handle is a variable that lets you refer to individual databases during a multi-database transaction. For example, when you access several databases in a transaction, you can use the database handle to qualify relation names. This is necessary if you use the same relation name in more than one database.

You can also use the handle to identify which database to close when you finish using one. You can release the system resources associated with the database you no longer need, and continue processing the other databases you still need to access. For example, the following **finish** statement closes the atlas.gdb database:

```
finish atlas;
```

Opening a Database

To open a database file for access, include a **ready** statement in your program. If you don't supply a **ready** statement, **gpre** generates one when it encounters a GDML statement that requires it. The following statement opens the atlas. gdb database with the **database** statement:

```
ready atlas;
```

When you run the program, InterBase performs several tasks upon finding a **ready** statement. Specifically, it:

- Determines whether the database is on your node or on a remote node
- Checks to see if you specified a valid database. If so, it opens the database for access.

If you declare an invalid database, InterBase generates an error.

Declaring Local Variables

Programs that access databases may use local variables to:

- Transfer database field values to local variables for a host program to use
- Hold user input solicited from a user until the input can be passed to InterBase

Rather than use local variables, GDML can reference database fields inside a **for** loop. For example, you can select records with a GDML statement and then use a host language statement to list the database fields to print:

```
for c in cities
  printf ("%s %s %d %d %d\n", c.city, c.state, c.altitude,
      c.latitude, c.longitude);
end_for;
```

However, there are circumstances in which you need to use local variables with GDML. For example, you need to use local variables when you want to use a value you have found outside the context of a **for** loop. Also, you need to use local variables when passing field values to subroutines. You must declare local variables in the subroutines you call.

To tie host language variable declarations to database field types, GDML provides the **based_on** declaration. When you preprocess the program, **gpre** supplies the host variable with the attributes of the database field. For example, the following statement declares a host variable STATE_CODE with the same characteristics as the STATE field in the STATES relation:

```
based_on states.state state_code;
```

Gpre provides STATE_CODE with the same datatype (fixed-length string) as STATE. If you change the datatype of STATE, you must preprocess the program again. This causes the datatype of STATE_CODE to change as well. To synchronize host variables with the database fields they reference, always preprocess the program again after changing the metadata. However, you can use the program after you change the datatype without preprocessing again. In this case, InterBase converts the data from the new type to the old type the program uses. If it cannot make the conversion, InterBase returns a conversion error.

Starting a Transaction

Within a transaction, you read and write data. To start a transaction, you have three options. You can:

- Explicitly begin a transaction with a **start_transaction** statement. If you do not name the transaction, it becomes the *default transaction* named gds_$trans. The default transaction is the transaction **gpre** automatically starts when it encounters a GDML statement that is not prefaced with **start_transaction** statement.
- Let **gpre** start the default transaction.
- Explicitly begin a transaction with a **start_transaction** statement and name the transaction. A *transaction handle* is the name you assign to a transaction. If you assign a transaction handle, you must commit or rollback that transaction with its handle. For example:

```
commit my_trans;
```

For simple programs, allowing **gpre** to start transactions removes one statement from the program and makes it that much simpler. You can even allow **gpre** to start transactions automatically for moderately complex programs if their transaction requirements are simple. However, as your program grows, allowing **gpre** to start transactions can cause errors. Therefore, it is a good habit to always explicitly start and end transactions.

For more complex programs, you need to start and end transactions explicitly. Some reasons for doing this are:

- The **start_transaction** and **commit** statements indicate logical starting and stopping points in the program. They document program flow.
- Unless you start transactions explicitly, **gpre** must generate code before every GDML statement to check if the database is attached and the transaction started. If you explicitly handle these functions, you reduce the size of the preprocessed program.
- The **start_transaction** statement has a few options. You can:
  - — Reserve relations for access
  - — Specify transaction options
  - — Name the transaction

For more information on transactions, see the chapter on transaction management.

Embedding GDML Statements

You can embed GMDL statements in your program using either of the following methods:

- A **for** loop
- The **start_stream** statement

The **for** loop is sufficient for most programming needs. However, if you have to process multiple independent streams in parallel or process a stream and exit gracefully before the stream terminates, then you probably should use the **start_stream** statement.

The following sections discuss using each of these methods.

Using the For Loop

A GDML **for** loop specifies the conditions a record must satisfy to be included in a *record stream*. A record stream is a group of records that satisfy specified selection criteria. The GDML looping construct is very similar to a host language loop. For example, a host language **for** loop specifies the conditions under which it continues iterating. In a GDML **for** loop, InterBase executes each statement within the scope of the **for** loop for each record in the stream. When the record stream is exhausted, the loop terminates.

The following **for** loop includes in its record stream all the records in the STATES relation. The C **printf** statement displays several fields from the qualifying records:

```
for s in states sorted by s.state
  printf ("%s %s\n", s.capital, s.state_name);
end_for;
```

The following program consists of GDML statements embedded in a host language, C in this case. The program joins the STATES and CITIES relations from the atlas.gdb database and then displays values from several fields:

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

main()

{
ready atlas;

start_transaction
for s in states cross c in cities over state
```

```
    printf ("%s, %s, %d\n", c.city, s.state_name, c.population);
end_for;

commit
finish atlas;

}
```

The first line of the loop, *s in states*, is the *record selection expression* or *RSE*. The record selection expression includes a *context variable*, namely "s", used to qualify fields. The context variable is required in GDML.

You can use the following statements, within a **for** loop, for data manipulation:

- **store** statement, which inserts a new record into a relation.
- **modify** statement, which changes one or more field values in a record stream.
- **erase** statement, which deletes a record.
- Any host language statement.

Using the Start_Stream Statement

The alternative to the **for** loop is the **start_stream** statement. You use the data manipulation statements listed below:

- **store** statement, which inserts a new record into a relation.
- **modify** statement, which changes one or more field values in a relation.
- **erase** statement, which deletes a record.

If you use the **start_stream** statement, you must also use:

- **fetch** statement, which advances a record pointer through the record stream.
- **end_fetch** statement, which ends the advancement of the record pointer.

Additional statements you use are those specifically for manipulating *blobs* (basic large objects). Use the following statements to read and write blob data:

- the **create_blob** statement, which creates a blob for storage
- the **open_blob** statement, which opens a blob field for retrieval
- the **for_blob** statement, which retrieves data from a blob field
- the **get_segment** statement, which reads a portion of a blob field
- the **put_segment** statement, which writes a portion of a blob field
- the **close_blob** statement, which closes a blob after retrieval or manipulation

Handling Errors

When you use **gpre** to preprocess programs, you can receive parsing errors. These are errors that **gpre** encounters when parsing a command, such as an unrecognized word or invalid syntax. The error 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 you should report. If you encounter a bugcheck, submit a problem report to Interbase Software Corporation. If you are using InterBase on an Apollo, execute a **tb** (traceback) in the shell and save the output and a copy of the database.

InterBase returns error messages to programs 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 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.

For a detailed listing of error messages, refer to the appendix of the *Programmer's Reference.*

The following conceptual view of the status vector shows what it might include:

```
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 even 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 the highest, most general terms, and subsequent longwords provide additional information about the failure. If the statement is successful, InterBase returns a zero in the second slot.

To display the contents of the status vector you use the GDML library routine **gds_$print_status (gds_$status)** in conjunction with the **on_error** clause.

You use the **on_error** clause to specify the action a program should take if an error occurs during the execution of a GDML statements.

Gpre declares **gds_$status**, so you do not have to define it in your program.

The following **ready** statement uses the **gds_$print_status** routine to examine the contents of the status vector:

```
ready filename1 as atlas
  on_error
      gds_$print_status (gds_$status);
      printf ('Enter the file name: ');
      gets (filename1);
      if (filename1 [0] !='\n')
          {
              printf ("Giving up.\n");
              goto failure;
      }
      else
          goto ready1;
  end_error;
```

A more complicated program might use the status vector routine to check for:

- A validation error, **gds_$not_valid**
- A deadlock, **gds_$deadlock**

Example Error Handling Program

The following program changes the type of ski areas within a subroutine that returns the status of the change. Validation errors are handled within the subroutine, thus avoiding restarting either the transaction or the **for** loop. Deadlocks are handled by the main routine that rolls back and retries. Other errors cause the status to be printed, the transaction to roll back, and the program to exit.

Note

In C you investigate **gds_$status [1]** since C is zero based. In other languages, you investigate **gds_$status [2]**.

```
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_$true;
  start_transaction;
  for ski in ski_areas with ski.name = area_name
re_mod:
      modify ski using
          ski.type := area_type;
      end_modify
      on_error
          if gds_$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_$false;
            rollback;
            return;
        end_error;
    end_for
    on_error
        if gds_$status [2] <> gds_$deadlock then
            gds_$print_status (gds_$status);
        modify_type := gds_$false;
    end_error;
    commit;
end;

begin
    ready;
    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.
```

Ending the Transaction

To write your transaction's changes to disk and make the transaction's changes permanent, you have two options:

- You can include a **commit** statement in your program. The **commit** statement writes changes to the database, makes the changes visible to other users, and ends the transaction.

```
start_transaction;
  for ski in ski_areas with ski.name = area_name
      modify ski using
          strcpy (ski.type, area_type);
      end_modify;
  end_for;
commit;
```

- You can include a **save** statement in your program. A **save** statement commits the current transaction and starts a new one, maintaining the context for any active **for** loops or streams. For example:

```
start_transaction;
  for ski in ski_areas with ski.name = area_name
      modify ski using
          strcpy (ski.type, area_type);
      end_modify;
  end_for;
save;
```

To undo a group of changes in your program, include a **rollback** statement in your program. The following code fragment contains error handling. InterBase rolls back any changes if it encounters errors:

```
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);
    modify_type = gds_$false;
    rollback;
```

```
    return;
end_error;
```

If you issue a **commit** or **rollback** statement without specifying a transaction handle, InterBase ends the default transaction.

Note

Should your program terminate without completing a transaction or detaching the database, InterBase automatically rolls back the program's changes. If, for example, you ready a database, or start a transaction, remember to finish the database and commit the transaction. Not doing so can produce undesired results. However, if you use **gpre** without its -m option, it adds a **commit** to any **finish** statement.

Closing the Database

When you are finished with the database(s) you have accessed, you should close it with the **finish** statement:

```
finish atlas;
```

The variable *atlas* is the database handle declared in the **database** declaration.

If you have multiple databases open, you can close any number with the **finish** statement. For example:

```
database atlas = filename 'atlas.gdb';
database emp = filename 'employees.gdb
database fin = filename 'finance.gdb
 ⇓
finish atlas, emp;
```

You can also close all open databases with an unqualified **finish** statement. For example, the following statement closes all open databases:

```
finish;
```

Preprocessing Your Program with Gpre

After you code your GDML program, you must use **gpre** to preprocess the program. **Gpre** translates GDML statements and database variables into statements and variables that the host language compiler will accept.

The example below shows how to use **gpre** to preprocess a Pascal program that contains embedded GDML statements:

```
gpre geo_survey.epas
```

As it preprocesses the program, **gpre** supplies the database declaration and several other declarations that the host language compiler expects to find.

For more information on preprocessing programs with **gpre**, refer to chapter on preprocessing your program.

Interactive GDML

Before writing programs using GDML statements, you can experiment with your programming logic and strategy. You can do so by using **qli**, InterBase's query and update facility. **Qli** supports an interactive subset of GDML, so you can test your algorithms interactively before you code your program.

Of the GDML capabilities discussed earlier in this chapter, **qli** supports the following:

- All data manipulation verbs except for **start_stream** and **fetch**
- The **ready** and **finish** verbs, but with somewhat different meanings
- The **commit** and **rollback** verbs, but only with single transactions
- Simplified blob processing operations

For More Information

For more information on programming with embedded GDML, refer to:

- Chapter 5, *Retrieving Data*, for more information on retrieving data
- Chapter 6, *Writing Data*, for more information on storing, modifying, and deleting data
- Chapter 17, *Mixing SQL with Other Interfaces*, for information on mixing GDML with SQL and OSRI calls

Also see the GDML section of the *Programmer's Reference* for more information on the following statements discussed in this chapter:

- **commit** statement
- **database** declaration statement
- **end_fetch** statement
- **fetch** statement
- **finish** statement
- **for** statement
- **on_error** clause
- **ready** statement
- **save** statement
- **start_stream** statement
- **start_transaction** statement

For more information about interactive GDML, see the chapter on accessing data using interactive GDML in the *Qli Guide*.

Chapter 5
Retrieving Data with GDML

This chapter describes how to retrieve data using GDML programs.

Overview

When you select records with GDML you:

- Create a record stream with a GDML **for** loop or a **start_stream** statement.
- Specify the search conditions a record must satisfy in order to be included in a record stream.

Creating a Record Stream

The following sections describe the two ways of creating record streams with a:

- GDML **for** loop
- **start_stream** statement

InterBase uses the same internal mechanism for **for** loops and stream access. The external interface is the only difference. In practical terms, the **for** loop is sufficient for most programming needs. However, if you have to process multiple independent streams in parallel or process a stream and exit gracefully before the stream terminates, then you probably should use the **start_stream** statement.

The For Loop

The **for** loop sets up the conditions a record must satisfy to be included in a record stream. InterBase executes the statements within the **for** loop as long as there is a record in the stream. You can limit the qualifying records in a number of ways. For example, you can select only those records with specified field values.

The following **for** loop creates a record stream that consists of CITIES records that satisfy the selection criterion of only those cities with populations greater than or equal to the value of MIN_POP.

```
based_on cities.population min_pop;

printf ("Minimum population to display: ");
scanf (%d, &min_pop);

for c in cities
  with c.population >= min_pop
  printf ("There are %d in the city of %s.\n", c.population,
      c.city);
end_for;
```

Stream Access

Although the **for** loop is the preferred way to retrieve data because it reduces the amount of code required in your program, you may have to process:

- Multiple independent streams in parallel
- A stream that exits gracefully before input data terminates

In either case, you create a record stream with the **start_stream** statement.

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

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

main
{
short   end_of_stream;
char    done;

end_of_stream = 0;

ready;
start_transaction;

  start_stream geodata using c in cities
      sorted by c.latitude, c.longitude;
  fetch geodata
      at end end_of_stream = 1;
  end_fetch;

  while (!end_of_stream)
      {
      printf ("\n%s, %s, %d, %s, %s\n",
               c.latitude, c.longitude, c.altitude,
               c.city, c.state);
      printf ("Seen enough? (Y/N): ");
      done = getchar();
      if (done == 'Y')
          end_of_stream = 1;
      fetch geodata
          at end
              {
              end_of_stream = 1;
              printf ("Sorry, there is no more.\n");
          }
      end_fetch;
  }
  end_stream;
  commit;
  finish;
}
```

This program is functionally equivalent to the following fragment with a **for** loop, except the **for** loop does not give the user the choice to terminate the display:

```
for c in cities sorted by c.latitude, c.longitude
    printf ("%d, %d, %d, %s, %s\n",
             c.latitude, c.longitude, c.altitude,
             c.city, c.state);
end_for;
commit;
finish;
```

Fetching Records

Once you create a record stream with the **start_stream** statement, you have to get records one at a time before you can do anything to them. Getting records is the function of the **fetch** statement, which advances a record pointer through the record stream. Once you fetch a record, you can then read or write it.

You should provide error handling for the end of the record stream, so your program knows what to do when it runs out of records.

Selecting Records

To select records, you use a record selection expression (RSE) that specifies the conditions for record retrieval. The RSE lets you:

- Select the target relation
- Specify search conditions for the records you want to retrieve
- Sort the output
- Limit the number of records in a stream
- Perform a join operation
- Perform a project operation

Value Expressions

The elements within a record selection expression are *value expressions.* A value expression is a symbol or string of symbols from which InterBase calculates a value. InterBase uses the result of the value expression when executing the statement in which the expression appears. GDML's value expressions let you:

- Reference database fields with the *database field expression*
- Represent a string of ASCII characters with the *quoted string expression*
- Represent a decimal number with the *numeric literal expression*
- Perform arithmetic operations with the *arithmetic expression*
- Access a user-defined function (UDF)

The following sections discuss each of these value expressions.

Referencing a Database Field Expression

Perhaps the most frequently encountered value expression is the database field value expression, the means by which you refer to a field in a relation. The *dbfield-expression* occurs in several clauses of the record selection expression and in Boolean expressions.

The following statement uses database field expressions in the sort clause (c.city, c.state):

```
for c in cities sorted by c.city, c.state
  printf ("%s %s %d\n", c.city, c.state, c.population);
end_for;
```

The following example includes a database field expression used as a join term (over state), another used to sort the stream (s.state):

```
for c in cities cross s in states over state sorted by s.state
  printf ("%s %s %d %d\n", c.city, c.state, c.latitude,
      c.longitude);
end_for;
```

Representing a Quoted String Expression

Another common value expression is the quoted string, a string of ASCII printing characters, blanks, and tabs enclosed in single (') or double (") quotation marks. ASCII printing characters are:

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

You use the quoted string expression to:

- Provide the value in a value-based retrieva
- Assign a character string
- Assign a date field

The following query includes a quoted string expression ("Birchwood Acres"):

```
for ski in ski_areas with ski.name = "Birchwood Acres"
  printf ("%s %c %s %s\n", ski.name, ski.type, ski.city,
      ski.state);
end_for;
```

Representing a Numeric Literal Expression

GDML accepts a string of digits and optional decimal point and interprets it as a decimal number. This is called a *numeric literal expression*. It returns an error if the numeric literal expression exceeds the maximum length of the field as specified by its datatype.

The following query includes the numeric literal expression 40 in a value-based retrieval:

```
for c in cities with c.altitude < 40
  printf ("%s %s %d %d\n", c.city, c.state, c.latitude,
c.longitude);
end_for;
```

Performing an Arithmetic Expression

GDML supports the addition, subtraction, multiplication, and division of value expressions in retrievals and assignments. It evaluates the operators (+, -, *, /) used in the arithmetic expression in the normal precedence: addition, subtraction, multiplication, division. You can use parentheses to change the order of evaluation. If you use parentheses, InterBase evaluates the values inside the parentheses first.

The following query calculates and displays in miles the altitude of cities with altidudes greater than one mile:

```
for c in cities
  with c.altitude > miles_above_sea *5280
     printf("%s %d\n", c.cioty, c.altitude/5280);
end_for;
```

Accessing a User-Defined Function

User-defined functions (UDFs) allow you to provide custom functions that are not part of GDML. For example, you can write UDFs to:

- Provide mathematical functions such as absolute value, integrals, and fast Fourier transforms
- Provide character string and substring manipulation functions and list processing functions
- Manipulate InterBase's multi-dimensional arrays

User-defined functions let you implement a wide range of functions in any host language that InterBase supports. In addition, UDFs let you isolate platform-specific code.

User-defined functions exist in a library which:

- Eliminates unnecessary redundancy in your application code.
- Makes it smaller.
- Simplifies your code maintenance effort.

UDF libraries of common functions allow database users on different platforms to use functions having the same syntax.

You define for a database as many or as few UDFs from the library as you need for working with that database.

Working with UDFs

There are three steps to making a UDF work. You must:

1. Create the UDF
2. Define it to the database
3. Call it in your program

These steps are described in the following sections.

Creating the UDF

You write a UDF in a host language. You compile it, creating or adding it to a function library. You then need to rebuild the appropriate InerBase executables and libraries. You perform these steps on each platform that your application will be compiled and run on. Refer to the *Data Definition Guide* for details on writing UDFs, creating UDF libraries, rebuilding executables and libraries. The *Data Definition Guide* also discusses preprocessing, compiling and linking your library.

Defining the UDF to your Program

Before you can use the function, you define it for a particular database through **gdef** or **qli**. You cannot define a function in a host-language program. The database contains the UDF definition in the same way that it contains a trigger definition. Refer to the *DDL Manual* for details on defining UDFs to a database.

Calling the UDF in your Program

You can reference UDFs only in RSEs. You can *not* reference UDFs in SQL statements.

To reference a UDF, simply write its name, followed by a pair of matching parentheses, with its arguments inside the parentheses. For example, assume you had defined to the database a UDF named *abs* which returns the absolute value of its argument. This following RSE would print states that had a population change of more than 250000. The call to the UDF is straightforward:

```
for p in populations with
  abs(census_1970 - census_1980) > 250000
  printf ("%s\n", p.state);
```

Because a UDF is invoked by the DBMS, it runs on the node where the access method is, and not necessarily where the application is. Therefore, a UDF may use data other than expected, and might produce unexpected results.

Selecting Relations

The RSE's *relation-clause* identifies the relations from which you create the record stream.

The relation clause specifies a relation name and associates a context variable with the relation. For most host languages, **gpre** is not sensitive to the case of the context variable. For example, it treats "B" and "b" as the same character. However, C is case-sensitive unless you preprocess the program with the **either_case** option.

The context variable can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character. The following queries contain examples of context variables, in this case "s" for the STATES relation and "c" for the CITIES relation:

```
for s in states sorted by s.state
  printf ("%s\n", s.state_name);
end_for;

for c in cities cross s in states sorted by s.state
  printf ("%s %s\n",c.city, s.state_name);
end_for;
```

Retrieving Data from Views

This chapter discusses retrieving data from relations. InterBase also supports *views*, virtual relations consisting of records from one or more actual relations. A view is never stored, but is automatically "materialized" when you reference it. For purposes of retrieval, a view is identical to a relation.

Using a Database Handle

If you are accessing multiple databases, you should qualify relations with a *database handle*. The database handle identifies the database in which a relation can be found. Use the **database** declaration to associate a database handle with a database. For example:

```
database emp = filename 'employees.gdb';
```

Retrieving Data in Joined Relations

The RSE's **cross** clause creates relationships by combining records from two or more different relations in the same database. For example, the following query joins the STATES and CITIES relations on the STATE field:

```
for c in cities cross s in states over state sorted by s.state,
c.city
  printf ("%s %s %d %d\n", c.city, s.state_name, c.latitude,
      c.longitude);
end_for;
```

Unlike most other RSE clauses, the **cross** clause can be repeated to include as many relations as are necessary. The following query joins the CITIES, STATES, and TOURISM relations, and retrieves the city name, state name, population, and zip code of cities with tourist offices:

```
for c in cities cross s in states over state cross
  t in tourism over state, city
  sorted by s.state

  if (c.population.null == GDS_$FALSE)
      printf ("%s %s %s %d\n", c.city, s.state_name, t.zip,
          c.population)
  else
      printf ("%s %s %s\n", c.city, s.state_name, t.zip);

end_for;
```

The *join* operation lets you combine records from multiple relations. For example, the following **for** loop joins records from the STATES and CITIES relations to return information about cities and the states in which they are located:

```
based_on states.state statecode;

  ⇓

printf ("Enter state code [2 characters, uppercase]: ");
gets (%s, statecode);

for s in states cross c in cities over state
  with s.state = statecode
  printf ("%s is located in %s", c.city, s.state_name);
end_for;
```

This **for** loop returns the values of two fields from the two relations.

Using Nested For Loops for Outer Joins

Occasionally you want to combine data from two relations, matching values where possible, but not excluding records that have no match. This capability is sometimes called an *outer join*. GDML does not directly support an outer join, but does provide nested **for** loops to achieve the same results. **For** loops can be nested arbitrarily deep.

The following example lists baseball teams and their stadiums by state:

```
for s in states sorted by s.state
  printf ("%s\n", s.state_name);
  for b in baseball_teams with b.state = s.state sorted by
         b.team_name
      printf ("%s\t %s\n", b.team_name, b.home_stadium);
  end_for;
end_for;
```

When you want to combine data from relations in several databases, you must use nested **for** loops.

Specifying 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. You can also compare two database fields (for example, PARTS with COST > WEIGHT). 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.

The following query returns only those records for which it is true the value contains "CA":

```
for c in cities with state = 'CA'
  printf ("%s %d %d\n", c.city, c.latitude, c.longitude);
end_for;
```

This search condition results in a value of "true," "false," or "missing" for each record in the CITIES relation. Such an expression, where the value is true, false, or missing, is called a *Boolean test* and is expressed by a *Boolean expression*.

Using the Not, And, and Or Operators

GDML supports logical **not**, **and**, and **or** operators, evaluating them in that order unless you put an expression in parentheses to change the order of evaluation.

If your query has more than one search condition, such as cities with a population greater than 1,000,000 and an altitude above 200 feet, the truth of the Boolean test depends on the combination of search conditions. In the following example, only those CITIES records for which both conditions are true are returned:

```
for c in cities with c.population > 1000000 and
  c.altitude > 200
      printf ("%s %s %s %s\n", c.city, c.state, c.latitude,
         c.longitude);
end_for;
```

Tables 5-1, 5-2, and 5-3 show truth tables for these operators.

*Table 5-1. The **Not** Operator*

Value of A	Not A
True	False
False	True
Missing	Missing

*Table 5-2. The **And** Operator*

Value of A	Value of B	A and B
True	True	True
True	False	False
True	Missing	Missing
False	True	False
False	False	False
False	Missing	False
Missing	Missing	Missing

*Table 5-3. The **Or** Operator*

Value of A	Value of B	A and B
True	True	True
True	False	True
True	Missing	True
False	True	True
False	False	False
False	Missing	Missing
Missing	Missing	Missing

Using the GDML Operators

In addition to **not**, **and**, and **or**, GDML has the following operators:

- **Containing** to test for the presence of a string in a value expression. This test is not sensitive to the case of the characters in the string.

```
for ski in ski_areas with ski.name containing 'Acres'
   printf ("%s %s %s %c\n", ski.name, ski.city, ski.state,
      ski.type);
end_for;
```

  The **containing** operator also works with blobs. The following example searches the GUIDEBOOK field in the TOURISM relation for occurrences of the string "Empire", joins the TOURISM and STATES relations, and displays the state name and guidebook entry for all qualifying records:

```
for t in tourism cross s in states over state with t.guidebook
containing "Empire"
   printf ("%s\n", s.state_name);
   printf ("\n");
   for blob in t.guidebook
      blob.segment[blob.length] = 0;
      printf ("%s", blob.segment);
   end_for;
   printf ("\n");
end_for;
```

  If you search a blob using **containing** with a local program varible (usually of type char*), you must cast the blob data using a query of the form

```
for <relation> with <field>.char[<length>] containing <var>
```

  where *var* is the name of the local variable and *length* is the name of the field from or to which you want to convert.

  For more information about displaying data from blobs the chapter on using blob fields.

- **Starting with** to test for the presence of a string at the beginning of a value expression. This test is case-sensitive. For example:

```
for s in states with s.state_name starting with 'New'
   printf ("%s %d %s\n", s.state_name, s.statehood,
s.capital);
end_for;
```

- **Matching** to test for the presence of a string with wildcards in a value expression. The asterisk (*) wildcard matches a run of characters, and the question mark (?) matches a single character. For example:

```
for c in cities with c.city matching "*ton*"
   printf ("%s %s %d\n", c.city, c.state, c.population);
end_for;

for s in states with s.state matching "N?"
   printf ("%s %s\n", s.capital, s.state);
end_for;
```

- **Matching using** to test for the presence of a string using a wildcard search pattern you define. For example:

```
for c in cities with c.city matching '+ton+' using '+=?*'
   printf ("%s\n", c.city);
end_for;
```

- **Any** to test for the existence of at least one qualifying record in a relation or relations. This expression is true if the record stream specified by the RSE includes at least one record. For example:

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

- **Unique** to test for the existence of exactly one qualifying record. This expression is true if the record stream specified by *rse* consists of only one record. The following query displays the number of states for which there is only one city stored for that state:

```
for u in cities with unique c in cities with c.state = u.state
   printf ("%s is the only interesting city in %s\n", u.city,
u.state);
end_for;
```

- **Missing** to test for the absence of a value in database field expression. The following query displays STATES records for which the value of the CAPITAL field is missing:

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

Selecting a Missing Value

Any field can have a missing value. Rather than storing a value for the field, InterBase sets a flag indicating that the field has no assigned value. Unless you specify otherwise in the field's definition, GDML returns zero for missing numbers, blanks for missing characters, blanks for missing dates when returned as char datatypes, November 17, 1858 for missing dates when returned as date datatypes, and a null pointer for missing blobs.

For example, many of the records in the CITIES relation do not have a value for the POPULATION field. Cities with no stored population have the missing value flag set for that field. The following query lists cities with populations less than 200,000 or for which the population is missing:

```
for c in cities with c.population lt 200000 or
      c.population missing
  printf ("%s %s %d\n", c.city, c.state, c.population);
end_for;
```

You might choose to display records with missing values differently than complete records. For example:

```
for c in cities with c.population lt 200000
   or c.population missing sorted by c.population
   if (c.population.null != gds_$false)
      printf ("%s unknown\n", c.city');
   else printf ("%s, %d\n", c.city, c.population);
end_for;
```

The missing value sorts last and defaults to ascending in this example. However, the missing value is not really a value; it is a flag that indicates the field value is different from actual stored values.

In addition to the sorting order, missing values have the following special considerations:

- If you perform some statistical operation involving a field that is missing for a given record, that record is ignored in the computation.
- A missing value never equals another value, even if the other value is also missing.
- A field with a missing value cannot participate in a join.

For more information about defining alternate missing values, see the *Data Definition Guide*.

Sorting Retrieved Records

The RSE's *sorted-clause* orders the record stream by the values of one or more sort keys. You can sort a record stream:

- Alphabetically
- Numerically
- By date
- By any combination of the above.

You can also specify if you want the sort to be case-sensitive of not. When you include a sorted clause in your RSE, you have the following options:

- **ascending**, the default order which sorts from A to Z, 0 to 9
- **descending**, which sorts Z to a, 9 to 0
- anycase, which ignores case differences
- exactcase, the default, which sorts uppercase before lowercase

If you specify **anycase,** InterBase temporarily converts all data to uppercase. Therefore, if two fields are identical in spelling but different in case, either one can appear first.

The following query sorts the CITIES relation by decreasing population:

```
for c in cities sorted by descending c.population
  printf ("%s %s %d\n", c.city, c.state, c.population);
end_for;
```

The **sorted by** clause lets you have as many sort keys as you want. The greater the number of sort keys, the longer it takes to execute the query.

Each sort key can specify whether the sorting order of the sort key is **ascending** (the default order for the first sort key) or **descending**, **anycase** or **exactcase**. The sorting order is "sticky"; that is, if you do not specify whether a particular sort key is **ascending** or **descending**, **gpre** assumes you want the order specified for the previous key. Therefore, if you list several sort keys, but only include the keyword **descending** for the first key, all keys are assumed to be in descending order. **Anycase** and **exactcase** are also "sticky."

Limiting Retrieved Records

GDML lets you limit the number of records in a record stream with the RSE's *first-clause*. For example:

```
for first 5 s in states sorted by s.state
  printf ("%s %s %s\n", s.capital, s.state_name,
      s.statehood.char[12]);
end_for;
```

When you use the **first** clause, you should also sort the record stream. Otherwise, you retrieve *n* random records, and not necessarily the same records each time.

Although the syntax calls for a numeric value, you can use any value expression that fits. For example, you might ask for the:

- *first 3-1 s in states*
- first (variable1 - variable2) s in states

In all cases, the value expression must evaluate to a number, so *first five states* is not legal. Finally, **gpre** truncates any fractional portion; therefore, *first 4.2 s in states* becomes *first 4 s in states*.

Projecting on Fields

The RSE's **reduced to** clause provides the relational *project* operation, retrieving only the unique values for a field from a record stream. When you ask for a record stream projected on a field or fields, InterBase considers the list of fields and eliminates records that do not have a unique combination of values for those listed fields. For example, the following query returns only the first occurrence of a value for the STATE field and ignores subsequent duplicates:

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

For More Information

See the GDML section of the *Programmer's Reference* for entries for:

- Value expression
- Boolean expression
- Record selection expression

For a discussion of retrieval using interactive GDML, see the *Qli Guide*.

For more information on user defined functions, see the *Data Definition Guide*.

Chapter 6
Writing Data with GDML

This chapter discusses how to write data using field value assignments and the manipulation of data using GDML.

Overview

To write data with GDML programs you use the following statements:

- **store** statement, which stores new data.
- **modify** statement, which changes existing data.
- **erase** statement, which deletes data.

To assign values to fields,use the **store** and **modify** statements with the assignment statement of whatever host language you are using.

Storing Records

The **store** statement inserts a new record into a relation. You can use the **store** statement inside a **for** loop or **start_stream** statement. Additionally, you can use it as a stand alone statement: without the **for** loop or **start_stream** statement. The source of values for an assignment can be any combination of the following:

- Value expressions
- Prompting expressions
- Host variables
- Field values

The following sections describe how to store values using these assignment values.

Using Value Expressions

A value expression can provide data in the form of a quoted literal expression or a numeric. The following **store** statement inserts a new record into the SKI_AREAS relation using constant values for assignments:

```
store ski in ski_areas using
   strcpy (ski.name, "Rte. 16");
   strcpy (ski.city, "N. Conway");
   strcpy (ski.state, "NH");
   ski.type = 'N';
end_store;
```

Using Prompting Expressions

A prompting expression reads an input value and assigns it directly to a database field. The following **store** statement inserts a new record into the SKI_AREAS relation using prompts:

```
store ski in ski_areas using
   printf ("Enter name of resort: ");
   gets (ski.name);
   printf ("Enter city: ");
   gets (ski.city);
   printf ("Enter state code: ");
   gets (ski.state);
   printf ("Enter type: ");
   gets (ski.type);
end_store;
```

Using Host Variables

You can assign values to host variables, and then assign those values to database fields. For example, the following **store** statement acquires all of its values through host variables:

```
based_on ski_area.name skiname;
based_on ski_area.city skicity;
based_on ski_area.state skistate;
based_on ski_area.type skitype;

      ⇓

printf ("Enter name of resort: ");
gets (skiname);
printf ("Enter city: ");
gets (skicity);
printf ("Enter state code: ");
gets (skistate);
printf ("Enter type: ");
gets (skitype);
store ski in ski_areas using
   strcpy (ski.name, skiname);
   strcpy (ski.city, skicity);
   strcpy (ski.state, skistate);
   ski.type = skitype;
end_store;
```

Using Field Values

You can use field values you create a record stream with an outer **for** loop. This option lets InterBase do some of the work involved in storing new records. This is a two step process:

1. Provide an RSE to find the record that has the values you want to clone.
2. Store a new record by assigning values from the old record to the new record.

For example, suppose you want to store a record for a city that is very close to another city. Since the CITIES relation contains mostly geographical data you can duplicate all of the values except for the city's name. Also, the assignments from the outer loop can include operations appropriate to the host language you use.

The following **for** statement finds a specific record in CITIES and stores a new CITIES record using some of the values from the record identified in the outer **for** loop:

```
based_on cities.city villeancienne;

      ⇓

printf ("Enter city to clone: ");
gets (villeancienne);
for oldcity in cities with oldcity.city = villeancienne
   store newcity in cities using
      printf ("Enter name of new city: ");
      gets (newcity.city);
      strcpy (newcity.state, oldcity.state);
      newcity.population = oldcity.population;
      newcity.altitude = oldcity.altitude;
      strcpy (newcity.latitude_degrees,
         oldcity.latitude_degrees);
      strcpy (newcity.latitude_minutes,
         oldcity.latitude_minutes);
      newcity.latitude_compass = oldcity.latitude_compass;
      strcpy (newcity.longitude_degrees,
         oldcity.longitude_degrees);
      strcpy (newcity.longitude_minutes,
         oldcity.longitude_minutes);
      newcity.longitude_compass = oldcity.longitude_compass;
   end_store;
end_for;
```

In the following code fragment, the **for** statement creates a record stream from which the **store** statement takes some values. Other values are provided by the user's response to prompts for values. Unreferenced fields are automatically assigned the missing value:

```
based_on cities.city villeancienne :

      ⇓

printf ("Enter city to clone: ");
gets (villeancienne);
for oldcity in cities with oldcity.city = villeancienne
  store newcity in cities using
      printf ("Enter name of new city: ");
      gets (newcity.city);
      strcpy (newcity.state, oldcity.state);
```

```
      newcity.population = oldicty.population;
      newcity.altitude = oldcity.altitude;
      strcpy (newcity.latitude_degrees,
         oldcity.latitude_degrees);
      strcpy (newcity.latitude_minutes,
         oldcity.latitude_minutes);
      newcity.latitude_compass = oldcity.latitude_compass;
      strcpy (newcity.longitude_degrees,
         oldcity.longitude_degrees);
      strcpy (newcity.longitude_minutes,
         oldcity.longitude_minutes);
      newcity.longitude_compass = oldcity.longitude_compass;
   end_store;
end_for;
```

Storing or Modifying Records with Missing Values

When you store or modify records with missing values, you have several options for assigning the missing value to a field. You can:

- Ignore the missing value field in the **store** statement
- Assign the explicit missing value defined for that field
- Reference the null flag for that field

The following sections discuss these options.

Ignoring the Field

If you fail to assign a value to a field in a **store** statement, InterBase sets a flag for that field indicating that its value is missing. Consider the following **store** statement:

```
store c in cities using
  printf ("Enter city name: ");
  gets (c.city);
  printf ("Enter state code: ");
  gets (c.state);
end_store;
```

This **store** statement does not include assignments to the LATITUDE, LONGITUDE, POPULATION, or ALTITUDE fields, so InterBase sets the missing value flag for all existing records until you provide a value.

This is not the case if the **store** contains conditional assignments. For example:

```
store c in cities using
  if strcmp(prog_flag, "true") =0
      {strcpy(c.city, "foo");
       strcpy(c.state, "MA");}
  else
      strcpy(c.city, "foo");
end_store
```

If the else clause in the preceding program is executed, state may containg garbage. If you use conditional assignments, you must explicity assign a null value. For more information on null values, see the section on referencing the null flag.

Assigning an Explicit Missing Value

When you create or modify the field definition using **gdef**, you have the option of supplying a missing value that the database software returns in place of zeroes or nothing. The substitute missing value should not overlap with any valid value. For example, you might assign a missing value of "-1" for the LATITUDE_DEGREES and LATITUDE_MINUTES field, and the missing value of "X" for the LATITUDE_COMPASS field. Neither of these values conflicts with a valid value.

When you assign the explicit missing value to a field as part of a **store** or **modify** statement, InterBase sets the missing flag for that field in that record. For example, store a new city and assign the explicit missing value -1 to LATITUDE_DEGREES:

```
store c in cities using
  printf ("Enter city name: ");
  gets (c.city);
  printf ("Enter state code: ");
  gets (c.state);
  strcpy (c.latitude_degrees,"-1");

      ⇓

end_store;
```

For information about specifying a substitute missing value, see the chapter on defining fields in the *Data Definition Guide*.

Referencing the Null Flag

As shown in the preceding sections, if you assign the explicit missing value to a field or do not assign a value at all, InterBase sets the missing flag for that field in that record. The missing flag is part of the stored record instance, and is also called the *null flag*.

You can reference the null flag for a record by using a database field expression with an appended **.null**. For example, the expression *c.population.null* references the null flag of the POPULATION field in a record stream for which you have declared a context variable *c*.

The advantage of addressing the null flag directly is you can decide at run-time whether a field should be left missing. However, direct use of the null flag requires some more attention to detail. Specifically, if you reference the null flag with **null** anywhere in a statement or its substatements, you have to continue referencing it. The following example looks for a record in the CITIES relation. If the value of POPULATION for that city is null (that is, missing), the **modify** statement changes POPULATION to *10*:

```
printf ('Enter city name:');
gets (cityname);
printf ('Enter state code: ');
gets (statecode);
for c in citieswith c.city = cityname and c.state = statecode
   if (c.population.null)
       {
       modify c using
           c.population = 10;
           c.population.null = GDS_$FALSE;
       end_modify;
       }
end_for;
```

There are two assignments in the **modify** statement:

- c.population = 10;

  This statement assigns the value *10* to POPULATION for the specified city.

- c.population.null =gds_$false;

  This statement assigns **gds_$false** to the null flag. Because you referenced the null flag in the **if** statement, you must reset the value of the null flag to indicate it was changed.

Had you not referenced the null flag in the **if** statement, and used a **missing** value expression instead, the second assignment would not have been necessary. Had you referenced the null flag as shown but made only the first assignment, the value of POPULATION would have remained null.

If you had to make a field value null and invalidate existing data, set the field's null flag to **gds_$true**. For example, the following program modifies the population of a selected city if the population is initially there, but you want to change it to missing. This discards the value previously in the field:

```
/* program map */

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

database db = filename 'atlas.gdb';

based on cities.city cityname;
based on cities.state statecode;

main()
{
```

```
    printf ("Enter city name: ");
    gets (cityname);
    printf ("Enter state code: ");
    gets (statecode);
    for c in cities with c. city = cityname and
        c.state = statecode
            if (c.population.null == GDS_$FALSE)
                {
                modify c using
                    c.population.null = GDS_$TRUE;
                end_modify;
                }
    end_for;

    for c in cities with c.city = cityname and
            c.state = statecode
        printf ("%s %d\n", c.city, c.population);
    end_for;

    rollback;
    finish;
}
```

Modifying Field Values in Records

You can modify field values inside a **for** loop or a **start_stream** statement. The source of values for an assignment can be any combination of the following:

- Value expressions
- Prompting expressions
- Host variables
- Field values

Value Expressions

The following example updates the POPULATION field of the CITIES relation using value expressions. The modification takes place inside a **for** loop:

```
based_on states.state statecode;
long multiplier;

      ⇓

printf ("Enter state code: ");
gets (statecode);
printf ("Enter population multiplier: ");
scanf ("%l", &multiplier);
for c in cities with c.state = statecode
  printf ("Current population: %f\n", c.population);
  modify c using
      c.population = c.population * multiplier;
  end_modify;
end_for;
```

Prompting Expressions

You can also prompt for a new value and read the response as the new value, as does the following **modify** statement:

```
for c in cities with c.state = hostvar
  printf ("Current population: %d\n", c.population);
  modify c using
      printf ("Increase population to: ");
      scanf ("%l", &c.population);
  end_modify;
end_for;
```

Modifying Views

If a view references a single relation, in most cases you can modify it just as you would a relation. However, you cannot modify records through views that reference more than a single relation or views that are created with the **qli restructure** operation.

This restriction obviously causes problems if your update involves more than one relation. As part of the normalization process described in the *Data Definition Guide*, one complex relation was broken down into several relations. The resulting relations are related through the value of join fields. As you update fields in or delete records from a database, be sure you update join fields wherever they appear.

For example, if you change an employee's identification number, you should also change it in all the related relations, such as salary and job history, medical history, and so on. Otherwise, the means by which you join records from different relations disappear.

In such case, you may often define a view which provides a unified view of all the related data, and allow updates to it directly. To permit this update you must specify a *trigger* that automatically performs the updates to the underlying relation. See the *Data Definition Guide* for more information about triggers.

In order to avoid overlooking updates of join fields, you need to modify all related join fields in different relations at the same time.You can do this with a sequence of **modify** statements, nesting the updates in a **for** loop. However, using triggers is often an easier solution.

In both cases, the whole operation is under transaction control, so all updates take place or none do.

Deleting Records

To delete records use the **erase** statement. The **erase** statement must be used inside a **for** loop that selects the record(s) for deleting or in the procession of statements within the **start_stream** and **end_stream** statements. The following example erases records from the CITIES relation:

```
based_on states.state statecode;

  ⇓

printf ("State to remove cities from: ");
gets (statecode);

for c in cities with c.state = statecode
  erase c;
end_for;
```

As with field updates, you cannot delete records from views that reference more than a single relation, unless you provide triggers to perform the action.

Casting

With casting you can perform datatype conversions. Casting moves data to and from different datatypes for input and display.

For example, you can convert:

- floating point data to strings
- strings to integer
- dates to strings (provided the strings contain only numeric characters)

To cast a field, append a casting name to the name of the field from or to which you want to convert.

The following C statement assigns today's date to a field:

```
strcpy(s.statehood.char[6], 'today');
```

The following two tables list the casting name in the leftmost column and the corresponding host language datatype name in other columns.

Note

InterBase does not support casting from or into the blob or array datatypes.

Table 6-1. Conversions Supported by Casting (non-Ada)

Casting Datatype Name	Host Language Datatype Name					
	BASIC	C	COBOL	FORTRAN	Pascal	PL/I
short	short	short	s9(4)comp	I*2	integer16	binary(15)
long	long	long	s9(9)comp	I*4	integer32	binary(31)
float	single	float	comp-1	real	real	float(24)
double	double	double	comp-2	real*8	double	float(53)
char[n]	string	char[n] *see restrictions*	x(n)	character dimension(n)	array[1...n] of char	character
string[n]	not supported	see restrictions	not supported	not supported	not supported	not supported
varying[n]	string	not supported	x(n)	character dimension	array[1...n] of char	character (varying)

Table 6-2. Conversions Supported by Casting (Ada)

Datatype	VERDIX and VAX
short	short_integer
long	integer
float	float
double	long_float
char[n]	string(1..n)
string[n]	not supported
varying[n]	string(1..n)

Gpre usually takes care of datatype conversions. For example, if you are using a language that does not support **varying**, **gpre** does the conversion for you. However, if you use **gpre**'s casting capability, note the following restrictions:

- Make sure that you pick a target datatype that is supported for your language. Otherwise, you will receive an error.
- If you have cast a field once in a request, you must continue referencing the field with its cast datatype throughout the request. Doing otherwise results in an error.
- If you are programming in C and have preprocessed your program using the **string** switch, use **.string[***integer***]** instead of **.char[***integer***]**. The **char** datatype is a null-terminated string unless you use the **string** switch when you preprocess your program; this switch results in an array of bytes.

For More Information

For information on writing transactions to disk, see the section on ending transactions in Chapter 4, *Getting Started with GDML*.

For more information about casting, see Chapter 10, *Using Date Fields*.

For information about storing, modifying, or deleting records, see the GDML section of the *Programmer's Reference* for entries for the following statements and expressions:

- Record selection expression
- **for**
- **store**
- **modify**
- **erase**
- **start_stream**

For more information about the use of triggers, see the chapter on security in the *Data Definition Guide*.

Chapter 7
Using Arrays

This chapter describes the basic array concept and how to use multi-dimensional arrays.

Overview

An InterBase array is a database field that is subdivided to store a large number of related data elements in a structured fashion.

You use an array when all three of these conditions exist:

- The data elements naturally form a set. (Individual elements are significant only in the context of the other elements.)
- You want to represent and control the entire set of data elements as a single database field.
- You want the capability to identify and access each element individually.

Storing data elements in an array is an alternative to lumping them together in a blob where they cannot be distinguished, or to spreading them out over many fields where

they lose their cohesiveness, and are more difficult to maintain as a set, and consume more system resources in overhead than an array.

If you want to keep the data elements distinct from each other, yet you rarely do field-level operations on them separately, then array storage is likely to be more appropriate than separate field storage.

As an example of where you might use an array, suppose that you have an application that accepts 15 input readings from 10 machines 5 times a second. Depending on what you consider their most important aspects to be, there are several ways to define storage for this data. If you most often use time/input readings as a set, you would store them in an array field, with the machine identifier in another field in the record. Or, if you had some higher-level grouping by which to organize the information, you could store all three kinds of elements in one three-dimensional array.

Consider another example. If encoding a pixel on a color screen takes one 16-bit word and you are encoding a 1000x1000 pixel screen, one image would take 1 million words of storage. This kind of data has little meaning stored in separate fields of a relation. However, storing this data in one array permits each word to be accessible individually, while it is handled conceptually at the record level as an image.

Basic Array Concepts

Because array terminology differs from language to language, the basic concepts are coverd here in order to provide a clear base for discussion.

Parts of an Array

An *array* is a rectangular, multi-dimensional data structure that holds data elements of the same type in subdivisions called *cells*. The distance along each dimension of the structure is marked by integer-based indexes, or *subscripts*. An array always has a name. The *bounds* of an array are the lowest and highest permissible subscript integers. The value of a dimension's lower bound is called its *base*.

An array named *sample* shown here is *one-based*. In one dimension, it has a lower bound of 1 and an upper bound of 4. In the other dimension, it has a lower bound of 1 and an upper bound of 5.

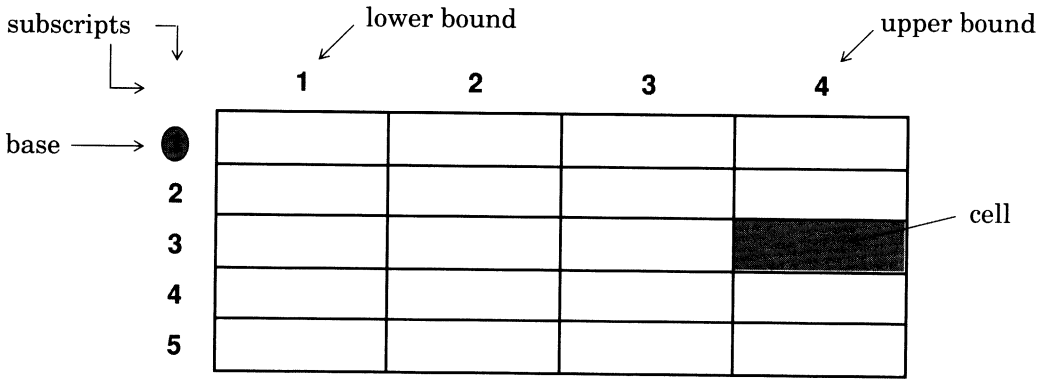

Figure 7-1. Sample Array

Referencing a Cell

A cell is uniquely identified by a set of subscripts, one subscript from each array dimension. A cell's subscript integers identify the point of intersection of the dimensions at which the cell lies. In general, you refer to a cell by giving the name of the array in which it appears and its subscript integers.

Differences in Host Language References to a Cell

Cell references in each host language differ in the details of using:

- Delimiters
- Array base
- Subscript bounds
- Range of subscript values
- Number of dimensions
- Order of subscripts

Delimiter Differences

Some languages enclose array subscripts in parentheses, others in brackets. Table 7-1 illustrates the differences.

Table 7-1. Delimiter Differences

Language	Delimiters
Ada	array (x,y)
Basic	array (x,y)
C	array [x][y]
COBOL	array (x,y)
FORTRAN	array (x,y)
Pascal	array [x,y]
Pl/I	array (x,y)

Array Base Differences

In some languages, every array's base is one (1), in others, it is zero (0). In other languages, there is no default base; the programmer must explicitly specify the lower bound of each dimension which then is used as the dimension's base. Table 7-2 illustrates the differences.

Table 7-2. Array Base Differerences

Language	Array Base
Ada	Explicit
Basic	0
C	0
COBOL	1
FORTRAN	Explicit (default is 1)
Pascal	Explicit
Pl/I	Explicit (default is 1)

Subscript Bound Differences

Some languages limit the largest possible value that can be used as the upper bound of an array dimension. The values differ from language to language and from one platform to another. Refer to host language and platform specific documentation for more information.

Range of Subscript Value Differences

Languages such as Ada, FORTRAN, Pascal, and PL/I permit the use of negative numbers as array subscripts; others do not. Table 7-3 illustrates the differences.

Table 7-3. Range of Subscript Value Differences

Language	Range Permits Negatives
Ada	Yes
Basic	No
C	Yes, by referencing memory outside of array
COBOL	No
FORTRAN	Yes

Table 7-3. Range of Subscript Value Differences continued

Language	Range Permits Negatives
Pascal	Yes
Pl/I	Yes

Number of Dimension Differences

Each language supports a different number of array dimensions. Table 7-4 illustrates these differences.

Table 7-4. Number of Dimension Differences

Language	Number of Dimensions
Ada	InterBase limit is 16
Basic	InterBase limit is 16
C	InterBase limit is 16
COBOL	InterBase limit is 16
FORTRAN	7
Pascal	InterBase limit is 16
Pl/I	InterBase limit is 16

Order of Subscript Differences

In some languages, the subscripts of an array appear in *row / column* order; in others, *column / row* order. Technically, the order of subscripts is called either *row major* or *column major* order. Table 7-5 illustrates these differences.

Table 7-5. Order of Subscript Differences

Language	Order of Subscript
Ada	Row major
Basic	Row major
C	Row major

Table 7-5. Order of Subscript Differences continued

Language	Order of Subscript
COBOL	Row major
FORTRAN	Column major
Pascal	Row major
Pl/I	Row major

Consider the difference between Pascal and FORTRAN. Pascal uses row major order and FORTRAN uses column major order. Figure 7-2 shows an array named wave_-height and how you reference a cell in Pascal and FORTRAN.

Array name is *wave_height*

	1	2	3	4
1				**12.45**
2				
3				
4				
5				

Figure 7-2. Wave_Height Array

Here's how the cell in the above host language array would be referenced in order to retrieve the value 12.45:

Pascal: `wave_height[1,4]`
FORTRAN: `wave_height(4,1)`

Figure 7-3 shows the Pascal and FORTRAN references to cells for an array named alpha:

Array name is alpha

	1	2	3	4
1	A	B	C	D
2	E	F	G	H

Figure 7-3. Alpha Array

Pascal: row major				**FORTRAN: column major**			
alpha[1,1]	=	A	Row 1	`alpha(1,1)`	=	A	Column 1
alpha[1,2]	=	B	Row 1	`alpha(1,2)`	=	E	Column 1
alpha[1,3]	=	C	Row 1	`alpha(2,1)`	=	B	Column 2
alpha[1,4]	=	D	Row 1	`alpha(2,2)`	=	F	Column 2
alpha[2,1]	=	E	Row 2	`alpha(3,1)`	=	C	Column 3
alpha[2,2]	=	F	Row 2	`alpha(3,2)`	=	G	Column 3
alpha[2,3]	=	G	Row 2	`alpha(4,1)`	=	D	Column 4
alpha[2,4]	=	H	Row 2	`alpha(4,2)`	=	H	Column 4

About Array Fields

The following sections discuss how InterBase handles arrays.

Characteristics of InterBase Arrays

An InterBase array is a database field. Through **gdef**, an InterBase array can be defined to support any InterBase datatype *except* blob and array. When defined, an InterBase array:

- Uses 1 as the default lower bound for each dimension
- Supports subscript values from (-2^{32}) to $((2^{32}) -1)$
- Supports up to 16 dimensions

GDML statements in host language programs can reference either an entire InterBase array, or a specific cell of an array. At runtime, an InterBase array uses static storage.

An Array is a Database Field

A host language array is a set of *variables* that uses memory storage. An InterBase array is a database *field* that uses disk storage. You read values from an array field into host language variables in the same manner you read values from other database fields. The same is true for writing values to an array field.

You define an array to a specific database through **gdef**, just as you define any other database field.

Array Datatypes

In an InterBase array, cells holds data elements of the same datatype. You cannot mix elements of different datatypes in the same array.

You can specify an array datatype as:

- Short
- Long
- Float
- Double
- Char
- Varying
- Date

When you define the array, you can specify it as any one of the InterBase datatypes *except* as a blob or as an array. As with all supported datatypes other than blobs, InterBase automatically reformats data as necessary during data transfer from one platform to another.

Defining the Characteristics of an Array Field

In an array definition, you specify:

- Array name
- Type of elements it holds
- Number of dimensions
- Subscript range of each dimension

You can override the default lower bound by specifying subscripts within the range of valid values.

- If you do *not* specify the subscript range for a dimension, the range value you give defaults to *one-based*. Therefore, if you specify (3), you get three elements whose subscripts are 1, 2, and 3.
- If you *do* specify the subscript range, the lower value becomes the lower bound. For example, if you specify (-2:1), you get four elements whose subscripts are -2, -1, 0, and 1.

The following definition is part of more detailed examples that appear later:

```
% define field high_low_temperature short (12,2)
```

This defines *high_low_temperature* as a two-dimensional array of shorts in which the subscripts of:

```
dimension 1 range from 1 to 12, and
dimension 2 range from 1 to 2.
```

The next three examples illustrate how to define a variety of datatypes, sizes, bounds, and dimensions.

Example —A Three-Dimensional Array

Example 1 defines a three-dimensional array of longs. The subscripts are listed in the following table:

Dimension	Range
Dimension 1	0 - 99
Dimension 2	10 - 20
Dimension 3	1 - 2

```
% define field example1 long (0:99, 10:20, 1:2) scale -2;
```

Example 2—A One-Dimensional Array

Example 2 defines a one-dimensional array of 16-byte character strings. The subscripts are listed in the following table:

Dimension	Range
dimension 1	1 - 3

```
% define field example2 char [16] (3);
```

Example 3—Another One-Dimensional Array

Example 3 defines a one-dimensional array of 32-byte character strings. The subscripts are listed in the following table:

Dimension	Range
Dimension 1	0 - 5

```
% define field example3 varying [32] (0:5) fixed;
```

The attribute *fixed* indicates that the elements of Example 3 are not truncated at the first null byte.

Note

As with other fields, you can define an array as a global field in the database or as a local field in a relation. For more information on defining global fields, refer to the chapter on defining fields in the *Data Definition Guide*.

Through **gdef**, you can define an array with negative subscripts. If your host language does not support negative array subscripts, **gpre** adjusts references, as described later in the section *Accessing an Array from Multiple Languages*.

Array definitions to **gdef** are *always* in row major order.

Using Static Runtime Storage

When **gpre** precompiles your source file, it creates array declarations in your host language code that allocate static storage for the array. This has two effects. You can:

- Use very large arrays.
- *Not* use arrays in recursive requests (i.e., those in which you specify the "level" request option).

Recursive requests use stack storage to hold the intermediate results of each level of recursion. Even a relatively small array (100x200) uses a large amount of stack storage memory (80K). A recursive request using an array would cause multiple images of the array to be stored on the stack which could quickly exceed the capacity of most reasonably sized stacks. A stack failure generally is not recoverable.

If InterBase were to allow recursive requests with arrays, it would have to severely limit array size. Given the choice, recursive requests are not permitted, allowing arrays sizes that are practically unlimited.

Using InterBase Arrays

This section explains how to refer to InterBase arrays in source code and discusses correct and incorrect results produced by various references when the file is precompiled, compiled, and executed. It also describes where you can use array references, the formal syntax, and various interactions that can take place between host languages and InterBase array references. This section ends with several complete examples.

Overview

In the source code, use an array reference to determine:

- How you write the reference
- How **gpre** processes the reference
- What errors **gpre** reports
- What errors the host language compiler reports
- What errors you may receive at runtime

You can reference an InterBase array field from only two places in source code. In a:

- Record Selection Expression (RSE)
- Host language statements within the block of a **for**, **modify**, or **store** loop.

There are two limitations on using an array reference in one of the two allowable places in source code. You can *not* reference an array in:

- Any SQL statement
- A recursive request (i.e., one in which you specify the "level" request option)

The following sections describe the two places for referencing an array field.

Using Array References in RSEs

You write an array reference in an RSE in order to use the value of a particular array element's contents to resolve the Boolean expression. In an RSE, you must fully specify the subscripts of the array reference.

Formal Syntax

An array reference in an RSE uses this syntax:

```
<array-reference> ::=
   context-variable.array-name [<subscript-commalist>] |
   context-variable.array-name (<subscript-commalist>)
<subscript-commalist> ::=
   <subscript> | <subscript>, <subscript-commalist>
<subscript> ::=
   <integral-expression>
```

- In <subscript-commalist>, multiple subscripts must be in row- or column-major order, according to the host language's way of handling array subscripts. However, regardless of host language, subscripts in RSE references *must* be enclosed by parentheses or square brackets and separated by commas.
- <integral-expression> consists of an expression of integer operations which can contain:
  - — Host language variables
  - — Database field references
  - — Compiler's manifest constants
  - — Integer constants
- <integral-expression> must evaluate to an integer value when the program is:
  - — Precompiled with **gpre**
  - — Compiled with the host language compiler
  - — At runtime

Note

Gpre accepts either parentheses or brackets, provided they match each other in the same reference. (Pascal and C uses only brackets) For example:

```
                x.wave_height[1,5]
or              x.wave_height(1,5)

but not:        x.wave_height(1,5]
```

How Gpre Processes the Reference

While precompiling your source code, **gpre** tries to ensure the Boolean expression in an RSE resolves at runtime. To the extent it can, **gpre** checks the RSE at runtime for actual values to compare under the specified condition. Although **gpre** cannot tell whether or not an actual value is present in the referenced cell at runtime, it *can* verify an actual cell is referenced. **Gpre** generates an error if it finds an array reference in an RSE with the wrong number of subscripts, or subscripts beyond the defined bounds. Other errors related to incorrect array references are described in a later section, *Array Reference Errors.*

You must handle any conditions relating to the actual value, such as ensuring the RSE compares values of the same type (i.e., an array element of type *char* with a literal or variable of type *char*).

The following examples refer to a hypothetical application monitoring the performance of jet engines in a test facility. The name of the array is *readings*.Among many other readings, the application stores values for oil pressure and temperature, fuel pressure, and engine RPMs in a relation called *engine_test*. Two of the fields in the relation are:

- A numeric field called SAMPLE_TIME
- A one-dimensional numeric array called READINGS

Figure 7-4 shows the array called READINGS.

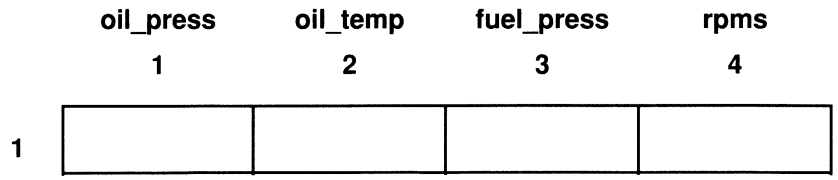

Figure 7-4. One-dimensional Array

After a test we will run the application and enter a valid temperature value into the variable *max_temp*. Assume the value of *oil_temp* is 2. This RSE in the application's code prints when the oil temperature exceeds the value entered. For example:

```
for x in engine_test with x.readings[oil_temp] > max_temp
  print x.sample_time
```

In this RSE, *x.readings[oil_temp]* is the array reference. It uses a variable, but could have used a constant (or the name of a database field):

```
for x in engine_test with x.readings[2] > max_temp
  print x.sample_time
```

At runtime, the reference actually causes a comparison of the value stored in the array in the specified cell with the value of *max_temp*.

If the array *readings* are defined to store five sets of readings per second, it would be a two-dimensional array. Figure 7-5 shows a two dimensional array.

		oil_press 1	oil_temp 2	fuel_press 3	rpms 4
Set 1	1				
Set 2	2				
Set 3	3				
Set 4	4				
Set 5	5				

Figure 7-5. Two-dimensional Array

By supplying *max_temp* and *set*, this RSE prints any time the oil temperature exceeded the value entered for the given set:

```
for x in engine_test with x.readings[set,oil_temp] > max_temp
  print x.sample_time
```

The order of *set, max_temp* is row major, which would be appropriate in C. In FORTRAN, it is:

```
for x in engine_test with x.readings[oil_temp,set] > max_temp
  print x.sample_time
```

Using Array References in Host Language Statements

You can write an array reference in statements within the block of statements of a GDML **for**, **modify**, or **store** loop in order to reference either a particular cell or an entire array.

Array Syntax in a Host Language Statement

An array reference in a statement other than an RSE uses this syntax:

```
<array-reference> ::= context-variable.array-name
```

Subscripts are *not* part of this syntax. If used, they must conform to the conventions of the host language. Because of this:

- **Gpre** does *not* notify you of an incorrect reference, so you must reference the array correctly.
- Your compiler may not recognize an incorrect or invalid array reference.

How Gpre Processes the Reference

When an array reference appears in the source code within the block of statements of a GDML **for**, **modify**, or **store** loop (i.e., other than in an RSE), **gpre** ignores all delimiters and subscript information following the name, if any were given. **Gpre** generates a correct reference by array name, but leaves all syntax and parameter checking for you and the host language compiler to handle.

The next two subsections explain how to reference to a particular cell and how to reference an entire array.

Referencing a Particular Cell in an Array

You reference a particular cell in an array by supplying its subscripts, using the conventions of the host language.

This example assigns the value of a particular cell to a numeric host variable, *test_val*. Array subscripts in FORTRAN appear in column major order, are enclosed in one set of parentheses, and separated by commas:

```
test_val = x.wave_height(4,1);
```

In C, array subscripts appear in row major order and are separated from each other, with each subscript separately enclosed in brackets:

```
test_val = x.wave_height[1][4];
```

Gpre does *not* check the subscripts in either of these cases. The host language compiler, however, may find and return errors.

An array reference to a particular cell in an RSE differs from one in a non-RSE statement. **Gpre** checks the subscripting for the one in an RSE, not in a non-RSE. In an

RSE, you satisfy **gpre** by using parentheses or square brackets and separating subscripts by commas. In non-RSE, use the conventions of the host language.

Referencing an Entire Array

Most host languages allow you to write your own functions. Ada lets you define your own operators. In both cases, it is usually possible to supply the operation with just the name of the array you are operating on or manipulating.

For example, suppose you wrote a function in C that prints the contents of an array. The arguments to the function are a control string and an array name. Within the function, you handle how to step through all the elements, obtaining their contents. A call to the function might look like one of the following:

```
array_print("%a",x.wave_height);
array_print("%a",x.wave_height[4][5]);
```

Determining the proper function call to use depends on the parameters your function requires. Because they occur in a non-RSE, **gpre** treats both of these C language references identically by looking only at the array name, and not at subscripts.

Because Ada lets you define your own operators, you could define the "+", for example, to add arrays using a statement such as:

```
array2 = array1 + x.wave_height
```

In these cases, the reference to the array *wave_height* might require the use of trailing brackets or parentheses delimiters, subscript ranges, or as shown, simply the name without subscripts. **Gpre** would correctly generate a reference to the array, but be sure the reference meets the requirements for the operator's arguments.

The function calls above include the context variable *x* because they can be used *only* within the context of a GDML **for**, **modify**, or **store** loop. For example:

```
for x in tidal_study
  printf("Sample %s",x.sample_id);
  array_print("%a",x.wave_height);
end_for
```

Accessing an Array from Multiple Languages

Because you define an array in the database, you can access it with any language that InterBase supports. When you access the array with only one language, you define its base and subscript ranges to match the array-handling conventions of your host language.

For example, C uses zero-based array references, COBOL uses one-based. If you intend to access the array with C, you define it to be zero-based. If you intend to access the array with COBOL, you define it to be one-based.

Similarly, you define subscripts your host language allows when handling arrays. Some host languages allow only positive integers, others allow zero and positive integers, and others allow negative and positive integers.

At times, however, you may need to access an array with a different host language than the one you intended when you defined the array. In such a situation, you may find the array's defined base or subscripts are invalid in the new language.

For example, if you defined the array to be zero-based for use with C, the first element's subscript is zero. Suppose you now need to access that array with a COBOL program. COBOL does not permit an array reference to an element having a subscript of zero. Although **gpre** will accept the reference, your COBOL compiler rejects it. Your COBOL program is not able to access the first array element.

Figure 7-6 illustrates how C and COBOL "see" the two-dimensional array field CONROL_SETTINGS defined to **gdef** by this statement:

```
% define field control_settings short (0:6,0:1)
```

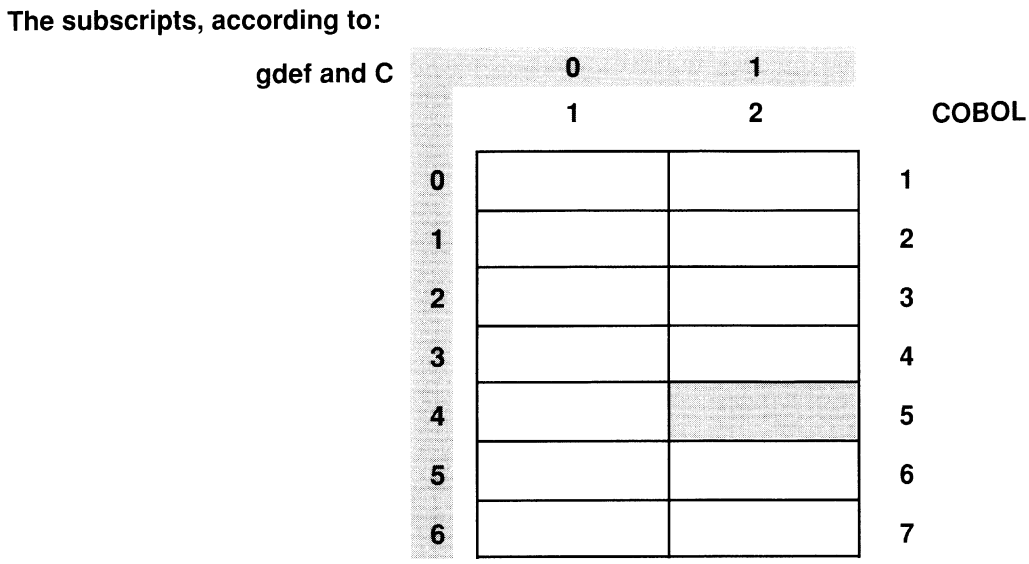

Figure 7-6. CONTROL_SETTINGS Array as seen by C and COBOL

You are not stuck without a solution, however. **Gpre** detects this problem and adjusts for it. **Gpre** normalizes the subscripts of an array reference from the host language to the **gdef** definition. For the situation shown in the diagram, **gpre** adjusts a COBOL reference (one-based) to the **gdef** array definition (zero-based). Now the COBOL reference of *x.control_settings (5,2)* returns the value from the shaded cell.

When you define an array, consider the possibility of encountering such situations. You might be able to specify the base and subscripts to avoid a conflict.

Array Reference Errors

Be sure to check the array definition before you write any references to it. Then, to avoid array reference errors:

- Use the correct number of subscripts.
- Use the correct subscript delimiters.
- Use integer constants or use variables, field references, or compiler manifest constants that will resolve to integer subscripts within the defined range.
- In an RSE, use the RSE syntax.
- In a host language statement, use the host language's array reference conventions.
- Match datatypes.

When **gpre** precompiles your source code, it reads the array's definition from the database. Currently, **gpre** only checks that there are the same number of subscripts in an RSE array reference as there are dimensions in the array definition. If not, it generates an error message.

Gpre does not check array references in host language statements. It accepts an array reference your compiler rejects due to incorrect use of host language conventions, subscripts out of defined bounds, incorrect number of subscripts, and mismatched datatypes.

Be careful when referencing subscripts with other than integer constants (host language variables, database field references, or compiler's manifest constants). Out-of-bound subscript values or mismatched datatypes may not be detected until runtime. Pay attention to values that could be present at runtime.

Array Examples

The following two examples assume that the array *high_low_temperature* has been defined through **gdef** to the database with the following statement:

```
% define field high_low_temperature short (12,2)
```

The two-dimensional array named *high_low_temperature* of type "short" contains 12 elements in the first dimension, with subscripts ranging from the default of 1 to the specified 12. There are 2 elements in the second dimension, with subscripts ranging from the default of 1 to the specified 2.

C Example

```
DATABASE atlas = FILENAME "atlas.gdb";
#include <stdio.h>
main ()
{
int i;
static char * month[12]={"Jan","Feb","Mar","Apr","May",
           "Jun","Jul","Aug","Sep","Oct","Nov","Dec"};
READY atlas;
START_TRANSACTION;
printf ("City   State   Month   Low Temp   High Temp\n");

FOR X IN CITIES WITH X.POPULATION > 50000
      {
      printf ("%s  %s", x.city, x.state);
      for (i =0; i < 12; i++)
          printf ("%s  %d  %d\n",
              month[i],
              x.high_low_temperature[i][0],
              x.high_low_temperature[i][1]);
     printf ("\n\n");
      }
END_FOR;

COMMIT;
FINISH atlas;
}
```

Observations on the C Example

The host language array references in the middle of the example:

```
for (i =0; i < 12; i++)
  print ("%s  %d  %d\n",
      month[i],
      x.high_low_temperature[i][0],
      x.high_low_temperature[i][1]);
```

C permits only *zero-based* array subscript references. To make the proper array references in C, the variable *i* (the *for* loop counter), is initialized to zero and counts to eleven, for a total of 12 elements. Similarly, the second dimension explicitly references two elements, using zero and one. **Gpre** adjusts the references to retrieve values from the first and second cell in the second dimension.

You are responsible for referencing the array correctly for the host language. **Gpre** can *not* detect subscript errors in host language array references.

Pascal Example

```
program array_example (input, output);
DATABASE atlas = FILENAME "atlas.gdb";
var
  month : array [1..12] of char :=
              [ "Jan", "Feb", "Mar", "Apr",
              "May", "Jun", "Jul", "Aug",
              "Sep", "Oct", "Nov", "Dec"];
  i : integer;

begin
READY atlas;
START_TRANSACTION;
writeln ("City   State   Month   Low Temp   High Temp");

FOR X IN CITIES WITH POPULATION > 50000
  begin
  writeln (X.CITY, X.STATE);
  for i := 1 to 12
      writeln (month[i],
          X.HIGH_LOW_TEMPERATURE[i,1],
          X.HIGH_LOW_TEMPERATURE[i,2]);

      writeln;
```

```
  end;
END_FOR;

COMMIT;
FINISH atlas;
end.
```

Observations on the Pascal Example

The host language references in the middle of the example:

```
for i := 1 to 12
  writeln (month[i],
      X.HIGH_LOW_TEMPERATURE[i,1],
      X.HIGH_LOW_TEMPERATURE[i,2]);
```

Pascal permits *one-based* array subscript references. Given the array was defined to the database through **gdef** as a one-based array, no adjustments needed to be made in the code in order to make a proper reference in Pascal.

For More Information

For more information on **gpre**, see the Chapter 18, *Preprocessing Your Program*.

For more information on defining array fields, see the chapter on defining fields in the *Data Definition Guide*.

For more information on the statements used with arrays, see the entries in the *Programmer's Reference* for:

- **for** loop
- **modify** statement
- **store** statement

Chapter 8
Using Blob Fields

This chapter discusses:

- GDML blob statements
- Blob library routines
- **Gds** calls for handling blobs
- Blob filters

Overview

The *basic large object* or blob datatype is available only with InterBase and other DSRI-compatible software. Fields defined as containing blob data do not have their structure specified to the database system as do fields containing character or numeric data. The only structure associated with them is the *segment*, a logical construct that is the unit of blob data manipulation. Programs accessing blob fields read or write successive segments of the blob in much the same way as you read successive lines of a text file.

Blobs are best suited for the storage of:

- Unformatted data, such as:
  - Text
  - Images
  - Digitized data
  - Telemetry
  - CAD drawings
- Any other entity that does not lend itself to storage as longwords or strings.

The blob datatype provides unstructured data with all of the advantages of a database management system, including:

- Full transaction control
- Maintenance by the same utilities as more structured data
- Manipulation with high-level user interfaces

With blobs, you can keep unstructured entities right in your database. You do not have to store pointers to non-database files, nor do you have to make sure the database and its attendant files remain synchronized.

Accessing Blobs

GDML supports several ways of accessing blob data. You can use:

- **For** loops for reading blobs
- Statements similar to those used for file processing for both reading and writing blobs
- A library of routines supporting blob and/or file interchange and stream-like processing
- Call interface routines
- Blob filters

The following sections discuss each of these approaches in more detail.

Note

Because InterBase does not interpret the contents of the blob, values in blobs cannot be used in most record selection expressions or **over** clauses, nor can they be indexed. Furthermore, the contents of a blob field are not translated when they are transferred between different machine or operating system types. **Gpre** does support the use of blobs with the **containing** or **missing** operators in record selection expressions.

Using For Loops

The **for** loop is the recommended way to access both structured data and blobs. You declare a context variable for each relation the **for** loop references. You then use the context variable to qualify individual field references. For example:

```
for tour in tourism sorted by tour.state
  printf ("%s %s %s\n", tour.city, tour.state, tour.zip);
end_for;
```

This **for** statement displays the values of several fields for each record in the TOURISM relation.

If you want to access a blob data field along with the other fields, add an inner **for** loop with a display statement for the blob field:

```
/* program test */

#include <stdio.h>

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

```
main()
{
for tour in tourism sorted by tour.state
  printf ("%s %s %s\n", tour.city, tour.state, tour.zip);
  printf ("\n");
  for o in tour.office
      o.segment[o.length] = '\0';
      printf ("%s", o.segment);
  end_for;
  printf ("\n");
end_for;
}
```

In the program above, the outer **for** loop:

- Creates the record stream
- Displays data from structured fields

The inner **for** loop:

- Declares a *blob variable*, o, for the blob field.

  InterBase associates the blob variable with successive segments of the blob
- Cycles through the blob segments
- Displays the segments one at a time.

The access method sets O.LENGTH to the length of the returned segment. O.SEGMENT refers to the actual segment of the blob. In order for it to print as a C language string it, the blob segment must be null terminated. The statement *o.segment[o.length] = '\0'* accomplishes this. You do not need this statement in languages that use counted strings.

Writing a blob is not that different from writing any other field. It does involve some additional structures within the **store** statement.

The following section discusses reading and writing blob fields.

Processing Blob Fields

Because they function as repositories of large clumps of data, blobs can be thought of as sequential or stream files. Therefore, GDML also supports several statements that let you process blob fields in a file processing sort of way:

- **open_blob** opens the blob field for retrieval.
- **get_segment** gets the next segment of a blob. Before you read a segment, you must open the blob with an **open_blob** statement.
- **create_blob** opens a blob field for writing.
- **put_segment** writes a segment of a blob.Before you write to a blob, you must open it with a **create_blob** statement.
- **close_blob** closes the blob after reading or writing to it.

Reading a Blob Field

You can use the statements listed above in conjunction with a **for** statement to loop through qualifying records and read blob fields in each record:

```
/* program test */

#include <stdio.h>

database db = filename 'atlas.gdb';

main()
{
for tour in tourism sorted by tour.state
  printf ("%s %s %s\n", tour.city, tour.state, tour.zip);
  open_blob b in tour.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;
      }
  close_blob b;
  printf ("\n");
end_for;
}
```

In the preceding program:

- The **for** statement creates the record stream.
- The **printf** statement displays field values from records in that stream.
- The **open_blob** statement associates a blob variable with the OFFICE field.

  This is like an open file statement associating a file variable with a file name. InterBase uses the blob variable to qualify each segment of the blob field.
- The **get_segment** statement retrieves each blob segment.
- The **close_blob** statement closes the blob field.

Writing to a Blob Field

You use the **create_blob** and **put_segment** statements to write to a blob field.

In the following C example, the **create_blob** statement creates the blob field, while the **put_segment** writes to the field. The **store** statement includes several assignments of conventional fields and also writes user input to the GUIDEBOOK field:

```
/* program store_tour */

database db = filename 'atlas.gdb';

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

main()
{
int i;

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 blurb one line at a time\n");
  printf (" A line containing '-30-' ends input\n");
  gets (b.segment);
  while (strcmp (b.segment,"-30-")) {
         for (i=strlen(b.segment)-1;i>=0;i--) {
             if (b.segment[i] != ' ')
```

```
                    break;
                }
            i++;
            b.segment[i] = '\n';
            b.length = i;
            put_segment b;
            gets (b.segment);
        }
    close_blob b;
end_store;
rollback;
finish;
}
```

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

```
DATABASE atlas = FILENAME "atlas.gdb";
DATABASE guide = FILENAME "coastal_guide.gdb";

main()

{
READY atlas;
READY guide;
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;
    }
```

The outer **store** statement assigns field values to the appropriate structured fields. The **for_blob** statement includes a **put_segment** statement to write to the blob field GUIDEBOOK.

Using Blob Library Routines

The GDML library includes blob routines that provide enhanced support for the following blob functions:

- The interchange of data between blob fields and operating system files
- Blob field access similar to stream file access

Many of these routines call for the transaction_handle. If you are using the default transaction, the transaction_handle is **gds_$trans**.

The following sections discuss blob routines.

Moving Blob Data Between Files Systems

The concept of a blob is new enough that most applications do not have data stored in blob fields, but instead in operating system files. To move data currently stored in operating system files into a blob field in a database, the GDML library provides routines that move data between operating system files and blob fields. You use these routines within host language programs. Additionally, the GDML library provides routines that give you byte-by-byte access to blobs for C programs. See the following section for the C specific blob library routines.

The GDML library routines for moving blob data:

- Load the contents of a file into a blob
- Dump a blob into a file
- Read a blob
- Write a blob

The blob routines you can use only in C programs:

- Open a blob stream
- Return a byte from an open blob stream
- Write a byte to an open blob stream
- Close a blob stream

The following sections give an explanation of these routines. Each section also gives the syntax for the routine. The syntax you use in C programs is different than the syntax you use in other languages.

Loading a Blob Field

The **BLOB_load** routine loads the contents of an operating system file into a blob field.

C syntax:

```
BLOB_load (&blob-field, database-handle,
transaction-handle, file-name)
```

Other languages:

```
BLOB_load (blob-field, database-handle,
transaction-handle, file-name, file-name-length)
```

Dumping a Blob

The **BLOB_dump** routine writes the contents of a blob into an operating system file.

C syntax:

```
BLOB_dump (&blob-field, database-handle,
transaction-handle, file-name)
```

Other languages:

```
BLOB_dump (blob-field, database-handle,
transaction-handle, file-name, file-name-length)
```

Reading a Blob

The **BLOB_display** routine reads a blob field, dumps it to a file, and calls the editor with the contents of the blob. On Apollo systems, it creates a window. For other systems, pass a null string for the *window-name* and 0 for the length.

C syntax:

```
BLOB_display (&blob-field, database-handle,
transaction-handle, window-name)
```

Other languages:

```
BLOB_display (blob-field, database-handle,
transaction-handle, window-name, window-name-
length)
```

Writing a Blob

You write a blob in a **store** or **modify** statement. The **BLOB_edit** routine reads a blob field, dumps it to a file, calls the editor with the contents of the blob. If you are using a **store** statement, this is an empty blob since it is new blob. It is a non-empty blob when you use a **modify** statement. The **BLOB_edit** routine dumps the data back into the blob when you exit from the editor. On Apollo systems, it creates a window. For other systems, pass a null string for the *window-name* and 0 for the length.

C syntax:

```
BLOB_edit (&blob-field, database-handle,
transaction-handle, window-name)
```

Other languages:

```
BLOB_edit (blob-field, database-handle,
transaction-handle, window-name, window-name-
length)
```

Blob Routines for C

The GDML library contains routines that provide byte-by-byte access to blobs for C programs. With these routines you can:

- Open a blob stream
- Return a byte from an open blob stream
- Write a byte to an open blob
- Close a blob stream

The following sections give an explanation of these routines. Each section also gives the syntax for the routine You can use this syntax only with C programs .

Opening a Blob Stream

The **bopen** routine is similar to the C **fopen** statement. The *access* parameter can be **r** for read or **w** for write. This routine returns the *bstream* pointer referenced in other routines.

Syntax: ***bstream = Bopen (&***blob-fields, database-handle, transaction-handle, "access"***)**

Returning a Byte

The **getb** routine is similar to C's **getc** statement.

Syntax: **getb (**bstream**)**

Writing a Byte

The **putb** routine is similar to C's **putc** statement.

Syntax: **putb (***bstream***)**

Closing a Blob Stream

The **BLOB_close** routine closes a blob stream.

Syntax: **BLOB_close** (*bstream*)

Blob Routine Example

The following C program uses the blob routines that are specific to C:

```
DATABASE atlas = FILENAME "atlas.gdb";

#include <stdio.h>

#define FALSE 0
#define TRUE 1
#define GET_STRING(p,b) get_string (p, b, sizeof (b))

main ()
{
char filename[40], choice[4], *state;
BSTREAM *tour;
short c;

READY;
START_TRANSACTION;

FOR s IN STATES CROSS t IN TOURISM OVER state
  {
  printf ("Here's the information on  %s.\n\n",s.state_name);
  if (tour = Bopen (&t.guidebook, atlas, gds_$trans, "r"))
      while ((c = getb (tour)) != EOF)
          putchar (c);
      BLOB_close (tour);
  GET_STRING
      ("\nIs this information up to date? (Y/N) ", choice);
  if (*choice  != 'y' && *choice != 'Y')
      MODIFY t USING
          {
          printf ("Enter new information from standard
  input.  ");
           printf ("Terminate with <EOF>.\n");
           if (tour = Bopen (&t.guidebook, atlas, gds_$trans,"w"))
      while ((c = getchar ()) != EOF)
                  putb (c, tour);
          BLOB_close (tour);
            }
      END_MODIFY;
  }
```

```
    END_FOR;
COMMIT gds_$trans;
}
```

Using Gds Blob Calls

The low-level call interface to the InterBase access method, described in the *OSRI Guide,* is used by all system components. For example, when you preprocess a program, **gpre** translates the high-level GDML statements into **gds** calls. In some cases, the difference between the high-level GDML statement and the corresponding **gds** call is minimal. The GDML statements that process blobs and the **gds** blob calls are one such case.

For more information on call interface routines, see Chapter 11, *Using OSRI Calls.*

You can use the **gds** blob calls in your GDML program, or you can use them if you write the special-purpose blob routines discussed in the preceding section. Each **gds** blob call is described in the following sections. The **gds** blob calls are:

- **gds_$cancel_blob** which releases internal storage
- **gds_$close_blob** which releases system resources
- **gds_$create_blob** which creates a context for blob storage and opens a blob for write access
- **gds_$get_segment** which reads a blob segment
- **gds_$open_blob** which prepares a blob for retrieval
- **gds_$put_segment** which writes the next blob segment

Each description also contains the syntax for that blob call. All parameters to **gds** calls are passed by reference. The datatype and usage of each parameter is listed in each syntax description. The datatype can be:

Datatype	Description
Byte	An 8-bit quantity
Ubyte	An unsigned byte
Short	A 16-bit quantity (word)
Ushort	An unsigned word
Long	A 32-bit quantity (longword)
Ulong	An unsigned longword
Quad	A 64-bit quantity (quadword)
Uquad	An unsigned quadword
Vector_char	A vector of characters
Vector_long	A vector of 20 longwords
Vector_byte	A vector of bytes

Datatype	Description
Unspec	The address of a buffer whose description is established elsewhere, or a binary value the size of which is given elsewhere

The usage parameter specifies how the argument is to be used. The three parameters uses are:

- In, which means the parameter is input only and not altered by InterBase
- Out, which means the parameter is output only and not read by InterBase. InterBase supplies a value for the parameter.
- Inout, which means InterBase reads and writes the paramters. Ordinarily, the input value is either zero or a value established in another call.

Blob calls have additional parameters you need in order to handle blobs. These parameters are listed below: 7z

- *status_vector*, a vector of 20 longwords used to return error messages to the program. If you pass zero as the address of the status vector and encounter an error, InterBase writes the message(s) to standard error and aborts your program. See Chapter 4 for more information on the status vector.
- *blob_handle*, an identifier returned by InterBase when a blob is created or an existing blob opened. The handle is used for all subsequent reference inside the transaction that creates it or opens it.
- *blob_id*, an internal identifier returned by InterBase when a blob is created or opened. This identifier is the blob field's name.
- *dbhandle*, an identifier for the database in which the blob is manipulated.
- *transaction_handle*, an identifier for the transaction in which the blob is manipulated.

Releasing Internal Storage

The **gds_$cancel_blob** routine releases internal storage used by a discarded blob and sets the handle to zero.

Syntax: `status = gds_$cancel_blob (status_vector, blob_handle)`

Arguments:

Parameter	Datatype	In/Out
status_vector	vector_long	out
blob_handle	ulong	in/out

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. Furthermore, the han-

dle is not null, which causes problems for anything that runs into it. Use this call whenever you want to ensure that the blob handle is set to null. If the handle is already null, the routine returns with a successful status.

Releasing System Resources

The **gds_$close_blob** routine releases system resources associated with blob update or retrieval. You should call **gds_$close_blob** as soon as you finish reading or writing a blob. The blob handle must be non-zero; this routine sets the value of the handle to zero.

Syntax:

```
status = gds_$close_blob (status_vector,
blob_handle)
```

Arguments:

Parameter	Datatype	In/Out
status_vector	vector_long	out
blob_handle	ulong	in/out

If you fail to close a blob you created, you may lose some of the data. Because the remote interface buffers segment transfer between participating nodes, it may truncate the last segment you write unless you explicitly close the blob.

Creating a Context and Opening a Blob

The **gds_$create_blob** routine creates the context for storing a blob and opens the blob for write access.

Syntax:

```
status = gds_$create_blob (status_vector,
db_handle,transaction_handle,
blob_handle,blob_id)
```

Arguments:

Parameter	Datatype	In/Out
status_vector	vector_long	out
db_handle	ulong	in
transaction_handle	ulong	in
blob_handle	ulong	in/out
blob_id	uquad	out

A successful call to **gds_$create_blob** creates the environment for storing a blob. However, InterBase does not store the blob until a **blr_assignment** statement assigns it to a relation.

InterBase also uses *blob_id* when it opens the blob with a call to **gds_$open_blob**. However, the value of *blob_id* when you create the blob is *not* the same as when you

open the blob. When you create the blob, it is essentially an "orphan" until a **blr_assignment** statement assigns the value of *blob_id* to the blob field in a relation.

InterBase automatically changes the value of *blob_id* at the time of assignment. Once you assign *blob_id* to its relation, the creation value disappears forever. Therefore, if you open the newly created blob later, you must read *blob_id* from the record. If you try to save the old *blob_id* and re-use it, InterBase returns an error.

Reading a Blob Segment

The **gds_$get_segment** routine reads a portion of a blob field.

Syntax:

```
status = gds_$get_segment (status_vector,
blob_handle, actual_segment_length,
segment_buffer_length, segment_buffer_address)
```

Arguments:

Parameter	Datatype	In/Out
status_vector	long	out
blob_handle	ulong	in
actual_seg_length	ushort	in
seg_buffer_length	ushort	in
seg_buffer_address	unspec	in

This routine is the *read* call for blob manipulation. Before you can read a blob, you must open it with a call to **gds_$open_blob** or an equivalent routine. What a call to **gds_$get_segment** does depends on previous blob calls. If the last call that used *blob_handle* was:

- **gds_$get_segment**, it reads the next segment
- **gds_$open_blob**, it reads the first segment.

Preparing a Blob for Retrieval

The **gds_$open_blob** routine prepares a blob for retrieval. Call **gds_$cancel_blob** before you open a blob to ensure that the handle is null and that unused resources have been released.

Syntax:

```
status = gds_$open_blob
(status_vector,db_handle, transaction_handle,
blob_handle, blob_id)
```

Arguments:

Parameter	Datatype	In/Out
status_vector	long	out
db_handle	ulong	in
transaction_handle	ulong	in

blob_handle	ulong	in/out
blob_id	uquad	in

Writing a Blob Segment

The **gds_$put_segment** routine writes the next segment of a blob. You cannot read segments written with calls to **gds_$put_segment** until you close the blob with a call to **gds_$close_blob** and then reopen the blob with a call to **gds_$open_blob**.

Syntax:

```
status = gds_$put_segment (status_vector,
blob_handle, segment_buffer_length,
segment_buffer_address
```

Arguments:

Parameter	Datatype	In/Out
status_vector	long	out
blob_handle	ulong	in
actual_seg_length	ushort	in
seg_buffer_length	ushort	in
seg_buffer_address	unspec	in

The following section shows how to use blob calls.

Blob Call Example

The following sequence copies one blob's data to another blob:

1. Open the new blob (the target blob) for storage by calling **gds_$create_blob**. The following call creates the context for storing a blob and opens the blob for write access.

```
if (gds_$create_blob (status_vector,
      to_dbb_handle,
      to_dbb_transaction,
      to_blob,
      &blob_id))
   ERRQ_database_error (to_dbb_handle, status_vector);
```

   The ERRQ_database_error routine prints the database status and performs reasonable cleanup.

2. Open the existing blob (the source blob) for read with a call to **gds_$open_blob**.

```
if (gds_$open_blob (status_vector,
      from_dbb_handle,
      from_dbb_transaction,
      from_blob,
      &blob_id))
   ERRQ_database_error (from_dbb_handle, status_vector);
```

   If you read input from a file, you would open the file here rather than open a blob.

3. Get segments from the source blob by calling **gds_$get_segment** and write them to the target blob by calling **gds_$put_segment**. The following sequence of calls reads through all segments of the source blob and writes them to the target blob:

```
while (!gds_$get_segment (status_vector,
         from_blob,
         length,
         sizeof (sqffer),
         buffer))
   if (gds_$put_segment (status_vector,
      to_blob,
      length,
      buffer))
   ERRQ_database_error (to_dbb, status_vector);
```

   The result of a call to **gds_$get_segment** depends on previous blob calls. The first time through the above loop, the call to **gds_$get_segment** reads the first segment, while subsequent calls get successive segments.

4. Close both blobs with calls to **gds_$close_blob**. This routine releases system resources associated with blob update or retrieval.

```
if (gds_$close_blob (status_vector,
      from_blob))
   ERRQ_database_error (from_dbb, status_vector);

if (gds_$close_blocb (status_vector,
      to_blob))
   ERRQ_database_error (to_dbb, status_vector);
```

   You should call **gds_$close_blob** as soon as you finish reading or writing a blob.

   If you fail to close a blob you created, you may lose some of the data. Because the remote interface buffers segment transfer between participating nodes, it may truncate the last segment you write unless you explicitly close the blob.

5. If you want to read from a blob field you closed, you must re-open it with a call to **gds_$open_blob** (step 2 above) and repeat steps 3 and 4.

The net result of this sequence is two copies of the same blob.

In general, it is good practice to cancel a blob by calling **gds_$cancel_blob** before doing anything with the blob. A successful return from this routine releases internal storage used by a discarded blob and sets the blob handle to null.

Because a call to this routine does *not* produce an error if the handle is null, call this routine before you call either **gds_$open_blob** or **gds_$create_blob**. This practice ensures that the access method cleans up earlier blob operations. If you abort a blob operation or if you do not trust the routine that passed the blob id, you should call **gds_$cancel_blob** before opening or creating a blob.

InterBase also has information calls you can use to check on the status of database entities. For information on the **gds_$blob_info** routine, and other informational **gds** routines, see Chapter 11 of this manual.

For More Information

For a more detailed discussion of the **gds** calls described here, see:

- Chapter 12, *Using OSRI Calls*
- the *OSRI Guide.*

Chapter 9
Using Blob Filters

This chapter discusses how to use blob filters.

Overview

A *blob filter* is a program that converts data stored in blob fields from one blob subtype to another. For example, if you need to manage C language code on several different platforms, you can run into problems. Some C compilers do not accept "$" in filenames, and some operating systems use different file naming conventions. You can resolve this problem by storing generic code formats in blobs and then filtering the code on input and output for the different environments.

Here are some other points about blob filters:

- A blob filter can operate on InterBase subtypes or user-defined subtypes
- The access method invokes a blob filter at the user's request

- You can use either C or Pascal to write a blob filter because they both support
  - — Calling and being called from C
  - — Structures
  - — The use of function pointers

In order to program with blob filters, you should be a high level programmer familiar with blobs and the GDML statements for handling blobs. For further information on blob fields, see the chapter on defining fields in the *Data Definition Guide*.

Programming with Blob Filters

There are six steps to programming with blob filters:

1. Decide which filters you need to write.
2. Write the filters in a host language.
3. Define the filters with **gdef**.
4. Build a shared filter library.
5. Make the filter library available.
6. Write an application which requests filtering.

Each of thse steps is discussed in the following sections.

Deciding Which Filters You Need

If you have identified a situation in which you need blob filters, your first step is to determine how many filters you need to handle the situation. For example, if you use a filter to convert data in a blob from *nroff* markup language to formatted output, you need to write one filter that invokes *nroff* when you retrieve the data in the blob.

However, when you have generic C code that you filter for different platforms on input and output, you need two filters for each platform. For example, if you filter the generic C for VMS, OS/2 and HP, you need the following six filters:

Output Filters	Input Filters	Result
Generic C to VMS	VMS to generic C	Converts include file pathnames
Generic C to OS/2	OS/2 to generic C	Converts include file pathnames
Generic C to HP	HP to generic C	Converts **gds_$** to **gds_ _**

Writing and Compiling a Blob Filter

When you use a blob filter, you are imposing a user-defined action on the DBMS. And, although a blob filter takes control of the action, it also returns the action to the DBMS. When this happens, you must let the DBMS know what action has just occurred. Because of this interaction, you need to create a line of communication between your program and the access method. In order to set up this communication, a blob filter must include two things:

- A control structure that functions as a messenger between the access method and the blob filter
- Program constants called *actions* that regulate the action between the DBMS and the blob filter.

The Control Structure

The filter code below is the control structure that must be included in every blob filter. You use this structure to pass around the buffers and information about what's taking place that your program needs :

```
typedef struct ctl {
    int(*ctl_source)();              /* Source filter */
    struct ctl *ctl_source_handle;   /* Argument to pass to
                                        source filter */
```

```
    short ctl_to_sub_type;          /* Target type */
    short ctl_from_sub_type;        /* Source type */
    SHORT ctl_buffer_length;        /* Length of buffer */
    SHORT ctl_segment_length;       /* Length of current
                                       segment */
    SHORT ctl_bpb_length;           /* Length of blob
                                       parameter block */
    char *ctl_bpb;                  /* Address of blob
                                      parameter block */
    CHAR *ctl_buffer;               /* Address of segment
                                       buffer */
    long ctl_max_segment;           /* Length of longest
                                       segment */
    long ctl_number_segments;       /* Total number of
                                       segments */
    long ctl_total_length;          /* Total length of blob */
    long *ctl_status;               /* Address of status
                                       vector */
    long ctl_data [8];          /* Application specific data */
} *CTL;
```

Control Structure Usage Parameters

Each field of the control structure has a usage parameter which specifies how an argument is to be used:

- IN, which means the parameter is input only and not altered by the access method.
- OUT, which means the parameter is output only and not read by the access method. The access method supplies a value for the parameter.
- IN/OUT, which means the access method reads and writes the paramters. Ordinarily, the input value is either zero or a value established in another call.

Control Structure Fields

The following list explains the fields of the control structure and specifies each field's usage parameter:

- **int (*ctl_source)()** is a pointer to the entry point of the next filter in line. When called, this function takes two arguments:
  - — the action to taken
  - — the CTL structure contained in the ctl_source_handle member of the CTL structure.

Its parameter is IN.

In the nroff_filter example in the *Blob Filter Example* section of this chapter, this function is repeatedly invoked with an ACTION_get_segment during ACTION_-open. The function gets each segment of the blob to build a file containing the entire blob. Once this file is built, the blob can be run through **nroff**.

- **struct ctl *ctl_source_handle** is a control structure. The function pointed at by ctl_source uses it as an argument. The ctl_status, ctl_buffer and ctl_buffer_length fields of this structure should be set prior to the call to the ctl_source function. Its paramter is IN/OUT.

  For an example, see the *caller()* function section in the *Blob Filter Example* section at the end of this chapter.

- **unsigned short ctl_to_sub_type** and **unsigned short clt_from_sub_type** are fields you use when a blob filter handles more than one blob subtype conversion. These fields allow the filter to select the appropriate action based on the actual conversion requested when you invoke the filter. The parameters for both fields are IN/OUT.

- **char * ctl_buffer** points to a buffer before an ACTION_get_segment. Its parameter is IN/OUT.

- **unsigned short ct_buffer_length** contains the number of bytes in the buffer referred to ty ctl_buffer, or the segment lenth of the blob. Its paramter is IN/OUT.

- **unsigned short ctl_segment_length** is the length of the segment currently being referenced. The access method automatically sets this field when an ACTION_put_segment is performed. In this case, the segment length is set to the length in bytes of the segment just returned. The blob filter should set this field before doing an ACTION_put_segment. On an ACTION_put_segment, this fields should be set to the length in bytes of the segment to be put. Its parameter is In/OUT.

- **char * ctl_bpb** and **short ctl_bpb_length** are reserved for future enhancements. The bpb fields point to a blob parameter block. For more information on the blob parameter block, see Chapter 12, *Using OSRI Calls*.

- **long ctl_max_segment_length**
  long ctl_number_segments
  long ctl_total_length
  These three fields are used to hold statistics about the blob. These statistics are returned in response to a **gds_$get _info** or **gds_$blob_size** call. You should make sure to keep these statistics up to date. Their parameters are IN/OUT. Set the fields as follows:

  — ctl_max_segment_length: length in bytes of the longest segment in the blob

  — ctl_number_segments: total number of segments in the blob

— ctl_total_length: total length in bytes of the blob

For an example of how to set up these fields, see the *set_statistics ()* routine in the *Blob Filter Example* section at the end of this chapter.

- **long * ctl_status** is a field the blob filter uses for returning status. A blob filter can return a status value or it can create and fill a standard InterBase status vector. For example, if the blob filter needs to return an arbitrary string to a user, it can return the following InterBase status vector:

Word	Content	Explanation
0	1	Major code follows
1	**gds_$random**	InterBase major code
2	4	String follows
3	(pointer to string)	Pointer to random string

For more information on the status vector, see the Appendix of the *Programmer's Reference.*

- **long ctl_data[8]** is an array of eight longwords the blob filter uses to store any data it needs. Data stored here is available to the blob fitler as long as the blob is open. The example at the end of the chapter, nroff_filter, uses the field to hold the file pointer to the file containing the nroff output. The array's parameter is IN/OUT.

The ACTION Macros

You should include the following five program constants in your program. These macros are the possible actions a blob filter can be called with. These actions closely correspond to the blob library calls **open_blob**, **get_segment**, **close_blob**, **create_blob**, and **put_segment**.

- #define ACTION_open0
- #define ACTION_get_segment1
- #define ACTION_close2
- #define ACTION_create3
- #define ACTION_put_segment4

The following list describes each of the ACTION macros and their function in a filter:

- ACTION_open. Use this action if the access method has already opened the blob before you invoke the filter. Unless you need to take special action, the filter can return without taking additional action. The sample at the end of the chapter shows the type of special action you might need to take. The filter reads in the entire blob and dumps it to a file when it is invoked with an ACION_open.
- ACTION_get_segment and ACTION_put_segment. With both these actions, you invoke the filter without the access method taking any action. Instead, the filter calls the ctl_source function to get or put a segment.

  In the nroff_filter example, the filter retrieves the necessary segments during the ACTION_open, so it does not call ctl_source during the ACTION_get_segment.
- ACTION_close. The access method closes the blob after the blob filter returns from being invoked with ACTION_close. If no other action is need, the filter can return immediately.
- ACTION_create. If the action requested of the blob filter is an ACTION_create, the access method creates the blob before invoking the blob filter. If no further action is necesary, the blob filter can return immediately.

Nroff Filter Example

When you write a blob filter, you have to provide appropriate code for these actions. The program fragment following is part of the complete example at the end of the chapter. The fragment shows how to use these program constants.

```
int nroff_filter (action, control)
unsigned short action;
CTL control;


/*************************************
 *
 *        n r o f f _  f i l t e r
 *                                                  Entry point of filter
 *************************************
 *
 * Functional description
 *        Translate a blob from nroff markup
 *        to nroff output.  Read the blob
 *        into a file and process it on open,
 *        then read it back line by line in
 *        the get_segment loop.
 *************************************/
```

```
int         status;
FILE         *text_file;
char         *out_buffer;


switch (action)
    {
    case ACTION_open:
        if (status = dump_text (action, control))
            return status;
        system ("nroff foo.nr >foo.txt");


          set_statistics("foo.txt", control);

      /* set up stats in ctl struct */

        break;

    case ACTION_get_segment:
        /* open the file first time through and save the file
           pointer */
        if (!control->ctl_data [0])
            {
            text_file = fopen ("foo.txt", "r");
            control->ctl_data[0] = (long)text_file;
            }

        if (status = read_text (action, control))
            return status;
        break;

    case ACTION_close:
     /* don't need the temp files any more, so clean them up */
        unlink("foo.nr");
         unlink("foo.txt");
  break;

    case ACTION_create:
    case ACTION_put_segment:
        return gds_$uns_ext;
    }
return SUCCESS;
}
```

Dumps entire blob to file and runs it through nroff.

Set up blob statistics for subsequent gds_$blob_infor calls.

Save file pointer in an application specific area

Read foo.txt one line at a time

This filter should not be invoked by a create_blob or put_segment, so just return the unsupported error.

Compiling a Filter

You compile the filter as follows:

- Under Apollo SR9.7, type:

```
cc -c nroff_filter.c
```

- Under Apollo SR10.0, type:

```
cc -c -W0,-pic nroff_filter.c
```

- Under SunOS 4.0, type the following for inclusion in a shareable object library:

```
cc -c -pic nroff_filter.c
```

  Type the following for inclusion in a non-shareable object library in SunOS 4.0:

```
cc -c nroff_filter.c
```

- Under UNIX systems that don't support dynamic libraries, type:

```
cc -c nroff_filter.c
```

- Under VMS, type:

```
cc/gfloat nroff_filter.c
```

These commands create an object module called *nroff_filter.o*, except under VMS, where the output file is called *nroff_filter.obj*.

Defining the Filters to the Database

You define a blob filter to the database by specifying the following information:

- Filter name
- Input blob subtype
- Output blob subtype
- Module name of the library where the filter is stored
- Entry point in the code

For example, suppose you want to filter data from blob subtype -1 to blob subtype 1. The filter you want to use is called *nroff_test_filter* and the filter resides in a library called *filterlib* and has an entry point of *nroff_filter*.

You would specify the following definition to **gdef**:

```
define filter nroff_test_filter
  input_type -1
  output_type 1
  module_name 'FILTERLIB'
  entry_point  'NROFF_FILTER'
  {This filter is used to convert a marked-up nroff file to
   nroff output};
```

You can define only one filter with the same input_type output_type combination.

Creating a Filter Library

You can create a filter library on any platform that InterBase supports. You should create one filter library for each platform on which the database involved resides.

Instructions for creating filter libraries and making them available to InterBase on each supported platform follow.

Creating a Filter Library under Apollo

To create a filter library under Apollo SR9.7 or SR10.0, use the **bind** utility to bind the object you just created into a filter library. When you do this, you must include a **-mark** option for each filter entry point.

For example, to bind *nroff_filter.o* into a library named *filterlib*, type:

% bind nroff_filter.o -bin filterlib -mark nroff_filter

If you want to provide access to the filter library locally, use the **inlib** command to make the library available to the process you are using:

```
% inlib filterlib
```

If you want to access the filter library through the remote server, install the filter library as a global library on each node. This ensures the library will be available to all processes that need to access it.

- To do this under Apollo SR9.7, rebind the **gdslib** to **inlib** the filter library:

```
% cd /interbase/lib
% cp gdslib gdslib.orig
% bind -inlib filterlib -bin gdslib gdslib.orig
```

- To do this under Apollo SR10.0, add the filter library to the */etc/sys.conf* file and perform a soft reboot of the system.

Creating a Filter Library under SunOS 4.0

To create a filter library under SunOS 4.0 by using shareable libraries:

1. Log on to the node where the database resides.
2. Add the name, entry point, and module name of each filter to the table ISC_-FUNCTIONS in the *functions.c* file in the examples directory. Later on, when you define the filter to the database, be sure to specify the module name and entry point exactly as they're specified here.

Note

The filter name you enter in ISC_FUNCTIONS must be unique.

ISC_FUNCTIONS provides a template for you to follow. The first time you open the file, you'll see the following description:

```
typedef struct {
     char         *fn_module;
     char         *fn_entrypoint;
     FUN_PTR      fn_function;
}FN;

static test();

static FN       isc_functions [] = {
     "test_module", "test_function", test,
    0, 0, 0};
```

To fill in the ISC_FUNCTIONS table for the NROFF_FILTER filter, type:

```
extern int nroff_filter();

static FN       isc_functions [] = {
     "test_module", "test_function", test,
     "FILTERLIB", "NROFF_FILTER", nroff_filter,
         0, 0, 0};
```

Note

Do not delete the final line of zeroes. They signal the end of the table.

3. Compile *functions.c*:

```
% cc -c -pic functions.c
```

4. Copy the shareable library */usr/interbase/lib/gdsflib.so.0.0* to your working directory:

```
% cp /usr/interbase/lib/gdsflib.so.0.0 new_gdsflib
```

5. Add the object files *nroff_filter.o* and *functions.o* to the shareable library:

```
% ld -o new_gdsflib -assert pure-text nroff_filter.o
functions.o
```

To make the filter library available under SunOS 4.0 by using shareable libraries:

1. Set up an environment variable to force the use of the new shareable library for testing purposes:

```
% setenv LD_LIBRARY_PATH /absolute directory path of
      new_gdsflib
```

2. Link */usr/interbase/lib/gdslib.so.0.0* to your working directory:

```
% ln -s /usr/interbase/lib/gdslib.so.0.0 libgdslib.so.0.0
```

3. Link *new_gdsflib* to the filename *libgdsflib.so.0.0*. This enables InterBase to find *new_gdsflib* at run-time:

```
% ln -s new_gdsflib libgdsflib.so.0.0
```

4. Test your filters.
5. Copy the new shareable library, *new_gdsflib*, to */usr/interbase/lib* on the node where the database resides:

```
% cp new_gdsflib /usr/interbase/gdsflib.so.0.0
```

To create a filter library under SunOS 4.0 without using shareable libraries, follow the instructions below for creating the filter library under other UNIX platforms.

Creating a Filter Library under other UNIX platforms

To create a filter library under SunOS 3.5, HP-UX, and Ultrix:

1. Log on to the node where the database resides.
2. Add the name, entry point, and module name of each filter to the table ISC_-FUNCTIONS in the *functions.c* file in the examples directory.

   For example, to fill in the ISC_FUNCTIONS table for the filter you coded earlier, type:

```
static FN       isc_functions [] = {
     "test_module", "test_function", test,
     "FILTERLIB", "NROFF_FILTER", nroff_filter,
         0, 0, 0};
```

3. Compile *functions.c* without using the **pic** switch:

```
% cc -c functions.c
```

4. Concatenate *functions.o* with the file that contains the filter code. In this example, the file is called *nroff_test_filter.o*:

```
% ld -r -o filterlib.o functions.o nroff_filter.o
```

5. Copy the InterBase back end to your working directory:

```
% cp /usr/interbase/lib/gds_b.a new_gds_b.a
```

6. Add the filter library to the new back end:

```
% ar rls new_gds_b.a funclib.o
```

7. Reinitialize the symbol table for the archive:

```
% ranlib new_gds_b.a
```

To make the filter library available under these operating systems, you build the filter library into the pipe server by following these steps:

1. Link the InterBase back end to the pipe server:

```
% cc /usr/interbase/lib/gds_pipe.a new_gds_b.a -o gds_pipe
```

   Depending on your platform and the type of filters you have defined, you may also need to link with the math library provided with your platform in the above example.

2. Set up an environment variable to force the use of the new pipe server for testing purposes:

```
% setenv GDS_SERVER /absolute directory path of your
      working directory/gds_pipe
```

3. Test your filters by using a GDML application program. Information on accessing filters is presented later in this chapter.

4. When you are satisfied with how your filters work, save the original version of the pipe server and the original version of *gds_b.a*:

```
% cp /usr/interbase/bin/gds_pipe
      /usr/interbase/bin/gds_pipe.bak

% cp /usr/interbase/lib/gds_b.a /usr/interbase/lib/gds_b.bak
```

5. Replace the original version of the pipe server with the pipe server you created in Step 1:

```
% cp gds_pipe /usr/interbase/bin/gds_pipe
```

6. Copy the new InterBase back end to */usr/interbase/lib*:

```
% cp  new_gds_b.a /usr/interbase/lib/gds_b.a
```

7. Relink any applications that were previously linked against *gds_b.a*.

If you want to be able to access your filters remotely, you must also rebuild the inet server so that it incorporates the new filters. To do this, follow these steps:

1. Link a new *gds_inet_server* using the new *new_gds_b.a* you built above:

```
% cc /usr/interbase/lib/gds_inet_server.a
      new_gds_b.a -o new_gds_inet_server
```

2. Test your filters remotely by using a GDML application program. Information on accessing filters is presented later in this chapter.
3. When you are satisfied with how your filters work, save the original version of the inet server:

```
% cp /usr/interbase/bin/gds_inet_server.a
      /usr/interbase/bin/gds_inet_server.bak
```

4. Replace the original version of the inet server with the inet server you created in step 1:

```
% cp new_gds_inet_server /usr/interbase/bin/gds_inet_server
```

Creating a Filter Library under VMS

To create a filter library under VMS, use a linker options file to make the filter library entry points universal:

```
link/share=filterlib.exe udf,sys$input/opt
  psect_attr = errno, noshr
  psect_attr = stderr, noshr
  psect_attr = stdin, noshr
  psect_attr = stdout, noshr
  psect_attr = sys_nerr, noshr
  psect_attr = vaxc$errno, noshr
  universal = nroff_filter
```

By not using shared writeable psects, you avoid having to use the **install** utility to install the shared filter library. For more information on shareable executables, refer to the VAX Linker documentation.

To make the filter library available under VMS, do any one of the following:

- Copy the shareable executable to the *sys$share* directory:

```
$ cp filterlib.exe sys$share:
```

- Make *sys$share* a list of pathnames that includes the directory that contains the filter library:

```
$ define sys$share $mydisk:[mydir], sys$sysroot:[syslib]
```

- Define a logical name that has the same name as the filter library module:

```
$ define filterlib $mydisk:[mydir]filterlib.exe
```

Accessing a Blob Filter

You can access a blob filter using embedded GDML in a host language program. You can also access a blob filter through **qli**.

To access a filter from a host-language program, include a statement specifying the subtype you are filtering from and the subtype you are filtering to.

The example below shows how to access a filter from a C program:

```
#include "/usr/interbase/include/gds.h"
database db = filename "slides.gdb";
main ()
{
ready;
start_transaction;
for t in text
  printf ("seminar:\t%s\t\ttalk:\t%s\n", t.seminar, t.talk);
  open_blob b in t.text filter from -1 to 1;
  get_segment b;
  while (!gds_$status [1] || gds_$status [1] == gds_$segment)
      {
      b.segment [b.length] = 0;
      printf ("\t%s", b.segment);
      get_segmentb;
      }
  close_blob b;
end_for;
commit;
finish;
}
```

Qli can invoke an existing blob filter. For example, you can define a filter that converts blob data from its defined subtype to text (a pre-defined subtype of 1, representing text). For example, if you ask to view the relation the program above uses, **qli** automatically invokes the nroff filter to filter the data from markup to formatted text (from -1 to 1). The data display as formatted text rather than markup language.

If you ask **qli** to display a blob filed in a **print** or **list** statement, **qli** uses this filter to convert the blob data from its stored type to text.

Blob Filter Example

The following example is a C program showing a blob filter called *nroff_filter*. This filter converts marked up text to readable output.

```
#include <stdio.h>
#include "/usr/interbase/include/gds.h"

#ifndef CHAR
#define CHAR        unsigned char
#endif

#ifndef SHORT
#define SHORT       unsigned short
#endif

static void set_statistics();

typedef struct ctl {
    int(*ctl_source)();              /* Source filter */
    struct ctl*ctl_source_handle;    /* Argument to pass to
                                        source filter */
    short ctl_to_sub_type;           /* Target type */
    short ctl_from_sub_type;         /* Source type */
    SHORT ctl_buffer_length;         /* Length of buffer */
    SHORT ctl_segment_length;        /* Length of current
                                        segment */
    SHORT ctl_bpb_length;            /* Length of blob
                                       parameter block */
    char *ctl_bpb;                   /* Address of blob
                                        parameter block */
    CHAR *ctl_buffer;                /* Address of segment
                                        buffer */
    long ctl_max_segment;            /* Length of longest
                                        segment */
    long ctl_number_segments;        /* Total number of
                                       segments */
    long ctl_total_length;           /* Total length of blob */
    long *ctl_status;                /* Address of status
                                       vector */
    long ctl_data [8];          /* Application specific data */
} *CTL;

#define ACTION_open              0
```

```
#define ACTION_get_segment        1
#define ACTION_close              2
#define ACTION_create             3
#define ACTION_put_segment        4

#define SUCCESS                   0
#define FAILURE                   1

#define BUFFER_LENGTH           512


int         nroff_filter (action, control)
    SHORT         action;
    CTL         control;
{
/**************************************
 *
 *        n r o f f _  f i l t e r
 *
 **************************************
 *
 * Functional description
 *        Translate a blob from nroff markup
 *        to nroff output.  Read the blob
 *        into a file and process it on open,
 *        then read it back line by line in
 *        the get_segment loop.
 *
 **************************************/
int         status;
FILE         *text_file;
char         *out_buffer;

switch (action)
    {
    case ACTION_open:
        if (status = dump_text (action, control))
            return status;
        system ("nroff foo.nr >foo.txt");
          set_statistics("foo.txt", control);

/* set up stats in ctl struct */
```

```
#ifdef DEBUG
        printf("max_seg_length: %d, num_segs: %d,
                 length: %d\n",
        control->ctl_max_segment, control->ctl_number_segments,
                 control->ctl_total_length);
#endif
        break;

    case ACTION_get_segment:
        /* open the file first time through and save the file
pointer */
        if (!control->ctl_data [0])
            {
            text_file = fopen ("foo.txt", "r");
            control->ctl_data[0] = (long)text_file;
            }
        if (status = read_text (action, control))
            return status;
        break;

    case ACTION_close:
      /* don't need temp files any more, so clean them up */
        unlink("foo.nr");
         unlink("foo.txt");
         break;

    case ACTION_create:
    case ACTION_put_segment:
        return gds_$uns_ext;
    }
return SUCCESS;
}

static int caller (action, control, buffer_length, buffer,
return_length)
    SHORT        action;
    CTL                control;
    SHORT        buffer_length;
    CHAR        *buffer;
    SHORT        *return_length;
{
/************************************
 *
 *        c a l l e r
```

```
 *
 ************************************
 *
 * Functional description
 *        Call next source filter.  This
 *        is a useful service routine for
 *        all blob filters.
 *
 ************************************/
int         status;
CTL         source;

source = control->ctl_source_handle;
source->ctl_status = control->ctl_status;
source->ctl_buffer = buffer;
source->ctl_buffer_length = buffer_length;

status = (*source->ctl_source) (action, source);

if (return_length)
    *return_length = source->ctl_segment_length;

return status;
}

static int dump_text (action, control)
    SHORT         action;
    CTL                 control;
{
/************************************
 *
 *        d u m p _ t e x t
 *
 ************************************
 *
 * Functional description
 *        Open a  blob and write the
 *        contents to a file
 *
 ************************************/

FILE         *text_file;
CHAR         buffer [512];
SHORT         length;
```

```
int         status;


if (!(text_file = fopen ("foo.nr", "w")))
    return FAILURE;

while (!(status = caller (ACTION_get_segment, control,
        sizeof(buffer) - 1, buffer, &length)))
    {
    buffer[length] = 0;
    fprintf (text_file, buffer);
    }

fclose (text_file);

if (status != gds_$segstr_eof)
    return status;

return SUCCESS;

}

static void set_statistics (filename, control)
char * filename;
CTL control;
{
/************************************
 *
 *s e t _ s t a t i s t i c s
 *
 ************************************
 *
 * Functional description
 *Sets up the statistical fields
 * in the passed in ctl structure.
 * These fields are:
 *       ctl_max_segment - length of
 *       longest seg in blob (in
 *       bytes)
 *       ctl_number_segments - # of
 *       segments in blob
 *       ctl_total_length - total length
 *       of blob in bytes.
 *Since the nroff changed the
```

```
 *values, we should reset the
 *ctl structure, so that
 *blob_info calls get the right
 * values.
 *
 ************************************/
register int max_seg_length;
register int length;
register int num_segs;
register int cur_length;
FILE * file;
char * p;
char c;

 num_segs = 0;
 length = 0;
 max_seg_length = 0;
 cur_length = 0;

 file = fopen ("foo.txt", "r");

 while (1)
    {
    c = fgetc (file);
    if (feof (file))
        break;
    length++;
    cur_length++;

    if (c == '\n')    /* that means we are at end of seg */
        {
        if (cur_length > max_seg_length)
           max_seg_length = cur_length;
        num_segs++;
        cur_length = 0;
        }
    }
control->ctl_max_segment = max_seg_length;
control->ctl_number_segments = num_segs;
control->ctl_total_length = length;
return;
}
```

```
static int read_text (action, control)
    SHORT        action;
    CTL                control;
{
/************************************
 *
 *        r e a d _ t e x t
 *
 ************************************
 *
 * Functional description
 *        Reads a file one line at a time
 *        and puts the data out as if it
 *        were coming from a blob.
 *
 ************************************/

CHAR         *p;
FILE         *file;
SHORT         length;
int         c, status;


p = control->ctl_buffer;
length = control->ctl_buffer_length;
file = (FILE *)control->ctl_data [0];

for (;;)
    {
    c = fgetc (file);
    if (feof (file))
        break;
    *p++ = c;
    if ((c == '\n')  || p >= control->ctl_buffer + length)
        {
        control->ctl_segment_length = p - control->ctl_buffer;
        return SUCCESS;
        }
    }
return gds_$segstr_eof;
}
```

For More Information

For a more detailed discussion of the **gds** calls described here, see:

- Chapter 12, *Using OSRI Calls*
- the *OSRI Guide*

Chapter 10
Using Date Fields

This chapter discusses how to program using the date datatype.

Overview

Because host languages do not directly support InterBase's date datatype, assigning values to date fields requires some manipulation. **Gpre** provides two ways to deal with dates:

- Casting the date as a character string
- Converting the date field to the UNIX date/time structure *tm* and then using the **gds_$encode_date** and **gds_$decode_date** routines.

The following sections discuss these two methods.

Casting a Date Field

Casting a date field moves dates to and from strings for input and display. To cast a date field you qualify a field name by using the **.char[*n*]** qualifier.

To assign or display a date with the **.char[*n*]** qualifier, follow these steps:

1. Append **.char[*n*]** to a field name that contains time or date information.

   The following C statement assigns today's date to a field:

   ```
   strcpy(s.statehood.char[6],'TODAY');
   ```

2. Set the field name equal to the date or time format you want. You use the assignment operator of the host language you are using.

Writing to a Casted Date Field

When you write to a casted date field, the value of *n* depends on the input format. You can assign relative dates, such as "today", or provide dates in a variety of formats. For example, "DD-MMM-YYYY", "DD.MM.YY" and "day month year" are all valid.

The following dates are all acceptable date formats for 15 January 1990:

```
'1-23-90'
'1/15/90'
'1/15/1990'
'January 15, 90'
'January 15, 1990'
'Jan 15, 90'
'15.1.90'
```

Note

The use of single (') or double (") quotes is language-specific.

Punctuation in dates is ignored, except when you use periods as separators. If you use periods as separators, InterBase assumes a "day—month" ordering.

InterBase also accepts three-letter abbreviations of months, such as "APR" and "JUL". However, if you use a shorter abbreviation, it is subject to first match. For example, "J 15" is interpreted as "January 15," although you may have intended to mean "July 15".

If you do not specify the century in the year, InterBase tries to figure out what you intend by looking at the current year. If the difference between the current year and the year in the date is greater than or equal to 50, InterBase assumes you mean the next century.

For example, consider the dates 1/15/30 and 1/15/40. The date 1/15/30 is stored as 1/15/2030. However, 1/15/40 is stored as 7/23/1940. In 1990 then, InterBase stores the date 1/15/40 as 1/15/2040.

InterBase also understands relative chronologies. For example, suppose today is 15 January 1990. The following quoted values equate to the specified dates:

- '**yesterday**' assigns the value 14 January 1990.
- '**today**' assigns the value midnight on 15 January 1990.
- '**now**' assigns the value current time on 15 January 1990.

Make sure the value you specify inside the brackets is large enough to contain the input string. For example, if you specify **.char[12]** and then input "15 January 1990" you get a truncation error. The input value of the date is acutally 15 characters since you have to include spaces in your count.

Punctuation in dates is ignored, except when you use periods as separators. If you use periods as separators, InterBase assumes you intend a day—month ordering.

Retrieving from a Casted Date Field

You can also use casting to retrieve a date. For example:

```
for s in states
  printf ("%s, %s\n", s.state_name, s.statehood.char[12]);
end_for;
```

Unlike the input format, InterBase always returns the date in the format "DD-MMM-YYYY", for example, "15-Jan-1990".

If you store a time value in a field, you can use **.char[26]** to return the full date-time string. For example:

```
for t in times
  printf ("%s\n", start_time.char[26]);
end_for;
```

Converting Date Fields

You can store or retrieve date and time information by using the InterBase **gds_$encode_date** and **gds_$decode_date** routines. These routines convert a date to one of the following formats:

- **gds_$encode_date** converts the UNIX time/date structure to the InterBase date format
- **gds_$decode_date** converts the InterBase date format to the UNIX time/date structure

The UNIX date/time structure is labeled *tm*. This is an array of 32-bit words that represents the second, minute, hour, day, month, year, day of week, day in year, and state of daylight savings time. The time/date structure is shown in the following example:

```
struct tm {

   int     tm_sec;
   int     tm_min;
   int     tm_hour;m
   int     tm_mday;
   int     tm_mon;m
   int     tm_year;
   int     tm_wday;
   int     tm_yday;
   int     tm_isdst;
};
```

Using the UNIX date/time structure is it keeps the information in numeric format. This allows you to do various calculations on the date/time information, as appropriate.

To store or retrieve a date in the C programming language:

1. Declare the UNIX time/date structure in your program:

   ```
   #include <time.h>
   ```

2. Declare the local time/date structure in your program:

   ```
   struct tm d;
   ```

3. Code the appropriate **gds_$encode_date** or **gds_$decode_date** statement:

   ```
   gds_$encode_date (local time/date structure, database
   field);

   gds_$decode_date (database field, local time/date
   structure);
   ```

For example, to store the date contained in the local time/date structure declared above, you could use this statement:

```
gds_$encode_date (&d,&X.BIRTH_DATE);
```

Both date and times in the date conversion routine are passed by reference.

C Date Example

The following C program adds birthdate information to employee records and prints it out twice, once using **char[12]** and the other time with **gds_$decode_date.** For the sake of simplicity, everyone in the company was born on 5 October 1948 at 1:15:45 PM.

If you use this program, remember that if you mix uppercase and lowercase C programs, you must preprocess it with the **either_case** switch.

```
DATABASE DB = FILENAME 'emp.gdb';
#include <sys/time.h>

struct tm d;

main ()
{
READY;
START_TRANSACTION;

FOR X IN EMPLOYEES
  MODIFY X USING
d.tm_year = 48;
d.tm_mday = 5;
d.tm_mon = 9;
d.tm_sec = 45;
d.tm_min = 15;
d.tm_hour = 13;
gds_$encode_date (&d, &X.BIRTH_DATE);
  END_MODIFY;
END_FOR;

FOR X IN EMPLOYEES
  gds_$decode_date (&X.BIRTH_DATE, &d);
  printf ("%s birthdate %d%d%d, %d;%d;%d\n",
      X.FIRST_NAME, d.tm_mon, d.tm_mday, d.tm_year,
      d.tm_hour, d.tm_min, d.tm_sec;
END_FOR;

FOR X IN EMPLOYEES
  printf ("%s birthdate %s\n,
      X.FIRST_NAME, X.BIRTH_DATE.CHAR[12]);
END_FOR:

COMMIT;
FINISH
}
```

Pascal Date Example

This program runs against the atlas database. It demonstrates how to retrieve a date using **gds_$decode_date**. It also uses **gds_$encode_date** to store a date. The last section of the program uses casting to retrieve a date field.

```
program show_time(output);

database atlas = 'atlas.gdb';

var
   tm_struct : gds_$tm;     (*  Declare tm structure. This
                                structure is defined in pascal
                                include file that gpre
                                automatically includes *)

(* The fields of the tm struct are:

   gds_$tm     = record
      tm_sec       : integer32;    seconds (0 - 59)
      tm_min       : integer32;    minutes (0 - 59)
      tm_hour      : integer32;    hours   (0 - 23)
      tm_mday      : integer32;    day of month (1 - 31)
      tm_mon       : integer32;    month of year (0 - 11)
      tm_year      : integer32;    year - 1900, so 1989 is
                                    stored as 89, 1820 is stored
                                       as -80
       tm_wday      : integer32;   day of week (Sunday = 0)
       tm_yday      : integer32;   day of year (0 - 365)
      tm_isdst    : integer32;   1 if DST is in effect    end; *)

begin
  ready atlas;
  start_transaction;

            writeln('This program demonstrates the use of
            writeln('InterBasedate routines');
            writeln;


      for s in states with s.state = 'ME'
         gds_$decode_date(s.statehood, tm_struct);

         writeln('Maine entered the union on ',
```

```
            tm_struct.tm_mon+1:2, '/', tm_struct.tm_mday:2,
            '/', 1900 + tm_struct.tm_year:4);

        writeln('This was day ', tm_struct.tm_yday:3, ' of the
            year');
    end_for;

    for s in states with s.state = 'ME'
        modify s using
            tm_struct.tm_sec := 0;
            tm_struct.tm_min := 30;
            tm_struct.tm_hour := 1;
            tm_struct.tm_mday := 19;
            tm_struct.tm_mon := 8;
            tm_struct.tm_year := 89;
            gds_$encode_date(tm_struct, s.statehood);
        end_modify;

        gds_$decode_date(s.statehood, tm_struct);
        writeln('The date added was ', tm_struct.tm_mon+1:2,
            '/', tm_struct.tm_mday:2, '/', 1900 +
            tm_struct.tm_year:4,' and the time was ',
            tm_struct.tm_hour:2, ':',tm_struct.tm_min:2,
            ':', tm_struct.tm_sec:2);
        writeln('This was day ', tm_struct.tm_yday:3, ' of the
            year');

    end_for;

    for s in states with s.state='ME'
        writeln('Using the casting feature, the date added
            was ', s.statehood.char[25]);
    end_for;
  rollback;
  finish;
end.
```

For More Information

For more information on dates fields, see the chapter on defining fields in the *Data Definition Guide*.

Chapter 11
Programming with Events

This chapter describes InterBase events and explains how to use them in your programs.

Overview

InterBase *events* are a mechanism for detecting and reporting changes in a database. The event mechanism gives your program the means to:

- Register interest in the occurrence of a specific database change, or *event*.
- Receive notice that the event has occurred.

Once your program receives notice that the event has occurred, it can take appropriate action.

Here are a few examples of the use of InterBase events:

- Stock trading

  An event could cause a "sell order" to be issued when the price of a particular stock changes more than a certain amount or percentage.

- Process control

  An event could cause an alarm to sound when temperature fluctuates more than 1 degree.

- Calendar management

  An event could cause a program to send mail when a meeting is scheduled, changed, or cancelled.

- Accounting

  In an accounting system, an event could cause a program to validate balances after a check writing program completes.

The InterBase event mechanism is part of the database engine. Because the engine immediately notifies the user program when a change occurs, the InterBase event mechanism does not have the problems found with two other methods used to detect database changes:

- *Constant polling*, which involves repetitively querying the database, wastes enormous amounts of CPU time.
- *Fixed interval checking*, which involves querying the database at specific time intervals, can miss significant changes.

The InterBase event mechanism allows an interested process to request either synchronous or asynchronous notification of an event's occurrence. A process requesting:

- Synchronous notification "sleeps" until the event occurs.
- Asynchronous notification continues running.

Note

A *program* you write, compile, and execute runs in an operating system *process*. Depending on the server configuration, either the process sleeps or the program sleeps. Unless we specifically mention an "operating system process," we do not intend a technical distinction.

Several GDML statements let you make synchronous notification requests, while more complex OSRI calls let you make both synchronous and asynchronous notification requests.

Understanding the Event Mechanism

The following explanation of the event mechanism is presented in the context of synchronous events implemented in GDML. Asynchronous events are explained later in this chapter.

Event Mechanism Components

The event mechanism usually consists of database and program components, and certain interactions between them.

The database component consists of a trigger that:

- Optionally contains trigger language conditional logic
- Has at least one **post** verb and its arguments

The program components are:

- In your program:
  - An **event_init** statement to register interest in an event
  - An **event_wait** statement to request synchronous notification when the event of interest occurs
- In any other program or interactive session:
  - A transaction in which data is stored, modified, or erased
  - The **commit** statement
- In the InterBase DBMS software:
  - The event manager
  - The event table

If you want to post a change that does *not* occur in the database, you must post it through a GDML program. An example of posting an event through a program is shown under the *Variations* section, later in this chapter.

Processing an Event

To explain how the event mechanism works, we'll discuss a theoretical stock trading application. The application receives synchronous notification of a change in the value of ACME stock, as determined by testing a field named PRICE.

The Event Setup

First, let's assume you have already defined the trigger CHECK_ACME to the STOCKS database:

```
define trigger check_acme for stocks
  modify:
       if abs(old.price - new.price) > 1
          then post new.name
end trigger
```

Let's examine the situation:

- The trigger:
  - Fires on a database modify
  - Calls a user-defined function named ABS that returns the absolute value of its argument
  - Contains field names that are prefixed with the *old* and *new* trigger language context variables
  - Uses the **if** condition to test whether the *price* of the record being modified has changed by a value greater than 1
  - Contains the **post** verb with the argument *new.name*

When this trigger fires, if the price of the modified stock has changed by more than a dollar, it attempts to "post an event." This consists of passing the name contained in its argument to the event manager. The event manager then checks to see if the name is in the event table, which lists "events of interest."

Now, let's assume that you have written and run a program that contains the following GDML statements:

```
    ⇓

event_init E1 ("ACME");
    ⇓

event_wait E1;
    ⇓
```

The stock trading application:

- Contains an **event_init** statement, with arguments that tell the event manager we are interested in an event named ACME, and that we can reference this "event

of interest" in the source code of this application as E1. When executed, this statement causes the event manager to enter the string "ACME" into the event table.

- Contains an **event_wait** statement that puts the program to sleep until an event that has the handle E1 occurs. (A handle is an abbreviated name for an event you can use in a program.)

Finally, let's assume another program called TICKER updates the price of ACME stock, as shown in the following code fragment:

```
ready stocks.gdb;
start_transaction;

for s in stocks with s.name = "ACME";
  modify s using s.price = s.price + 2;
end_for;

commit;
finish;
```

The program named TICKER:

- Contains a transaction that increases ACME's price by two dollars.
- Contains a **commit** command that makes the change visible to all users.

The usual rules of transaction control apply to this program.

The Event Process

Assuming that ACME's price stands at 38, let's see how all of this works:

- Your application registers interest in an event named "ACME". This causes the event manager to enter the string "ACME" into the event table.
- Your application waits for notification from the event manager.
- When TICKER commits the transaction, the price of ACME becomes 40.
- After TICKER commits the transaction, the CHECK_ACME trigger fires.
- Because the conditions of the trigger are satisfied, the **post** verb executes.
- Posting tells the event manager an event named "ACME" has occurred.
- The event manager checks its event table for an "event of interest" named ACME.
- Finding "ACME" in its table, the event manager sends notification to all interested processes, including your stock trading program.
- Your application "wakes up" and does whatever task is appropriate, such as putting a message on your screen or initiating a buy or sell order.

Variations

The above situation describes a typical use of events, as well as the simplest sequence of processing. Some of the ways that this situation can differ are described below.

The Trigger

We used a fairly simple trigger. Triggers can differ from this example in a number of ways:

- You can use as much or as little conditional logic in the trigger as you need to qualify the event. However, a trigger that contains no conditional logic puts the burden of qualifying the event back onto the program. Avoid this type of abuse.
- The trigger can also contain several conditions, each of which, if satisfied, can post a different event.
- You can define the trigger with any valid characteristics, not just the defaults used here. You can also make it fire on a statement other than a **modify**. For more information on defining triggers, refer to the *Data Definition Guide* and the *DDL Reference*.
- The argument to the **post** verb can be any alphanumeric name up to 31 characters in length. The event manager looks for an event that has the same name.

The Program

The stock trading program registered interest in an event named "ACME" through the **event_init** statement. Here are some other variations:

- If another program has already registered interest in an event having the same name, the event manager does not make another entry in the event table. It will, however, add the process to its list of who to notify when that event occurs.
- An **event_init** statement can register multiple event names, as shown in examples to follow.
- Your program may register interest in more than one event under different handles by executing as many **event_init** statements as needed, each with a unique handle.

The Timing of the Event

The TICKER program posted the event of interest after the stock trading program executed the **event_wait** statement. The real world does not always demonstrate the same, clean timing:

- If an attempt is made to post an event (through a trigger's **post** verb) and no program has registered any interest in it, there is not an entry in the event table. In this case, the event manager ignores the attempted posting.

- If an attempt is made to post an event and a program has registered an interest, but has not yet begun to wait on the event, the event manager takes note of the event. When the program tries to wait, the event manager immediately sends it notification.
- Your program may contain more than one **event_wait** statement, but it sleeps at the first one and waits until that event occurs before continuing.
- If one event occurs while your program is processing another, the event manager sends notification when the program returns to a wait state.

Posting an Event Through a Program

Because the stock trading program was interested in an event that reflected a database change, that event was posted through a trigger. If a program is interested in an event that reflects an external change, you must post that event through a GDML program.

For example, suppose you want to write an application that monitors program errors. You could set up the monitor program to register interest in a program-error event. Then, whenever other application programs have errors, you could have them post that event.

You can also use externally-posted events to communicate between programs running on different platforms, as long as the programs are attached to the same database. For example, a VAX program can post an event to let a Sun program know that it has completed.

You post an event from a program by using the **blr_post** statement. The following example shows how to post an event by using this statement:

```
DATABASE DB = "events.gdb";

static char blr_stuff [] =
{
blr_version4,
blr_post,
    blr_literal, blr_text, 6,0, 'E','V','E','N','T','1',
blr_eoc
};

main ()
{
  int *request_handle;

  READY
  ON_ERROR
```

```
        gds_$print_status (gds_$status);
    END_ERROR;

    START_TRANSACTION
    ON_ERROR
        gds_$print_status (gds_$status);
    END_ERROR;

    if (gds_$compile_request (
            GDS_VAL (gds_$status),
        GDS_REF (DB),
        GDS_REF (request_handle),
        sizeof (blr_stuff),
        blr_stuff))
        gds_$print_status (gds_$status);

    if (gds_$start_request (
        GDS_VAL (gds_$status),
        GDS_REF (request_handle),
        GDS_REF (gds_$trans),
        0))
        gds_$print_status (gds_$status);

    COMMIT;

    FINISH;
}
```

When to Use Events

While the event mechanism is a very useful feature, you should not use this mechanism to accomplish tasks better suited to other mechanisms. For example, although it would be possible to create an event to delete the appropriate occurrence of the POSITION relation when an employee leaves the company, this kind of task falls under the scope of referential integrity. A trigger can easily handle this, without events.

The event mechanism is for simple notification that an event occurred. It enables your program to initiate a task that the database cannot perform, such as running or controlling an external process.

Programming with Events in GDML

Using examples more fully developed than in the previous section, this section explains in more detail how to program events in GDML.

Syntax of Synchronous Events Used in GDML

The following is an informal review of the syntax for the two synchronous event statements in GDML. The formal syntax is presented in the *Programmer's Reference*.

- event_init <event_handle> (<expression>, <expression>, ...);
  - — <event_handle> is a label that is used to identify this event throughout the GDML module. An initialized event can be referred to in subsequent **event_wait** statements.
  - — <expression> can be a quoted string, a host variable, or a database field that names the event of interest.
- event_wait <event_handle>
  - — <event_handle> is the handle declared in **event_init**.
  - — will suspend processing of the program until the posting of one of the <expression> values from the preceding **event_init** statement that has the same <event_handle>.

Defining an Event

An event is defined by specifying:

1. A unique string to identify the event.
2. The conditions under which the event manager should notify interested programs the event has occurred.

For example, suppose you must write a program for an investment bank that wants to track changes to stocks in its portfolio. The program must input stock prices from the ticker, update the price of each stock in the database, and decide whether to buy or sell based on changes to the stock price.

Assume the database has a relation named STOCKS. The relation has two fields, COMPANY and PRICE. To define a trigger that will post an event defined as a 1% change in the price of any stock, you make the following data definition:

```
define trigger for stocks
  modify:
      if new.price / old.price > 1.01 or
```

```
            new.price / old.price < 0.99
            then post new.company;
end_trigger;
```

Upon modification of any record in STOCKS, the trigger automatically fires. The trigger logic checks to see if the change exceeded 1%, and posts an event with a name equal to the company whose stock changed.

Of course, this trigger would not catch incremental changes that added up to more than one percent. This problem could be resolved through the use of another field that represents the price when the previous event was posted for that stock.

Waiting on an Event

Waiting on an event synchronously is supported through two GDML statements:

- **event_init** declares the names of events in which the application is interested.
- **event_wait** does the actual waiting, returning when one of the declared events occurs.

The syntax for **event_init** is as follows:

```
event_init <event_handle> (<expression>, <expression>, ...);
```

The <event_handle> is a label that can be used throughout the GDML module to reference this event declaration. The <expression> can be a simple quoted string, a host variable or a database field. As usual, referencing database fields can only be done in the context of a database request.

The syntax for **event_wait** is as follows:

```
event_wait <event_handle>;
```

<event_handle> is the same as above, and its scope is the GDML module in which the **event_init** was made.

Analyzing an Event

After the **event_wait** statement is executed, at least one of the declared events has occurred. The array **gds_$events[]** specifies which events have occurred. This is an array of longwords. Each element of the array corresponds to an argument in **event_init**. The value of each element is equal to the number of times this event occurred since execution of the last **event_wait** with the same handle.

Transaction Control of Events

Events are under transaction control, which means an event can be committed or rolled back. Interested programs do not receive notification of an event until the transaction from which it was posted is committed. An event can only happen once per transaction. Regardless of how many times a particular event is posted during a transaction, it is regarded as a single event for purposes of notification.

Application Example

Let's assume an investment bank wants to track changes to stocks in its portfolio. An existing program inputs stock prices from the ticker and updates the price of each stock in the database. The bank needs a stock trading program that decides whether to buy or sell based on changes to the stock price.

Example solution:

1. Create a trigger to post an event when the price changes. In this example, a change in stock price posts an event with the same name as the company whose stock changed.

```
define trigger for stocks
  modify:
      if new.price / old.price not between 0.99 and 1.01
         then post new.company;
end_trigger;
```

2. Create an application program that waits for the event.

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

READY DB;
START_TRANSACTION;

/* allocate the event parameter blocks */

length = gds_$event_block (event_buffer, result_buffer, 5,
```

```
    "APOLLO", "DEC", "HP", "IBM", "SUN");
gds_$event_wait (gds_$status, DB, length, event_buffer,
    result_buffer);
gds_$event_counts (count_array, length, event_buffer,
    result_buffer);

while (1)
    {
    /* wait on the events and get the change in the events counts
*/

    gds_$event_wait (gds_$status, DB, length, event_buffer,
        result_buffer);
    gds_$event_counts (count_array, length, event_buffer,
        result_buffer);

    ROLLBACK;
    START_TRANSACTION;

    for (i=0; i < number_of_stocks; i++)
  if (count_array [i])
      {
            FOR S IN STOCKS WITH S.COMPANY EQ event_names [i]
                printf ("COMPANY: %s PRICE: %f\n\n",
                      S.COMPANY, S.PRICE);
            END_FOR;
            }
    }

}
```

Programming Events with OSRI Calls

In addition to the synchronous event statements in GDML described in the previous section, InterBase provides OSRI calls which allow the application to receive notification of posted events either synchronously or asynchronously. These calls are described below.

The Synchronous Call

The GDML **event_init** and **event_wait** statements use **gds_$event_wait()** to do their work. This call can be used to wait synchronously until an event is posted. Control will return from this routine only when one of the specified events is posted.

The calling sequence fo the **gds_$event_wait** call is shown below:

```
gds_$event_wait (status, database_handle, length, event_list,
result_list)
```

The parameters of this call are shown in Table 11-1.

Table 11-1. Gds_$event_wait() Parameters

Parameter	Description
status	InterBase status vector (array of 20 longwords)
database_handle	Handle of previously attached database (long)
length	Length of passed and returned event parameter blocks (short)
event_list	The event parameter block, which registers interest in a number of events (pointer to character buffer)
result_list	The returned event parameter block, which indicates which events have been posted (pointer to character buffer)

Requesting Asynchronous Notification

The call **gds_$que_events()** can be used to request asynchronous notification. Control will return immediately, and notification of an event will happen via a call to a user-specified routine, or asynchronous trap (AST).

The calling sequence for **gds_$que_events** is

```
gds_$que_events (status, database_handle, event_handle, length,
event_list, ast, ast_argument)
```

The parameters of this call are shown in Table 11-2.

Table 11-2. Gds_$que_events() Parameters

Parameter	Description
status	InterBase status vector (array of 20 longwords)
database_handle	Handle of previously attached database (long)
event_handle	Returned event identifier (pointer to long)
length	Length of passed and returned event parameter blocks (short)
event_list	The event parameter block, which registers interest in a number of events (pointer to character buffer)
ast	Address of routine to call for notification (pointer to int)
ast_argument	First argument to be passed to ast (long)

Writing an Asynchronous Trap

The asynchronous trap should be written to take three arguments, as follows:

```
user_ast (ast_argument, length, events_list)
```

The first is the argument that is specified in the call to **gds_$que_events**. This allows the programmer to determine which event or set of events has been posted. The length and events_list arguments specify the updated event parameter block.

An ast is called from InterBase, so the user should process the trap quickly and return control to the database. A suggested scheme for writing a trap is to flag the event as having happened and return. At appropriate intervals in the execution of the code, this flag could then be checked.

Note

You cannot generate database requests from within an asynchronous trap.

The Event Parameter Block

Both the synchronous and asynchronous calls utilize an event parameter block to indicate the events of interest. Each event has an event count, which is the number of times that an event has been posted since the first process registered an interest in it. Initially, the event count should be zero. The event manager will update this count.

On each successive call, the parameter block will be checked against the current event count for a given event. If the event count is greater than the count in the event parameter block, an event has occurred since the last time the process was notified.

The format of the block is:

- The version number, a one-byte field set by default to 1.
- An event name, consisting of a one-byte length and the name itself.
- The event count, a VAX-style longword. This means that there are four bytes, with the low-order byte first.
- A null, to indicate the end of the parameter block.

Allocating an Event Block

There are two utilities that are useful in manipulating event parameter blocks. (They are, in fact, used by **gpre** to get its work done.) The first, **gds_$event_block()**, is used to allocate an event block dynamically. The format of this call is as follows:

```
unsigned long gds_$event_block (&event_buffer, &result_buffer,
    number_args, event1, event2, ...)
```

The parameters of this call are shown in Table 11-3.

Table 11-3. Gds_$event_block() Parameters

Parameter	Description
event_buffer	The address of a character pointer in which to store the address of the allocated event parameter block
result_buffer	The resultant event parameter block.
number_args	A short that indicates the number of arguments to follow
event <n>	A null-terminated string that names an event to register

Getting the Event Counts

The call **gds_$event_counts** can be used to get the change in value of the events in the event parameter block, as well as to prepare the block for the next call to **gds_$event_wait** or **gds_$que_events**.

The calling sequence for **gds_$event_counts** is as follows:

```
gds_$event_counts (count_array, buffer_length, event_buffer,
result_buffer)
```

The parameters of this call are shown in Table 11-4.

Table 11-4. Gds_$event_counts() Parameters

Parameter	Description
count_array	An array of longwords of order equal to the number of events declared
buffer_length	A short equal to the length of the event parameter lock
event_buffer	The address of the initial event buffer
result_buffer	The address of the output buffer from an event call

This call updates the *count_array* to contain the difference in the event counts for each of the events in the parameter block. The *event_buffer* is also updated to contain the values in the *result_buffer*, to prepare for the next event call.

Sample Synchronous Event Program

Using the example presented earlier in this chapter, assume you want to write a program that waits until an event is posted for one of the stocks, then prints out the price of all stocks. A sample program to do this follows, using InterBase's GDML language embedded in C. In this program the change in event counts is not used.

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

main()
{
char      *event_buffer, *result_buffer;
long      count_array [5];
short     length;

ready db;
start_transaction;

/* allocate the event parameter block and the result block */

length = gds_$event_block (&event_buffer, &result_buffer, 5,
    "APOLLO", "DEC", "HP", "IBM", "SUN");

while (1)
   {
   /* wait on the events and get the change in the events counts
*/

   gds_$event_wait (gds_$status, db, length, event_buffer,
        result_buffer);
   gds_$event_counts (count_array, length, event_buffer,
        result_buffer);

   FOR S IN STOCKS
       printf ("COMPANY: %s PRICE: %f\n\n", S.COMPANY, S.PRICE);
   END_FOR;
   }

commit;
}
```

Sample Asynchronous Event Program

This program below also prints out the price of all stocks when the appropriate event is posted. However, this program uses an asynchronous event rather than a synchronous one:

```
/* Here is an example of getting asynchronous notice
   stock price changes.
   NB: An event may occur between our transactions so
   that the proce printed is NOT the price associated
   with the event_flag(s) that caused the printing.
   In this case, the new price will be printed a
   second time (unless the event occurs again sometime
   before the print loop is reached!).
*/
DATABASE DB = "stocks.gdb";
#define number_of_stocks 5
#define TRUE   1
#define FALSE  0
char *event_names [] = {"APOLLO", "DEC", "HP", "IBM", "SUN"};
static int event_flag = 0;
static ast_routine();

main()
{
char      *event_buffer, *result_buffer;
long      count_array [number_of_stocks];
short     length;
long      event_id;
int       i, first_time;

READY DB;
START_TRANSACTION;

/* GDS_REF and GDS_VAL are macros defined in gds.h to ensure
   that arguments to functions are passed correctly regardless
   of platform */


length = gds_$event_block (GDS_REF(event_buffer),
GDS_REF(result_buffer), 5, "APOLLO", "DEC","HP", "IBM", "SUN");
gds_$que_events (GDS_VAL(gs_$status), GDS_REF(DB),
GDS_REF(event_id),
    length, GDS_VAL(event_buffer), ast_routine,
```

```
GDS_VAL(result_buffer));

if (gds_$status[1])
   {
   gds_$print_status (gds_$status);
   return;
   }
first_time = TRUE;

while (TRUE)
   {
   /* can substitute other processing instead of sleeping */
   if (!event_flag)
       sleep (20);
   else
       {
       event_flag = 0;
       gds_$event_counts (GDS_VAL(count_array), length,
GDS_VAL(event_buffer),
           GDS_VAL(result_buffer));
       ROLLBACK;
       START_TRANSACTION;
       FOR (i=0; i < number_of_stocks; i++)
           if (count_array [i])
               {
               FOR S IN STOCKS WITH S.COMPANY EQ
                   event_names [i]
                   printf ("COMPANY: %s PRICE: %f\n",
                   S.COMPANY, S.PRICE);

                   if (!first_time)

                   printf ("AT LEAST %d PRICE CHANGE(S)
                   SINCE LAST POSTING\n\n",
                   count_array[i]);
               END_FOR;
               }
           gds_$que_events (GDS_VAL(gds_$status), GDS_REF(DB),
GDS_REF(event_id),
       length, GDS_VAL(event_buffer), ast_routine,
GDS_VAL(result_buffer));
       if (gds_$status[1])
           {
           gds_$print_status (gds_$status);
```

```
            return;
            }
        first_time = FALSE;
        } /* end event_flag */
    } /* end while */
} /* end main */

static ast_routine (result_buffer, length, updated_buffer)
    char *result_buffer, *updated_buffer;
    short length;
{
event_flag++;
while (length--)
    *result_buffer++ = *updated_buffer++;
}
```

For More Information

For more information on writing triggers, refer to the section on preserving data integrity in the *Data Definition Guide.*

For more information on trigger language and the **define trigger** statement, see the *DDL Reference.*

See the GDML section of the *Programmer's Reference* for syntax for:

- **event_init**
- **event_wait**

Chapter 12
Using OSRI Calls

This chapter describes the **gds** call interface routines. You can use these calls in building applications that generate dynamic queries or in developing information calls for performance measurement and tuning.

Overview

The InterBase call interface is called Open Systems Relation Interface (OSRI). It is designed to be independent of any particular database implementation, and is suitable for network transmission between dissimilar machines.

The call interface is used internally by all InterBase components. The higher level interfaces, such as GDML and SQL are layered on top of the call interface. At its simplest, the call interface provides routines that:

- Attach and detach databases
- Start and stop transactions

- Provide information about what is happening during a database session
- control message traffic between the access method and the calling program.

The call interface consists of:

- function, or **gds**, calls
- a fully programmable sublanguage called Binary Language Representaion (BLR) that provides:
  - Loops
  - Control structures
  - Synchronization
  - Recursion
  - Aggregates
  - Comparison operators
  - A field-level interface with automatic datatype conversion

You can learn a lot about the request language by using the **set blr** option in **qli**. The **set blr** option displays the BLR of a query before displaying the results of a query.

OSRI Components

The call interface contains the following components:

- The status vector
- Arguments for **gds** calls
- Parameter blocks

The following sections discuss each of the components.

The Status Vector

Each of the routines discussed in this chapter returns error messages to its calling program through the *status vector*. The status vector is a vector of 20 longwords. A vector is an array datatype with a single dimension.

You can supply an array of 20 longwords for each call to a **gds** routine. You can use the status vector *gds_$status*, or you can define your own.

To display the contents of the status vector, you use the GDML library routine **gds_$print_status** (*gds_$status*). **Gpre** declares *gds_$status*, so you do not have to define it in your program.

In a GDML program, you use the **gds_$print_status** routine with the **on_error** clause. The **on_error** clause specifies the action a program should take if an error occurs during the execution of a GDML statement.

If you pass zero as the address of the status vector and your program encounters an error, InterBase writes the message(s) to standard error and aborts your program. The status vector for information calls indicates that InterBase accepted and understood the request for information. It does not indicate that it supplied all requested information.

You should code your program so that it checks the value returned in the status vector. By checking the status vector, you can confirm that a call has succeeded. The second element of the status vector is 0 when a call succeeds. It is non-zero when it fails. If you do not include error checking, a call can fail without your knowing about it.

Parameters for Gds Calls

The datatype and usage of each parameter are listed in each syntax description. The datatype can be:

- Byte, an 8-bit quantity
- Ubyte, an unsigned byte
- Short, a 16-bit quantity (word)
- Ushort, an unsigned word
- Long, a 32-bit quantity (longword)
- Ulong, an unsigned longword
- Quad, a 64-bit quantity (quadword)
- Uquad, an unsigned quadword
- Vector_char, a vector of characters
- Vector_long, a vector of 20 longwords
- Vector_byte, a vector of bytes
- Unspec, the address of a buffer whose description is established elsewhere, or a binary value the size of which is given elsewhere

The usage parameter specifies how the argument is to be used:

- In, which means the parameter is input only and not altered by the access method.
- Out, which means the parameter is output only and not read by the access method. The access method supplies a value for the parameter.
- Inout, which means the access method reads and writes the paramters. Ordinarily, the input value is either zero or a value established in another call.

Parameter Blocks

InterBase has three parameter blocks:

- The *database parameter block* (DPB) is used in the **gds_$attach_database** and **gds_$create_database** calls.
- The *transaction parameter block* (TPB) is used in the **gds_$start_transaction** and **gds_$start_multiple** calls.
- The *blob parameter block* (BPB) is used with the **gds_$open_blob2** and **gds_$create_blob2** calls.

Each parameter block is specific to particular calls although the format of both blocks is the same. The format of the paramter block and components of the option list follow:

Syntax

```
pb ::= version_number option_list-commalist
version_number ::= {gds_$dpb_version1 |
   gds_$tpb_version3 | gds_bpb_version1}
option_list ::= option_code length value
```

Components	Datatype	In/Out
version_number	ushort	in
option_code	byte	in
length	ubyte	in
value	vector_byte	in

The options for these parameter blocks are described in the following places:

- The DPB parameter options are listed in the *Creating and Attaching a Database* section
- The TPB parameter options are listed in the *Starting and Stopping Transaction* section
- The BPB parameter options are listed in the *Using Blob Calls* section.

Representing Numeric Values

Numeric values, such as the page size, are passed as a string of byte values representing the columns of a base-256 number. For example, the number 2048 is represented in base-256 as:

$$(2048)_{10} = (80)_{256}$$

Thus, the value 2048 is encoded as a two-byte vector containing the values 0 and 8. The values are stored in order from smallest column value (units) to largest.

For example, the following code fragment shows how to use the DPB to specify a page size of 2048 bytes:

```
static char
  dpb [] = {
      gds_$dpb_version1,
      gds_$dpb_page_size, 2, /* create db with large page size*/
        0, 8};               /* byte-backward version of 2048 */
```

To convert numeric values with datatypes of short to base-256 numbers, you can use the macro *blr_word (n)* which is included in the *gds.h* file. This macro has the following format:

```
define blr_word (n) (n % 256), (n/256)
```

Creating and Attaching a Database

The following sections describe the:

- Database parameter block
- Options forOSRI the database parameter block (DPB)
- **gds_$create_database** routine
- **gds_$attach_database** routine

The Database Parameter Block

On calls to **gds_$attach_database** and **gds_$create_database**, you use messages to pass parameters to function calls. A parameter option is passed as a message using the database parameter block (DPB). The DPB is a byte vector that starts with a one-byte version number (as specified by the compile-time constant **gds_$dbp_version1**) followed by zero or more options. Option codes are compile-time constants that you use in C or Pascal.

Each option contains an option code and length (which may be zero) followed by a value of the specified length (if appropriate).

The syntax for the DPB follows:

Syntax

```
dpb ::= version_number option_list-commalist
version_number ::= gds_$dpb_version1
option_list ::= option_code length value
```

The following is a list of parameter option codes for the DPB:

- gds_$dpb_page_size, which specifies the page size in bytes.
- gds_$dpb_dbkey_scope, which specifies the scope of the dabase key's context.
- gds_$dpb_num_buffers, which specifies the number of buffers allocated for the database.
- gds_$dpb_enable_journal, which names the journaling subsystem.
- gds_$dpb_disable_journal, which turns off journaling.
- gds_$dpb_sweep, which cause the access method to remove unneeded records.
- gds_$dpb_sweep_interval, which specifies how long between sweeps.
- gds_$dpb_activate_shadow, which creates a database from a shadow file.

DPB Options

Because the **gds_$create_database** routine creates and attaches a database, the DPB options for this routine include both create and attach options.

Gds_$dpb_page_size is a create option. You should override the default of 1024 if you have very large relations. The choices for page size are 1024, 2048, 4096, and 8192. To change the page size, you must back up the database and then restore it with the new page size. Increasing the page size increases the size of the index page. This reduces the depth of the index tree and could improve performance in very large databases.

Gds_$dpb_dbkey_scope is a create and an attach option. If you are using database keys for direct access to records, you can specify the duration of their stability. The scope can be the entire transaction (value of 0) or the entire database attach (value of 1).

OSRI supports the same dbkey retrival option at the BLR level that DSRI supports. If you have a DSRI-compliant application, it can operate with OSRI in the same manner.

Gds_$dpb_num_buffers is a create and an attach option. It affects the size of the cache used to buffer data on disk. The default number of buffers is 75. The value cannot be less than 1. Increasing the number of buffers can improve performance, but increases the amount of memory used by the access method.

The following two sections provide complete examples of the **gds_$create_database** routine in VAX C and Apollo C, respectively.

Creating a Database

The **gds_$create_database** routine creates a new, empty database, and attaches it to the calling program. Although the database contains no user data, it does contain a full set of system relations.

The **gds_$create_database** routine supersedes any database with the same name. Therefore, if you want to keep your present databases, try to attach to a database before you create a new one. If the attach succeeds, don't call the create routine.

Gdef and the restore process of **gbak** are the primary users of the **gds_$create_database** routine. You rarely need this routine. However, if you develop an application that dynamically creates individual databases, you must to call this routine.

The calling sequence of the **gds_$create_database** routine follows.

Syntax: *status* = **gds_$create_database** (*status_vector, db_name_length, db_name, db_handle, dpb_length, dpb*)

Arguments:

Parameter	Datatype	In/Out	Value/Reference
status	long	out	
status_vector	vector_long	out	reference
db_name_length	ushort	in	value
db_name	vector_char	in	reference
db_handle	ulong	in/out	reference
parm_buffer_length	ulong	in	value
parm_buffer	ulong	in	reference

The **gds_$create_database** call accepts:

- The database name.
- A pointer to the database parameter block.
- A pointer to a database handle, which must be set to zero.

It returns a:

- Non-zero as its function value if an error occurred.
- Status vector.
- Database handle, which can be used in subseqeuent calls to refer to this particular database attachment.

The following code fragment shows the **gds_$create_database** routine with its parameters specified.

```
if (gds_$create_database
   (status_vec, 0, "new.gdb", &db, sizeof (dpb), dpb))
   gds_$print_status (status_vec);
```

The database name is assumed to be a null-terminated string if *db_name_length* is zero. Otherwise, you must provide the length of the name. The database handle must always be initialized to zero prior to an attach or create call.

The following two sections provide complete examples of the **gds_$create_database** routine in VAX C and Apollo C, respectively.

VAX C Database Creation Example

This routine uses the call interface to create a new database with a non-standard page size. This program is in VAX C, using the normal C call mechanism.

```
/*****************************************
 *
 *c r e a t e _ d a t a b a s e
 *
 *****************************************/

#include <gds.h>

int
   status_vec [20],/* status vector */
   *db;        /* database handle */

static char
   dpb [] = {
       gds_$dpb_version1,
       gds_$dpb_page_size, 2,/*create db with large page size */
         0, 8};/* byte-backward version of 2048 */
main ()
{
db = 0;

if (!(gds_$create_database
   (status_vec, 0, "new.gdb", &db, sizeof (dpb), dpb)))
   gds_$detach_database (status_vec, &db);

if (status_vec [1])
   {
   printf ("error in CREATE module");
   gds_$print_status (status_vec);
   }
}
```

Apollo C Database Creation Example

This routine uses the call interface to create a new database with a non-standard page size. This program is in Apollo C, using the standard call mechanism. In other versions of C, the call arguments would need referencing information.

```
/*****************************************
 *
 *c r e a t e _ d a t a b a s e
```

```
 *
 ****************************************

#include "/sys/ins/gds.ins.c"

int
   status_vec [20],/* status vector */
   *db;        /* database handle */

static char
   dpb [] = {
       gds_$dpb_version1,
       gds_$dpb_page_size, 2,/* create db with large page size */
         0, 8};/* byte-backward version of 2048 */

main ()
{
db = 0;

if (!(gds_$create_database
   (status_vec, 0, "new.gdb", db, sizeof (dpb), dpb)))
          gds_$detach_database (status_vec, db);

if (status_vec [1])
{
printf ("error in CREATE module");
   gds_$print_status (status_vec);
   }
}
```

Attaching a Database

A call to **gds_$attach_database** opens an existing database for program access. It is functionally equivalent to the GDML **ready** statement, which opens an existing database for access. If you want to override the defaults of the **ready** statement's attach, use the **gds_$attach_database** routine.

The **gds_$attach_database** routine first passes the name of the database file (and the length of the name) to the access method. The access method returns the database handle, a longword identifier used in calls to other **gds** routines. The attach routine also passes the address and length of the DPB to the access method.

The calling sequence of the **gds_$attach_database** routine follows.

Syntax:

```
status = gds_$attach_database (status_vector,
db_name_length, db_name, db_handle, parm_buffer_-
length, parm_buffer)
```

Arguments:

Parameter	Datatype	In/Out	Value/Reference
status	long	out	
status_vector	vector_long	out	Reference
db_name_length	ushort	in	Value
db_name	vector_char	in	Reference
db_handle	ulong	in/out	Reference
parm_buffer_length	ulong	in	Value
parm_buffer	ulong	in	Reference

The database name is assumed to be a null-terminated string if *db_name_length* is zero. Otherwise, you must provide the length of the name. The database handle must always be initialized to zero prior to an attach or create call.

Table 12-1 lists the options of the attach call.

Table 12-1. ***Gds_$attach_database*** *Options*

Option	Meaning	Default
gds_$dpb_dbkey_scope	The scope of the database key's context: either the transaction (0) or the entire database attach session (1).	0
gds_$dpb_num_buffers	Number of buffers allocated for use with the database.	75
gds_$dpb_sweep	Causes the access method to read all records in the database and remove versions that are no longer needed.	Disabled

*Table 12-1. **Gds_$attach_database** Options continued*

Option	Meaning	Default
gds_$dpb_sweep_interval	Specifies how long between sweep intervals.	20,0000 transactions
gds_$dpb_enable_journal	Names the journaling subsystem that maintains an after-image journal for the database.	None
gds_$dpb_disable_journal	Turns off the after-image journaling.	None
gds_$dpb_verify	Causes the access method to validate that internal structures are consistent.	gds_$dpb_pages
gds_$dpb_shadow	Creates a database from a shadow file.	None

The following options require exclusive access to the database:

- **gds_$dpb_enable_journal**
- **gds_$dpb_disable_journal**
- **gds_$dpb_verify**

When you call the **gds_$attach_database** routine with a DPB parameter option requiring exclusive access, the system waits until all users finish (**detach** in OSRI, **finish** in GDMl, or **release** in SQL) before attaching the database. When the function completes, the database is attached.

Table 12-2 lists the length and value for each of the DPB options.

Table 12-2. DPB Option Lengths and Values

Option	Meaning	Default
gds_$dpb_dbkey_scope	1	0 or 1
gds_$dpb_num_buffers	1	Number of buffers to allocate
gds_$dpb_sweep	1	**gds_$dpb_records**
gds_$dpb_sweep_interval	1, 2, or 4	A base 256 number

Table 12-2. DPB Option Lengths and Values continued

Option	Meaning	Default
gds_$dpb_enable_journal	Length of journal file name in bytes	Journal name
gds_$dpb_disable_journal	0	None
gds_$dpb_verify	2	gds_$dpb_pages gds_$dpb_records gds_$dpb_no_update gds_$dpb_repair
gds_$dpb_shadow	0	None

When you specify a value for **gds_$dpb_num_buffers** or **gds_$dpb_sweep_interval**, its size determines the number of bytes.

For more detailed information on **gds_$dpb_dbkey_scope** and **gds_$dpb_num_buffers**, see the preceding section on creating a database with the **gds_create_database** routine.

VAX C Database Attachment Examples

The following two sections provide examples of the **gds_$attach_database** routine in VAX C and Apollo C, respectively.

Sweep option

This routine uses the call interface to attach the employee database for a garbage collection pass. This program is in VAX C, using the normal C call mechanism.

```
/*****************************************
 *
 *s w e e p _ d a t a b a s e
 *
 *****************************************

#include <gds.h>

int
   status_vec [20],/* status vector */
   *db;        /* database handle */
```

```
static char
  dpb [] = {
      gds_$dpb_version1,
      gds_$dpb_sweep, 1,/* attach for sweep of db */
      gds_$dpb_records 1 gds_$dpb_pages 1/* sweep everything */
      gds_$dpb_indices 1 gds_$dpb_transactions };

main ()
{
db = 0;

if (!(gds_$attach_database
  (status_vec, 0, "emp.gdb", &db, sizeof (dpb), dpb)))
  gds_$detach_database (status_vec, &db);

if (status_vec [1])
  {
  printf ("error in SWEEP module");
  gds_$print_status (status_vec);
  }
}
```

Journaling option

The following section gives an example of the use of the **gds_$dpb_enable_journal** option in VAX/VMS C.

This routine uses the call interface to attach the employees database with journaling the the file *emp.jnl* enabled. This program uses the normal C call mechanism.

```
#include <gds.h>

int
  status_vec [20], /* status vector */
  *db;             /* database handle */

static char
  dpb [] = {
      gds_$dpb_version1,
      gds_$dpb_enable_journal, 7,
'E' 'M' 'P' '.' 'J' 'N' 'L'}

main ()
{
db = 0
```

```
if (!(gds_$attach_database
   (status_vec, 0, "emp.gdb", &db, sizeof (dpb), dpb)))
   gds_$detach_database (status_vec, &db);
if (status_vec [1])
   gds_$print_status (status_vec);
}
```

Apollo C Database Attachment Example

This routine uses the call interface to attach the employee database for a garbage collection pass. This program is in Apollo C, using the standard call mechanism. In other versions of C, the call arguments would need referencing information.

```
/*****************************************
 *
 * s w e e p _ d a t a b a s e
 *
 *****************************************

#include "/sys/ins/gds.ins.c"
int
   status_vec [20],/* status vector */
   *db;        /* database handle */

static char
   dpb [] = {
       gds_$dpb_version1,
       gds_$dpb_sweep, 1,              /* attach for sweep of db */
       gds_$dpb_records 1 gds_$dpb_pages 1 /* sweep everything */
       gds_$dpb_indices 1 gds_$dpb_transactions };

main ()
{
db = 0;

if (!(gds_$attach_database
   (status_vec, 0, "emp.gdb", db, sizeof (dpb), dpb)))
   gds_$detach_database (status_vec, db);

if (status_vec [1])
   {
   printf ("error in SWEEP module");
   gds_$print_status (status_vec);
   }
}
```

Detaching a Database

When you finish with a database, you can close it with a call to the **gds_$detach_database** routine. This routine is functionally equivalent to the GDML **finish** statement. It reduces the use of virtual memory. It also releases the local and remote resources you use when you access a remote database.

If you call **gds_$detach_database** while there are active transactions not in limbo, the detach fails and the access method returns an error code in the status vector. If there are any transaction in limbo when you call **gds_$detach_database**, they stay in limbo.

If you have a program that does not have any GDML statements, you can call **gds_$detach_database** to close the database. By coding the detach call rather than the **finish** statement, you eliminate the need to preprocess the program with **gpre**.

The calling sequence of the **gds_$detach_database** routine follows.

Syntax:

```
status = gds_$detach_database (status_vector,
db_handle)
```

Arguments:

Parameter	Datatype	In/Out	Value/Reference
status	long	out	
status_vector	vector_long	out	reference
db_handle	ulong	in/out	reference

The following conditional statement detaches a database:

```
if (handle)
  gds_$detach_database (status_vector, handle);
```

You must specify a valid handle that identifies an attached database. The specified database is detached when this statement executes. A successful return from the detach call sets the handle to null.

Starting and Stopping Transactions

The following sections describe the:

- Transaction parameter block
- Options for the transaction parameter block
- **gds_$start_transaction** routine
- **gds_$commit_transaction** routine
- **gds_$rollback_transaction** routine

The Transaction Parameter Block

On calls to **gds_$start_transaction** and **gds_$start_multiple**, you use the transaction parameter block (TPB). The TPB is a byte vector that starts with a one-byte version number followed by zero or more options. Option codes are compile-time constants that you use in C or Pascal.

Each option contains an option code and length (which may be zero) followed by a value of the specified length (if appropriate).

The syntax for the TPB follows.

Syntax

```
tpb ::= version_number option_list-commalist
version_number ::= gds_$tpb_version3
option_list ::= option_code length value
```

The following is a list of parameter option codes for the TPB:

Transaction Parameter Block Options

Transaction options consist of one-byte codes, as specified by compile-time constants with the prefix **gds_$tpb**. In the case of lock options, the one-byte lock level code is followed by a length, a relation name, and then a one-byte access option code.

Table 12-3 lists valid parameter values. The default option is listed first for each transaction option.

Table 12-3. Transaction Parameter Block Options

Transaction Option	Meaning
gds_$tpb_concurrency **gds_$tpb_consistency**	The default mode for a transaction specifies a high throughput, high concurrency transaction with generally acceptable consistency. The optional mode specifies that the operations performed in the transaction should be serializable in some order.
gds_$tpb_wait **gds_$tpb_nowait**	The default action if your program encounters a locked relation is to wait until the lock goes away. This avoids the dynamic conflicts which can occur when the nowait option is used.
gds_$tpb_write **gds_$tpb_read**	The default intention of a transaction is that it will write data.
gds_$tpb_lock_level	Specifies the intention of a transaction toward a specified relation. The format of **gds_$tpb_lock_level** is: *lock option, length, relation_name, access_option* *lock_option::=* **(gds_$tpb_lock_read |** **gds_$tpb_lock_write)** *access_option::=* **(gds_$tpb_shared |** **gds_$tpb_protected \|** **gds_$tpb_exclusive)** The default *access_option* is concurrent shared access, the protected option allows concurrent restricted access, and the exclusive option disallows any concurrent access.
gds_$tpb_ignore_limbo	Treats limbo transactions as if they are active.

A call to **gds_$commit_transaction** or **gds_$rollback_transaction** terminates the transaction. A GDML **commit** or **rollback** does the same thing more simply.

Gds_$start_transaction

A call to **gds_$start_transaction** or **gds_$start_multiple** initiates a transaction. These routines are functionally equivalent. Use the **gds_$start_multiple** routine with languages that do not handle a variable number of arguments on a call. It is also useful if you must code the start transaction function before you know how many databases you will access. For a complete description of the **gds_$start_multiple** routine, see the *OSRI Guide*

The calling sequence for the **gds_$start_transaction** routine follows.

Syntax:

```
status = gds_$start_transaction (status_vector,
transaction_handle, db_handle_count, {db_handle,
tpb_length, tpb_address}... )
```

Arguments:

Parameter	Datatype	In/Out
status	long	out
status_vector	vector_long	out
transaction_handle	ulong	out
db_handle_count	ulong	in
db_handle	ulong	in
tpb_length	ushort	in
tpb_address	ubyte	in

The *db_handle_count* specifies the number of database handles (db_handle) passed in this call. Because a single transaction can access multiple databases, this routine passes information about each database it accesses and the conditions of access for that database.

For more information about the significance of these options, see the discussion of the **start_transaction** statement in the GDML section of the *Programmer's Reference*.

The following two sections provide examples of the **gds_$start_transaction** call interface routines in VAX C and Apollo C, respectively.

VAX C Transaction Example

This routine uses the call interface to start a read/write transaction, reserving the BADGE_NUM relation of the employee database. This transaction does not deadlock

while it retrieves a unique new badge number. This program is in VAX C and uses the normal C calling sequence, passing "out" and "inout" parameters by reference.

```
/*****************************************
 *
 *g e t _ n e w _ b a d g c
 *
 *****************************************

#include <gds.h>

int
   status_vec [20],/* status vector */
   *db,        /* database handle */
   *trans,     /* transaction handle */
   status;     /* status returned by each call */

static char
   tpb [] = {
       gds_$tpb_version1,
       gds_$tpb_write,/* read/write transaction */
       gds_$tpb_consistency, /* serializable */
       gds_$tpb_wait,/* wait on lock */
       gds_$tpb_lock_write, 9, /* reserving BADGE_NUM for update
*/
       'B','A','D','G','E', '_', 'N', 'U', 'M',
       gds_$tpb_protected }; /* don't allow other writers */
main ()
{
db = trans = 0;

status = gds_$attach_database (status_vec, 0, "emp.gdb", &db,
0, 0);
if (!status)
   status = gds_$start_transaction
       (status_vec, &trans, 1, &db, sizeof (tpb), tpb);


if (!status)
   status = do_work (db, trans);

if (!status)
   gds_$commit_transaction (status_vec, &trans);
else if (trans)
   gds_$rollback_transaction (status_vec, &trans);
```

```
if (db)
  gds_$detach_database (status_vec, &db);

if (status_vec [1])
  {
  printf ("error in READY/START module");
  gds_$print_status (status_vec);
  }
}
```

Apollo C Transaction Example

This routine uses the call interface to start a read/write transaction, reserving the BADGE_NUM relation of the employee database. This transaction does not deadlock, and allows you to retrieve a unique new badge number. This program is in Apollo C, using the standard call mechanism. In other versions of C, the call arguments need referencing information.

```
/*****************************************
 *
 *g e t _ n e w _ b a d g e
 *
 *****************************************
 *

#include "/sys/ins/gds.ins.c"

int
  status_vec [20],/* status vector */
  *db,       /* database handle */
  *trans,    /* transaction handle */
  status;    /* status returned by each call */

static char
  tpb [] = {
      gds_$tpb_version1,
      gds_$tpb_write,/* read/write transaction */
      gds_$tpb_consistency, /* serializable */
      gds_$tpb_wait,/* wait on lock */
      gds_$tpb_lock_write, 9, /* reserving BADGE_NUM for update
*/
      'B','A','D','G','E', '_', 'N', 'U', 'M',
      gds_$tpb_protected }; /* don't allow other writers */
```

```
main ()
{
db = trans = 0;

status = gds_$attach_database (status_vec, 0, "emp.gdb", db, 0,
0);
if (!status)
   status = gds_$start_transaction (status_vec, trans, 1,
       db, sizeof (tpb), tpb);

if (!status)
   status = do_work (db, trans);

if (!status)
   gds_$commit_transaction (status_vec, trans);
else if (trans)
   gds_$rollback_transaction (status_vec, trans);

if (db)
   gds_$detach_database (status_vec, db);

if (status_vec [1])
   {
   printf ("error in READY/START module");
   gds_$print_status (status_vec);
   }
}
```

Checking on Status with Information Calls

When you use information calls, you use:

- An item list buffer
- A result buffer

The InterBase call interface provides a variety of information calls that let you check on the status of all current entities. These routines are:

- **gds_$database_info** for information about attached databases.
- **gds_$blob_info** for information about open blobs.
- **gds_$transaction_info** for information about current transactions.

The following sections discuss these two buffers and the information routines.

Specifying the Item List Buffer

When you call an information routine, your program must specify in the *item list buffer* the logical items about which you want data. This buffer is an unstructured but otherwise regular byte vector. The calling program lists the items about which it requires information in the item list buffer.

Each item about which you can request information has a compile-time constant value associated with it. This logical value maps to a byte value that is defined in an include file for your host language. The include file is installed during InterBase installation.

The following call requests information about transactions in limbo:

```
static char buffer [40],
  limbo_infor [] = {gds_$infor_limbo};

if (gds_$database_info (status_vector,
      &handle,
      sizeof (limbo_info),
      limbo_info,
      sizeof (buffer),
      buffer))
  {
  gds_$print_status (status_vector);
  return;
  }
```

This call returns database information in the result buffer, which is discussed in the following section.

Using the Result Buffer

The access method returns data about requested items to the *result buffer*, a buffer consisting of multiple *triplets* (units of type, length, and value). Your program must interpret the results of this buffer.

The result buffer has the following syntax:

Syntax:

```
result buffer ::= triplet-list

triplet-list ::= {info-type length value}
```

Arguments:

Name	Datatype
info-type	ubyte
length	ushort
value	short

The value of info-type is the item you requested in the item list buffer. See Table 12-4 for a complete list of these items.

The triplets returned to the result buffer are not aligned. Furthermore, binary numbers are in a generic format. You must convert these numbers to a datatype native to your computer before interpreting them. In a generic binary value, the least significant byte is first, and the most significant is last. The sign is in the last byte.

To interpret a binary value returned by an information call:

1. Determine the size, which can be 1, 2, or 4 bytes.
2. Reverse the order of the bytes.

The following routine converts the contents of a binary value in the result buffer into something you can read:

```
long REV_integer (ptr, length)
  unsigned char *ptr;
  short length;
{
/*************************************
 *
 *R E V _ i n t e g e r
 *
 *************************************
 *
 * Functional description:
```

```
 * Pick up (and convert) an integer
 * of length 1, 2, or 4 bytes.
 *
 ************************************/
long   value;
short  shift;

value = shift = 0;

while (--length >= 0)
   {
   value += (*ptr++) << shift;
   shift += 8;
   }

return value;
}
```

For all information calls, you may receive one of the following triplets:

- If there were no errors, the access method returns a triplet with the value **gds_$info_end** at the end of the result buffer.
- If output into the result buffer was truncated because the result buffer is not large enough to hold all the information you requested, the access method returns a triplet with the type **gds_$info_truncated** as the last triplet in the result buffer. If your program encounters this triplet, all preceding information is valid, but at least one item is missing.
- If an item of requested information is not available, the access method returns an error triplet. This triplet has the same form as other triplets, but the information portion contains only the information type value and a code indicating why the information is not available.

Checking Database Information

You may want to use the **gds_$database_info** call when you are tuning programs. For example, you can find out how much space the database requires for page caches. Another use would be to compare the efficiency of two update strategies.

The calling sequence for the **gds_$database_info** routine follows:

Syntax:

```
status = gds_$database_info (status_vector,
db_handle, item_list_buffer_length,
item_list_buffer, result_buffer_length,
result_buffer)
```

Arguments:

Parameter	Datatype	In/Out	Value/Reference
status	long	out	
status_vector	vector_long	out	reference
db_handle	ulong	in	reference
item_list_buffer_length	ushort	in	value
item_list_buffer	vector_byte	in	reference
result_buffer_length	ushort	in	value
result_buffer	unspec	out	reference

Table 12-4 lists the tuning items about which you might want information. Each information item is a compile-time constant.

Table 12-4. Database Information Items

Information Item	Meaning
gds_$info_end	Marks the end of the information results buffer.
gds_$info_truncated	Marks the end of the result buffer but indicates there are more values to be returned than the buffer has room for.
gds_$info_error	Indicates that the requested information could not be found. The value of the triplet contains the info_type and a reason code.
gds_$info_db_id	Returns the database identifier.
gds_$info_reads	Indicates the number of page reads since the database was attached.
gds_$info_writes	Specifies the number of page writes since the database was attached.
gds_$info_fetches	Indicates the number of reads from the cache since the database was attached.
gds_$info_marks	Provides the number of writes to the cache since the database was attached.
gds_$info_implementation	Specifies the database implementation.
gds_$info_version	Specifies the version identification string of the implementation being accessed.
gds_$info_page_size	Returns the page size for the attached database.

Table 12-4. Database Information Items continued

Information Item	Meaning
gds_$info_num_buffers	Specifies the number of buffers requested for this attachement.
gds_$info_limbo	Provides information about transactions left in limbo as a result of a failure during a two-phase commit.
gds_$info_current_memory	Returns the size of memory currently being used by the database system.
gds_$info_max_memory	Returns the maximum size of memory that was used since the last time this information was requested.

The following call finds out about transactions in limbo for subsequent reconnects or rollbacks.

```
static char buffer [40],
  db_info [] = {gds_$info_page_size, gds_$info_num_buffers};

if (gds_$database_info (status_vector,
      handle,
      sizeof (db_info),
      db_info,
      sizeof (buffer),
      buffer))
  {
  gds_$print_status (status_vector);
  return;
  }
```

Checking Open Blobs

The **gds_$blob_info** routine provides information about an open blob. Use the **gds_$-blob_info** call when you have special blob handling characteristics. The information call can tell you how many segments the blob has and the maximum segment length.

The calling program passes its request for information through the item list buffer. The access method returns the information to the result buffer. The calling sequence for the **gds_$blob_info** routine follows.

Syntax:

```
status = gds_$blob_info (status_vector,
blob_handle, item_list_buffer_length,
item_list_buffer, result_buffer_length,
result_buffer)
```

Arguments:

Parameter	Datatype	In/Out	Value/ Reference
status	long	out	
status_vector	vector_long	out	reference
blob_handle	ulong	in	reference
item_list_buffer_length	ushort	in	value
item_list_buffer	vector_byte	in	reference
result_buffer_length	ushort	in	value
result_buffer	unspec	out	reference

Table 12-5 lists the blob infromation calls and their meanings.

Table 12-5. Blob Information Items

Information Item	Meaning
gds_$info_num_segments	The number of segments that comprise the blob.
gds_$info_max_segments	The length of the longest segment in the blob.
gds_$info_total_length	The total length of the blob.
gds_$info_type	The blob sub-type.

The following call to the **gds_$blob_info** routine opens a blob and determines its vital statistics:

```
static char blob_items [] = {
      gds_$info_max_segment,
      gds_$info_number_segments,
      gds_$info_blob_type};

char  blob_info [32];

/* Open the blob and get its vital statistics */

if (gds_$open_blob (status_vector,
      DB,
      &gds_$trans,
      &blob,
      *blob_id))
  error ("gds_$open_blob failed", status_vector);

if (gds_$blob_info (status_vector,
      blob,
      sizeof (blob_items),
      blob_items,
      sizeof (blob_info),
      blob_info))
  error ("gds_$blob_info failed", status_vector);
```

Checking Active Transactions

The **gds_$transaction_info** returns information about the current transaction. A call to **gds_$transaction_info** returns information necessary for keeping track of permanent transaction ids.

The calling sequence for the **gds_$transaction_info** routine follows:

Syntax:

```
status = gds_$transaction_info (status_vector,
transaction_handle, item_list_buffer_length,
item_list_buffer, result_buffer_length,
result_buffer)
```

Table 12-6 lists shows the only information item for this routine.

Table 12-6. Transaction Information Items

Information Item	Meaning
gds_$info_tra_id	Transaction identifier.

The following call to **gds_$transaction_info** returns the identifier for a current transaction:

```
static char tra_items [] =
  {gds_$info_tra_id}

char tra_info [32];

if (gds_$transaction_info (status_vector,
      blob,
      sizeof (tra_items),
      tra_items,
      sizeof (tra_info),
      tra_info))
  error ("gds_$transaction_info failed", status_vector);
```

Using Blob Calls

The Blob Parameter Block

The following sections describe the:

- Blob Parameter Block (BPB)
- Options for the blob paramter block
- **gds_$open_blob2** routine to open a blob
- **gds_$create_blob2** routine to create a blob

On calls to **gds_$open_blob2** and **gds_$create_blob2**, you use messages to pass parameters to function calls. A parameter option is passed as a message using the blob parameter block (BPB). The BPB is a byte vector that starts with a one-byte version number (as specified by the compile-time constant **gds_$dbp_version1**) followed by zero or more options. Option codes are compile-time constants that you use in C or Pascal.

Each option contains an option code and length (which may be zero) followed by a value of the specified length (if appropriate).

The syntax for the BPB follows:

Syntax

```
bpb ::= version_number option_list-commalist
version_number ::= gds_$bpb_version1
option_list ::= option_code length value
```

BPB Options

The following is a list of parameter option codes for the BPB:

- gds_$bpb_source_type, which specifies the length in bytes of the blob subtype integer. The integer specifies the subtype a blob filters from. The length can be 1, 2 or 4 bytes depending on what integer you use for the subtype.
- gds_$bpb_target_type, which specifies the length in bytes of the blob subtype integer. The integer specifies the subtype a blob filters to. The length can be 1, 2 or 4 bytes depending on what integer you use for the subtype.

Opening a Blob

The **gds_$open_blob2** routine prepares a blob for retrieval and specifies the predefined subtypes for blob filtering. This call is equivalent to the **open blob** statement when you use it with the optional **from** *subtype* **to** *subtype* clause.

The calling sequence of the **gds_$open_blob2** routine follows.

Syntax:

```
status = gds_$open_blob2 (status_vector, db_handle,
transaction_handle, blob_handle, blob_id,
bpb_length, bpb)
```

Arguments:

Parameter	Datatype	In/Out	Value/ Reference
status	long	out	
status_vector	vector_long	out	reference
db_handle	ulong	in	reference
transaction_handle	ulong	in	value
blob_handle	ulong	in/out	reference
blob_id	uquad	in	reference
bpb_length	ulong	in	value
bpb	ulong	in	reference

Creating a Blob

The **gds_$create_blob2** routine creates a blob and specifies the predefined subtypes for blob filtering. This call is equivalent to the **create blob** statement when you use it with the optional **from** *subtype* **to** *subtype* clause.

The calling sequence of the **gds_$create_blob2** routine follows.

Syntax:

```
status = gds_$create_blob2 (status_vector,
db_handle, transaction_handle, blob_handle,
blob_id, bpb_length, bpb)
```

Arguments:

Parameter	Datatype	In/Out	Value/ Reference
status	long	out	
status_vector	vector_long	out	reference

Arguments:	Parameter	Datatype	In/Out	Value/ Reference
	db_handle	ulong	in	reference
	transaction_handle	ulong	in	value
	blob_handle	ulong	in/out	reference
	blob_id	uquad	in	reference
	bpb_length	ulong	in	value
	bpb	ulong	in	reference

For More Information

For more information about **gds** calls, see

- Chapter 8, *Using Blob Fields*
- Chapter 9, *Using Blob Filters*
- the *OSRI Guide*

Part III
Programming with SQL

Chapter 13
Getting Started with SQL

This chapter describes how to program with embedded SQL.

Overview

Language compilers in host operating environments don't directly support embedded SQL. Therefore, when you want to include SQL statements and commands in your program, you must do several things you don't have to do when accessing non-database data files:

1. Declare an area for communicating with InterBase.
2. Declare the host variables used for data transfer between the host language and InterBase.
3. Name the database you want to access
4. Embed the appropriate SQL statements and commands in your program.
5. Provide error recovery for your program.

6. Close the default transaction, when you are finished.
7. Preprocess the program before compiling it.

The remainder of this chapter discusses each of the steps required for the InterBase implementation of embedded SQL. This is followed by an SQL program example.

Declaring a Communications Area

The first step to preparing your program for embedded SQL is to declare the SQL communications area, or SQLCA. SQL programs communicate messages to the host program through the SQLCA by using a variable called SQLCODE.

As InterBase executes an SQL statement, it sets SQLCODE with a return code. You can check this return code for errors, as described later in this chapter.

You can declare the communications area in either of the following ways:

- Include the SQLCA:

```
/* map program */

#include <stdio.h>

exec sql
   include sqlca;
```

- Declare a section:

```
/* map program */

#include <stdio.h>

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

You must declare the communications area in the part of your program where you declare data. The position is language-specific. For more information, refer to Chapter 3, *Host Language Considerations.*

Declaring Host Variables

The SQL language structure requires the extensive use of host variables. Specifically, you need host 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.

As with a variable used in any program, you must declare it before you use it. Variables for use with embedded SQL statements have the same form as other variables declared in your host language. For example, the following C variable declarations are for use with corresponding database fields:

```
char state [3],
     state_name [26],
     capital [26];
```

For all host languages except BASIC, you can define a host structure to correspond to some collection of database fields. For example, you may decide to map address fields in a record to a host structure called BILLING_ADDRESS. Even though the destination is a structure that has the same format and field names as a database record, you must assign field values one at a time.

Naming the Database

You can access only one database at a time when you use pure embedded SQL statements. SQL requires that you declare this database at the time of program binding.

In this implementation of SQL, you declare the database when you preprocess your program with **gpre**. Instructions for preprocessing your program are found later in this chapter and in Chapter 21, *Preprocessing Your Program*.

Because the target database is not specifically named in the program, you should include a comment in the source program to identify the database. For example:

```
/*  This program uses the atlas.gdb database  */
```

Gpre includes the name of the target database when it processes the program, so you can find the declaration in the post-processed program. However, if you change the program, modify the original source program rather than **gpre**'s output. The original source should be as complete and self-documenting as possible.

If you want to access multiple databases, you can mix SQL statements with GDML statements. For information on mixing SQL with GDML statements, refer to Chapter 17, *Mixing SQL with Other Interfaces*.

Embedding SQL Statements

You can embed the following SQL statements in your program:

- A **select** statement, which allows you to select data, project data, join tables, and append tables
- An **insert** statement, which allows you to store data into a table
- An **update** statement, which allows you to modify rows in a table
- A **delete** statement, which allows you to delete rows from a table
- The following SQL metadata commands:
  - **create database**, which allows you to delete databases
  - **create table**, **alter table**, and **drop table,** which allow you to create, modify, and delete tables
  - **create index** and **drop index**, which allow you to create and delete indexes
  - **create view** and **drop view**, which allow you to create and delete views
  - **grant** and **revoke**, which allow you to give access to and restrict access to tables
- The following cursor operation statements, which allow you to retrieve multiple rows:
  - declare cursor
  - open
  - fetch
  - close

When you embed SQL statements, you must:

- Preface each SQL statement with the keywords **exec sql**.
- End the statement with whatever statement terminator your host language requires.

For example:

```
exec sql select state, state_name, capital into
  :statecode, :name, :cap_city from states
  where state = :input_variable;
```

Note

The semicolon is used here as a generic end-of-statement symbol. You should supply the symbol required by your language.

Handling Errors

You should always provide error recovery for your SQL programs. As mentioned earlier, InterBase uses the SQLCODE variable to pass messages to an SQL program. SQLCODE can pass the following messages:

- **not found** indicates end of file. This condition corresponds to an SQLCODE value of 100.
- **sqlerror** indicates the statement did not complete. This condition corresponds to an SQLCODE with a negative value.
- **sqlwarning** indicates a general system warning or informational message. This condition corresponds to an SQLCODE with a value between 1 and 99, inclusive.
- **0** indicates successful completion.

There are three ways to provide error recovery in an SQL program:

- Use the **whenever** statement to check the value of SQLCODE.
- Check the value of SQLCODE directly.
- Use the **gds_$print_status** routine to test for specific errors.

These methods are discussed below, followed by a discussion of error handling considerations.

Using the Whenever Statement

You can use the **whenever** statement to check the value of SQLCODE. This statement automatically checks the value of SQLCODE after each SQL statement completes. You need to include one **whenever** statement in each module.

For example, this **whenever** statement executes logic that prints SQLCODE messages:

```
exec sql
  whenever sqlerror go to Label_999;

Label_999: if (SQLCODE)
  print_error();
}
/*********************************************************

    Print out error message.

*********************************************************/
print_error()
```

```
{
  printf ("Database error, SQLCODE = %d\n", SQLCODE);
}
```

Testing SQLCODE Directly

You can test the SQLCODE directly after each SQL statement. For example, these statements check for an SQLCODE of either -1 or 100 after the **select** statement completes:

```
exec sql
  select city into :city from s
      where state = :stat:statind;

if (SQLCODE)
    {
    if (SQLCODE == -1)
        printf ("too many records found\n");
    else if (SQLCODE == 100)
        printf ("no records found\n");
    else
        gds_$print_status (gds_$status);
    exit (0);
  }

printf ("found city named %s\n", city);

    exec sql
        rollback release;
}
```

Testing for Specific Errors

A single SQLCODE can represent multiple InterBase errors. (For example, the SQLCODE -901 represents 37 different InterBase errors.) To test for these specific errors, you must check the actual message returned by InterBase. You do this by using the **gds_$print_status** routine.

Note

The SQLCODEs 0, 100, and -1 have no equivalent **gds_status** error. Calling **gds_$print_status** with any of these values results in an incomprehensible message.

For example:

```
exec sql
  whenever sqlerror go to Label_999;

Label_999: if (SQLCODE ==  1)
  printf ("too many records matched \n");
else
  print_error();
}
/*********************************************************

    Print out error message.

*********************************************************/
print_error()
{
  printf ("Database error, SQLCODE = %d\n", SQLCODE);
  gds_$print_status (gds_$status);
}
```

Considerations for Handling Errors

The following considerations apply to handling errors in an SQL program. If you plan to:

- Move your SQL program between InterBase and other database management systems, you should do either of the following:
  - Limit yourself to checking the SQLCODE message. This preserves the portability of your program.
  - Call **gds_$print_status** from a single routine that you can replace with system-specific error handling when you port your program. This preserves the portability of your program without sacrificing the additional error information that the **gds_$print_status** routine provides.
- Use your program on InterBase or on other DSRI database management systems, you should check both the SQLCODE message and the actual message returned by InterBase. This gives you more specific error information.

For a detailed listing of SQLCODEs and InterBase error messages, refer to the appendix on error handling in the *Programmer's Reference*.

Note

For the sake of brevity, most examples in this document omit general error handling.

Closing the Default Transaction

You can use only one transaction per program when you use embedded SQL statements. SQL does not give you the capability of starting a transaction. Instead, it automatically starts a default transaction for you when you first use an SQL statement.

When you are finished with the default transaction, you should close it with a **commit** or **rollback** command:

- The **commit** command causes any changes you made to be written to the database and thus made available to other users:

```
exec sql
   commit;
```

- The **rollback** command undoes any changes you made:

```
exec sql
   rollback;
```

The **release** option on the **commit** and **rollback** commands releases all of the resources used by the database attachment. Specifically, this option:

- Closes the database files
- Closes the connection for remote databases
- Releases the memory that holds the database metadata descriptions and the compiled requests

For example, to specify this option on the **commit** command, type:

```
exec sql
  commit release;
```

You should always use the **release** option when you have finished using a database. This makes the system resources used by your program available to other users. Don't use this option if you haven't finished using a database, because you will have to reacquire the released resources.

If you want the ability to start transactions explicitly, you can mix SQL statements with GDML statements and/or OSRI calls. For information on mixing SQL with GDML statements and OSRI calls, refer to Chapter 17, *Mixing SQL with Other Interfaces*.

Preprocessing Your Program

After you code your SQL program, you must use **gpre** to preprocess the program. **Gpre** translates SQL statements and database variables into statements and variables that the host language compiler accepts. **Gpre** also has a database option (**-d**). This option takes the file specification that follows it as the database declaration.

The example below shows how to use **gpre** to preprocess a C program containing embedded SQL statements. In this example, the file *geo_survey.e* is preprocessed for use with the database stored in *atlas.gdb*:

```
gpre geo_survey.e -d atlas.gdb
```

As it preprocesses the program, **gpre** supplies the database declaration and several other declarations the host language compiler expects to find.

For more information on preprocessing programs with **gpre**, refer to Chapter 18, *Preprocessing Your Program*.

SQL Example

The following SQL program declares a communications area and host variables, embeds SQL statements, checks for errors, and closes the default transaction:

```
/* program mapper */
/*  This program uses the atlas.gdb database  */

exec sql
     include sqlca;

char  statecode[5];
char  cityname[26];

exec sql
  whenever sqlerror go to Label_999;

main()
{
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
    commit release;

Label_999: if (SQLCODE)
```

```
    print_error();
}
/************************************************************

    Print out error message.

************************************************************/
print_error()
{
    printf ("Database error, SQLCODE = %d\n", SQLCODE);
    gds_$print_status (gds_$status);
}
```

For More Information

For more information on programming with embedded SQL, refer to:

- Chapter 14, *Retrieving Data with SQL*, for information on retrieving data with embedded SQL.
- Chapter 15, *Writing Data with SQL*, for information on storing, modifying, and deleting data with embedded SQL.
- Chapter 16, *Defining Metadata with SQL*, for information on defining, modifying, and deleting metadata with embedded SQL.
- Chapter 17, *Mixing SQL with Other Interfaces*, for information on mixing SQL with GDML and OSRI calls.
- Chapter 18, *Preprocessing Your Program*, for information on using **gpre** to preprocess your SQL program.
- The chapter on SQL statements in the *Programmer's Reference*, for information on specific SQL statements and commands and a list of InterBase error messages.

Chapter 14
Retrieving Data with SQL

This chapter describes how to retrieve data by using embedded SQL.

Overview

Retrieving data through embedded SQL involves four relational operations. You use the:

- Select operation to retrieve specific rows from a table.
- Project operation to extract specific columns from a table.
- Join operation to combine two or more tables to form a third table. You join tables based on columns that have common values.
- Union operation to append two or more tables together to form a third table.

The remainder of this chapter discusses how to perform each of these relational operations.

Selecting Data

You use the **select** statement to retrieve data in embedded SQL. Instructions for selecting a single table row, selecting multiple table rows, specifying search criteria, selecting rows with missing values, and selecting rows through a view are presented below.

Selecting a Single Row

You can use a simple version of the **select** statement (that is, a **select** without a cursor) when you are sure that the search conditions you specify will return no more than a single row. For example, you might want to use this type of **select** statement to select a column defined with a unique index, or to select an aggregrate value like **count** or **avg**.

Once you establish whether you can use a simple **select** statement, you need to construct a search condition for the row you want. You construct a search condition by using the **select** statement's **where** clause. This clause contains a *Boolean expression*, which evaluates to true, false, or missing when you select a row. If it evaluates to true, the row is returned.

For example, to select information from the STATES table for the state called 'NY', you could type:

```
exec sql
  select state, state_name, capital
      into :statecode, :statename, :city
      from states where state eq "NY";
```

When SQL returns the selected row, it moves the values of STATE, STATE_NAME, and CAPITAL for that row into the host variables :STATECODE, :STATENAME, and :CITY.

If you don't construct a search condition for the example above, SQL returns *all* rows from the table you selected. This causes an error because SQL expects to find only one row when you use this type of **select** statement. In this case, the SQLCODE is set to -1.

Considerations for Selecting Data

You should be aware of the following considerations when you select data:

- Each host variable in your SQL program must be prefaced with a colon (:). This convention alerts SQL to the fact that the target variables of the **into** clause are not database columns.

- The order of items in the **into** clause must correspond to the order of items listed in the **select** statement. Otherwise, you risk either overflow or data conversion problems when you read database column values into those variables.

Selecting Multiple Rows

If you are unsure of the number of rows that satisfy a particular search condition, you can't use the single-row **select** approach. Instead, you must ask that qualifying rows be placed in a temporary, table-like collection called a *results table.*

Because the results table can contain more than one row, SQL provides a pointing device called a *cursor.*

To select rows using a cursor, follow these steps:

1. Declare the cursor.
2. Open the cursor.
3. Fetch rows from the cursor.
4. Close the cursor.

Declaring the Cursor

You declare a cursor by using the **declare cursor** statement. This statement names a cursor and specifies which columns and rows to return. Because the **declare cursor** statement is non-executable, you need not check for SQLCODE when you use this statement.

For example, the following statement declares a cursor for all cities with a population greater than 100,000:

```
exec sql
   declare big_cities cursor for
      select city, state, population
      from cities where population gt 100000;
```

You can sort the row stream that gets returned by using the **order by** clause. For example, to sort the BIG_CITIES cursor output by state and then city, you could type:

```
exec sql
   declare big_cities cursor for
      select city, state, population
      from cities where population gt 100000
      order by state,cities;
```

The **order by** clause automatically sorts in ascending order. To sort the states in ascending order and the cities in descending order, you could type:

```
exec sql
   declare big_cities cursor for
      select city, state, population
      from cities where population gt 100000
      order by state,cities desc;
```

Opening the Cursor

To access rows selected by a cursor, you must open the cursor. The **open** statement activates the cursor and builds the results table. It builds this table based on the selection criteria you specified in the **declare cursor** statement. The rows in the results table comprise the *active set* of the cursor.

For example,the following statement opens the BIG_CITIES cursor that you declared above:

```
exec sql
  open big_cities;
```

When InterBase executes the **open** statement, it leaves the cursor pointing to a position before the first row in the results table.

Fetching Rows with the Cursor

Now that the cursor is open, you can retrieve rows from its active set by using the **fetch** statement. Your first **fetch** statement advances the cursor to the first row in the active set. Subsequent **fetch** statements advance the cursor through the active set, one row at a time.

In both the initial access and subsequent reads, you must retrieve the columns into host variables or structures by using the **into** clause:

```
exec sql fetch big_cities
  into :cityname, :statecode, :census;
```

If you want to loop through the rows selected by the cursor, you can enclose the **fetch** statement in a looping construct:

```
exec sql
  fetch big_cities into :cityname, :statecode, :census;

printf("\n");
while (!SQLCODE) {
```

```
    printf ("%s is in %s and %d people live there\n",cityname,
statecode, census);
    exec sql
        fetch big_cities into :cityname, :statecode, :census;
    }
```

Because cursors select finite numbers of rows, you should include some end-of-file handling in your program. The example above checks for the end-of-file by testing whether a non-zero SQLCODE is returned. You should also provide error handling tied to the specific condition indicated by SQLCODE's value.

Note

The only supported cursor movement is forward. If you reach the end of the active set of a cursor and want to walk through the rows of the cursor again, you must close the cursor and then re-open it with another **open** statement.

Closing the Cursor

Once you have reached the end of the cursor's active set, you should close the cursor to free up all associated resources:

```
exec sql
  close big_cities;
```

Example of Selecting Multiple Rows

The following program declares a cursor, opens the cursor, and then loops through the cursor's active set, fetching and printing values. The program closes the cursor when all processing is finished:

```
/* mapper program */

#include <stdio.h>

exec sql
  include sqlca;

exec sql
  begin declare section;
exec sql
  end declare section;

char statecode[4];
```

```
char cityname [26];
int  census;

main ()
{

exec sql
  whenever SQLERROR go to abend;

exec sql
  declare big_cities cursor for
      select city, state, population from cities
      where population > 1000000;

exec sql
  open big_cities;

exec sql
  fetch big_cities into :cityname, :statecode, :census;

printf ("\n");
while (!SQLCODE) {
  printf ("%s is in %s and %d people live there\n",cityname,
statecode, census);
  exec sql
      fetch big_cities into :cityname, :statecode, :census;
  }

exec sql
  close big_cities;

abend:

if ( (SQLCODE) && (SQLCODE != 100)) {
  printf ("Encountered SQLCODE %d\n", SQLCODE);
  printf ("  expanded error message -\n");
  gds_$print_status (gds_$status);
  exec sql
      rollback release;
  }
else
  exec sql
```

```
        commit release;
}
```

Specifying Search Criteria

When you specify a search condition, InterBase evaluates the condition for each row that might possibly qualify. InterBase compares the value you supplied with the value in the database field you specified. If the two values satisfy the relationship indicated by the operator you specified (for example, "equals"), the search condition is true and that record becomes part of the cursor's active set.

For example, the following declaration selects only those rows for which it is true that the city is located in California:

```
exec sql declare quakecities cursor for
  select city, latitude, longitude from cities
  where state = 'CA';
```

This search condition results in a value of "true," "false," or "missing" for each row in the CITIES table. This type of expression is called a *Boolean expression*.

If your query has more than one search condition, such as cities with a population greater than 1,000,000 and at an altitude above 200 feet, the truth of the Boolean test depends on the combination of search conditions.

In the example below, SQL returns only those CITIES rows for which both conditions are true:

```
exec sql declare big_and_high cursor for
  select city, state, latitude, longitude from
  cities where population > 1000000 and altitude > 200;
```

Using the Not, And, and Or Operators

You can use the **not**, **and**, and **or** logical operators when you specify search conditions. SQL evaluates these operators in the order listed above, except when you put an expression in parentheses. The use of parentheses changes the order of evaluation, because SQL evaluates the inside parentheses first.

Tables 14-1, 14-2, and 14-3 show truth tables for these operators. Note that the value of B is irrelevant for the outcome of the **not** operator.

*Table 14-1. The **Not** Operator*

Value of A	Value of B	Not A
True	True	False
True	False	False
True	Missing	False
False	True	True
False	False	True
False	Missing	True
Missing	Missing	Missing

*Table 14-2. The **And** Operator*

Value of A	Value of B	A and B
True	True	True
True	False	False
True	Missing	Missing
False	True	False
False	False	False
False	Missing	False
Missing	Missing	Missing

*Table 14-3. The **Or** Operator*

Value of A	Value of B	A or B
True	True	True
True	False	True
True	Missing	True
False	True	True
False	False	False
False	Missing	Missing
Missing	Missing	Missing

Using Other SQL Operators

In addition to the standard equality, inequality, and range operators, you can use:

- **like** to test for the presence of a string with wildcards in a scalar expression. The percent sign (%) wildcard matches a run of characters, and the underscore (_) matches a single character. The **like** operator is case-sensitive.

  For example, this cursor returns information about cities that contain the letters "ton":

```
exec sql
   declare ton_cities cursor for
      select city, state, population
      from cities
      where city like "%ton%";
```

  To search for a substring that includes a percent or underscore character, use the **escape** clause with the **like** operator. The escape clause lets you define a special character that tells InterBase to treat the next character literally.

  For example, this cursor retrieves the names of all cities from the CITIES relation, because the percent character matches all names:

```
exec sql
   declare all_cities cursor for
      select city from cities
      where city like "%";
```

This cursor retrieves no records, because it specifies that the percent character be treated literally rather than as a wildcard, and no city is named %:

```
exec sql
   declare no_cities cursor for
      select city from cities
      where city like "@%" escape "@";
```

For more information on defining escape characters, refer to the description of the **select** statement in the SQL chapter of the *Programmer's Reference*.

- **exists** to test for the existence of at least one qualifying row in one or more tables. The expression is true if the row stream specified by the select expression includes at least one row.

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

  For example, this cursor returns all states that have ski areas:

```
declare any_skiareas cursor for
      select state_name
      from states s
      where exists (select *
         from ski_areas where state = s.state);
```

- **null** to test for the absence of a value in a database field expression.

  For example, this cursor returns all states with no capitals:

```
exec sql
   declare missing_capital cursor for
      select state_name
      from states
      where capital is null;
```

- **in** to test for a value that equals at least one value in a value list. The values in the value list must be parenthesized and separated by commas.

  For example, this cursor returns information on all cities in a specified list of states:

```
exec sql
   declare city_info cursor for
      select city, latitude, longitude from cities
      where state in ( CT, RI, MA, NH, VT, ME );
```

  You can also use the **in** operator to compare a value against the results of a subquery. Subqueries are described later in this chapter.

- **all** to compare a value to a value list returned by a subquery. The expression is true only if the comparison is true for all values in the list.

  For example, this cursor returns information on rivers that are longer than all rivers in British Columbia:

```
exec sql
   declare longest_river cursor for
      select river, source, length from rivers where
      length > all (select length from rivers where
      source = "BC")
```

- **any** to compare a value to a value list returned by a subquery. The expression is true if the comparison is true for at least one value in the list.

  For example, this cursor returns information on rivers that are longer than at least one river in British Columbia:

```
exec sql
   declare long_river cursor for
      select river, source, length from rivers where
      length > any (select length from rivers where
      source = "BC");
```

- **some** to compare a value to a value list returned by a subquery. This operator yields the same results as the **any** operator. The expression is true if the comparison is true for at least one value in the list.

  For example, this cursor also returns information on rivers that are longer than at least one river in British Columbia:

```
exec sql
   declare long_river cursor for
      select river, source, length from rivers where
      length > some (select length from rivers where
      source = "BC");
```

Searching for Groups of Rows

You can search for groups of rows by using the **group by** clause in conjunction with the **having** clause. The **group by** clause groups rows together The **having** clause then operates on the groups of rows in the same way that **where** operates on individual rows. You provide a search condition or combination of search conditions. InterBase then evaluates the condition for each group of rows (for example, all CITIES rows with a value of "NY" for the STATE column).

The following example limits the cities considered to those that fall in the specified geographical band. Then it separates them into groups and chooses only those groups with larger than average populations:

```
exec sql
  declare avg_pop_mid_america cursor for
  select avg (population), state
      from cities
      where latitude_degrees between 33 and 42
          and longitude_degrees between 79 and 104
      group by state
      having avg (population) > 40000;
```

When you specify a search condition, InterBase evaluates the condition for each group of rows that might possibly qualify. InterBase compares the value you supplied with the value in the database field you specified. If the two values satisfy the relationship indicated by the operator you specified (for example, "equals"), the search condition is true and that rows become part of the cursor's active set. As with the **where** selection demonstrated earlier in this chapter, the search condition results in a value of "true", "false", or "missing" for each row.

Having eliminates groups of rows, while **where** eliminates individual rows. You can usually use subqueries to obtain the same results that you get with the **having** clause. The main advantage of using the **having** clause is brevity. However, some users may find that a subquery is easier to understand. Subqueries are discussed later in this chapter.

Selecting Rows with Missing Values

InterBase lets any column have a missing (or null) value. Rather than storing a value for the column, InterBase sets a flag indicating the column has no assigned value. Unless you specify otherwise in the column's definition, InterBase returns zero for numbers, blanks for characters, and 17 November 1858 for dates.

For example, many of the rows in the CITIES table do not have a value for the POPULATION column. Cities with no stored population have the missing value flag set for that column.

The following **select** statement lists cities with populations less than 200,000 or for which the population is missing:

```
exec sql
  declare mid_sized_cities cursor for
      select city, state, population
      from cities
```

```
        where population < 200000 or population is null
        order by population;
```

In this example, the missing value sorts last, even though the sorting order defaults to ascending. This is because the missing value is not really a value, but instead, a flag indicating that the column value is different from actual stored values.

If you need to know whether or not a column has a missing value, use an *indicator parameter*. If the value retrieved is:

- Missing, InterBase sets the indicator parameter to -1.
- Not missing, InterBase sets the indicator parameter to 0.

For example, the following **select** statement prints the values of the CITY and POPULATION columns. If the POPULATION column contains a missing value, the program prints the word "MISSING":

```
exec sql begin declare section;
exec sql end declare section;
main ()
{
char      city [20], pop [20];
long      population;
short     pi;

exec sql
    declare c cursor for select city, population
  from cities where state = "MA" order by city;
exec sql
    open c;
exec sql
    fetch c into :city, :population indicator :pi;
while (!SQLCODE)
    {
    sprintf (pop, "%d", population);
    printf ("%20s\t%s\n", city, (pi) ? "MISSING" : pop);
    exec sql
  fetch c into :city, :population indicator :pi;
    }
if (SQLCODE != 100)
    gds_$print_status (gds_$status);

exec sql rollback;
}
```

In addition to the sorting order, missing values have the following special considerations that result from their status as non-values:

- If you perform a statistical operation involving a column that is missing for a given row, InterBase ignores that row in the computation.
- A missing value never equals another value, even if the other value is also missing. Therefore, you can't use a missing value in a comparison.
- A missing value doesn't appear as the result of a negated test.
- A column with a missing value can't satisfy a join condition.

For information about defining alternate missing values, refer to the chapter on defining fields in the *Data Definition Guide*.

Selecting Rows Through a View

You select rows through a view in the same way that you select rows through a stored table. For example, the following cursor selects rows from the POPULATION_DENSITY view:

```
exec sql
  declare n_states cursor for
      select state, density_1950, density_1960
      from cities
      where state like '%N%';
```

You can also:

- Join views with stored tables and other views
- Update through a view, under certain circumstances

For information on updating through views, refer to Chapter 15, *Writing Data with SQL*. For information on creating views, refer to Chapter 16, *Defining Metadata with SQL*.

Projecting Columns

You can retrieve a subset of columns from a table by using the relational *project* operation. By projecting columns, you can reduce the size of a row returned to you in a results table.

You specify which subset of columns to retrieve by using a *comma* list in the **select** statement. For example, to retrieve a subset of columns from the STATES table, type:

```
exec sql
      select state, state_name, capital
          into :statecode, :statename, :city
          from states;
```

This returns the STATE, STATE_NAME, and CAPITAL columns for all rows in the STATES table.

To retrieve *all* columns from a table, use an asterisk (*) in the **select** statement:

```
exec sql
      select *
          into :statcode, :statename, :area, :date, :city
          from states;
```

Note

The **select** * form of the **select** statement is not recommended, except for use with the **exists** operator. Adding and dropping columns from a table selected with this syntax can result in program errors.

When you ask for a table projected on one or more columns, you can use the **distinct** option to eliminate rows that do not have a unique combination of values for the listed columns. For example, the following **select** statement returns only one occurrence of a value for the STATE column and ignores subsequent duplicates:

```
exec sql
  declare unique_states cursor for
      select distinct state
          from river_states;
```

Joining Tables

You can create dynamic relationships by matching rows from two or more tables in the same database. There are many types of joins you can make by using embedded SQL. The more common joins are discussed below:

- An *inner join* is the standard join. This join combines rows from one table with rows from another based on a common predicate. An inner join can be an equi-join or a nonequi-join:
  - An *equi-join* joins rows from two or more tables based on a common value or an equality relationship in the join column.

    For example, to return city and state information for all cities whose state exists in the STATES table, you could join the CITIES and STATES tables where the STATE columns have equal values.
  - An *unequi-join* joins rows from two or more tables based on a non-equality relationship in the join column. This type of join is used less frequently than the equi-join.

    For example, to return a listing of all Canadian provinces that are larger than Alaska, you could join the STATES table to the PROVINCES table where the AREA column in the PROVINCES table is larger than the AREA column in the STATES table, and the STATE_NAME column equals 'Alaska'.
- An *outer join* is an extended inner join. In an outer join, *all* rows are returned, whether or not they have a counterpart in the other specified tables.

  For example, you could produce an outer join that joins the CITIES and STATES tables. This join returns all rows in the CITIES tables, whether or not there is a corresponding row in the STATES table. It also returns all rows in the STATES table, whether or not there is a corresponding row in the CITIES table.

  You could also produce a left or right outer join of the CITIES and STATES tables. A left outer join returns all rows in the CITIES table, whether or not there is a corresponding row in the STATES table. A right outer join returns all rows in the STATES table, whether or not there is a corresponding row in the CITIES table.

  Although the InterBase version of SQL does not directly support the outer join, you can simulate an outer join by using the **union** statement. An example of this is shown below, under *Join Examples*.
- A *reflexive join*, or *self join*, is a join in which a table is joined to itself.

  For example, you could return information on each river and the rivers it flows into by joining the RIVERS table to itself. In this case, you would match the RIVER column to the OUTFLOW column.

Examples of these types of joins are presented in the following sections after a discussion of aliases.

Using Aliases

Whenever you specify columns from different sources that have the same name in a join, you must either qualify the column names with the associated table name or use aliases. It's good practice to qualify *all* column names in a join, whether or not the column names are unique.

For example, to join the CITIES and STATES tables on the STATE column, you could qualify each column with its associated table name:

```
exec sql
  declare city_state_join cursor for
      select cities.city, state.state_name, cities.latitude,
       cities.longitude from cities, states
        where cities.state = states.state
        order by states.state, cities.city;
```

You can also use *aliases* to qualify a column name. An alias is a temporary variable that represents a table name. It can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_), but must always start with an alphanumeric character. The use of aliases can significantly reduce the keystrokes involved in a complicated join.

To use aliases in the above example, type:

```
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;
```

Note

Gpre is sensitive to the case of the alias in a C program, unless you specify the **either_case** option when you preprocess your program. **Gpre** is not sensitive to the case of the alias in other programs. For example, it would treat C and c as the same character in non-C programs.

Join Examples

Examples of different types of joins follow.

Example 1 — Joining Two Tables with an Equi-Join

The following cursor joins the CITIES and STATES tables on the STATE column:

```
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;
```

Example 2 — Joining More than Two Tables with an Equi-Join

The following cursor joins the CITIES, STATES, and TOURISM tables. It returns the name, state name, and zip code of every city that has a tourism office:

```
exec sql
  declare city_state_tourism cursor for
      select c.city, s.state_name, t.zip
        from cities c, states s, tourism t


where c.state = s.state
          and s.state = t.state
          and t.city = c.city
      order by s.state;
```

Example 3 — Joining Two Tables with an Unequi-Join

The following cursor joins the STATES and PROVINCES tables in an unequi-join. This query returns information on all provinces that are larger than Alaska and the comparable information on Alaska:

```
exec sql
  declare large_provinces cursor for
      select s.state_name, s.area, p.province_name, p.area
        from states s, provinces p
        where p.area > s.area
          and s.state_name eq 'Alaska';
```

Example 4 — Producing a Left Outer Join

The following cursor simulates a left outer join for the STATES and RIVERS tables. It selects those states that are the source of rivers. It then selects those states that are not the source of rivers. It returns a list of states matched with their associated rivers, followed by a list of states matched with the words "No river":

```
exec sql
  declare x cursor for
  select s.state_name, r.river
      from states s, rivers r where s.state = r.source
  union
  select s2.state_name, "No river"
      from states s2 where not exists (select * from rivers r2
         where s2.state = r2.source);
```

This join involves the use of subqueries and the **union** statement. Subqueries and the **union** statement are discussed later in this chapter.

Example 4 — Producing a Reflexive Join

The following cursor joins the RIVERS table to itself in order to find out which rivers flow into other rivers:

```
exec sql
   declare reflexive_rivers cursor for
       select r1.river, r2.river
          from rivers r1, rivers r2
         where r1.river = r2.outflow
          order by r1.river, r2.source;
```

Appending Tables

SQL also supports *union*, an operation that creates dynamic tables by appending tables. Unlike the join operation, union does not match rows from one table to rows from another. Instead, it concatenates one table to the other. The resulting dynamic table contains columns from each of the source tables. The union operation automatically eliminates duplicate rows.

The union operation is useful when you have similar data in different tables, but no natural way to "roll up" the information into one collection. For example, three tables, SKI_AREAS, CITIES, and STATES, each contain names of cities. The sample database *as defined* does not contain an integrity constraint that automatically stores a new row in CITIES for each new SKI_AREAS or STATES row with a capital city.

Note

In most cases, you would define a trigger to maintain data integrity. For information on defining triggers, refer to the chapter on preserving data integrity in the *Data Definition Guide*.

However, you can still collect all cities in one place with the union operation. For example, the following cursor selects all cities from the CITIES, SKI_AREAS, and STATES table:

```
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;
```

Unions are commonly used to perform aggregates on multiple tables.

Using Statistical and Aggregrate Functions

You can use SQL to calculate and return:

- The average value of a column (**avg**)
- The minimum value stored in a column (**min**)
- The maximum value stored in a column (**max**)
- The sum of values stored in a column (**sum**)
- The number (**count**) of rows that satisfy the select expression

In each of the statistical operations, SQL calculates a value based on every qualifying row in a stream. For example, the following program returns the average length of all rows in the RIVERS table:

```
exec sql begin declare section;
exec sql end declare section;
main ()
{
double avg_riv;

exec sql
    select avg (length)
  into :avg_riv
  from rivers;

if (!SQLCODE)
    printf ("average river length %g\n", avg_riv);

exec sql commit release;
}
```

This program returns a count of rows in the CITIES table, as well as the maximum population and the minimum population of cities in that table:

```
exec sql begin declare section;
exec sql end declare section;
main ()
{
int counter, minpop, maxpop;

exec sql
    select count (*), max (population), min (population)
  into :counter, :minpop, :maxpop
  from cities;
```

```
if (!SQLCODE)
    {
    printf ("Count: %d\n", counter);
    printf ("Max Population: %d\n", maxpop);
    printf ("Min Population: %d\n", minpop);
    }

exec sql commit release;
}
```

Note

If you are programming in Pascal, you must leave a space between the open parenthesis and the asterisk in the **count** function. Because Pascal uses the sequence "(*" for comments, failure to leave a space results in a compilation error.

If a column value involved in a statistical calculation is missing, that row is automatically excluded from the calculation, rather than being figured as zero. This exclusion prevents averages from being skewed by spurious zeros or inappropriate values.

You can also perform statistical operations on groups of rows. Such operations yield *group aggregate* values. For example, you may be interested in the average population of all cities in a particular state.

The following program demonstrates the use of grouping, aggregates, and indicator variables. Several states have no cities with valid populations. To avoid printing zero as their average population, the program tests a special indicator, POP_IND, that is negative when the average does not reflect valid data:

```
exec sql begin declare section;
exec sql end declare section;
main ()
{
double avg_pop;
int    pop_ind;
char   state_code[5];

exec sql
    declare avg_pop_by_state cursor for
     select avg (population), state
       from cities
       group by state;

exec sql
```

```
    open avg_pop_by_state;

exec sql
    fetch avg_pop_by_state into :avg_pop :pop_ind, :state_code;

while (!SQLCODE)
    {
    if (pop_ind >= 0)
  printf ("Average population of cities in %s is %g\n",
      state_code, avg_pop);
    else
  printf ("Cities in %s are unpopulated according to our
data\n",
       state_code);

    exec sql
        fetch avg_pop_by_state into :avg_pop :pop_ind,
:state_code;
    }

exec sql commit release;
}
```

InterBase calculates the average population of each state as follows:

1. It separates the rows into groups for each state. These groups are called *control groups*.
2. It computes the aggregate expression for each control group. In this case, it returns the average population of cities in "NY", "MA", and so on.

You can also compute an aggregate value in the *select-clause* and the *having-clause* of the *select-expression*.

Using Subqueries

Some queries are impossible to express as views. For example, you can't get a list of states with a larger than average area by joining states to itself. However, you can easily get that list with a subquery:

```
exec sql
  select state_name into :bigstate from states where
      area > (select avg (area) from states);
```

The preceding example demonstrates a simple, one-level subquery. None of its values depend on values in the main query. Often, values in a subquery do depend on values from the main query. Suppose you wanted a list of states for which there were three or more cities stored in the CITIES table:

```
exec sql
  declare lotsa_cities cursor for
      select state_name from states
      where 3 <= (select count (*) from
      cities where cities.state = states.state);
```

The preceding example is a *correlated subquery*, so called because of the correlation between the subquery and its parent.

You can nest and mix simple and correlated subqueries to build complex requests. The following query prints the state name, capital, and largest city of states whose area is larger than the average area of states that have at least one city within 100 feet of sea level:

```
exec sql
  declare sealevel cursor for
      select s1.state_name, s1.capital, c1.city from
      states s1, cities c1 where
      s1.state = c1.state and
      c1.population = (
          select max (population) from
          cities c2 where c2.state = c1.state) and
              s1.area > (
          select avg (s2.area) from
          states s2 where exists ( select *
              from cities c3 where
          c3.state = s2.state and
          c3.altitude <= 100 ))
```

As tables re-appear in subqueries, aliases are the only way to identify a column with the appropriate instance of its table.

Retrieving Special Types of Columns

Because host langauges do not directly support InterBase's date datatype, and SQL does not support the blob or array datatype, you must do the following if you intend to retrieve data from columns that have these datatypes:

- For dates, use the following **gds** call interface routines:
  - — The **gds_$decode_date** routine, to convert the InterBase internal date format to the UNIX time structure
  - — The **gds_$encode_date**, to convert the UNIX time structure to the internal InterBase date format

  These routines are described in Chapter 10, *Using Date Fields*.
- For blobs, use GDML blob handling statements, GDML library routines, or the **gds** call interface routines. These statements and routines are described in Chapter 8, *Using Blob Fields*.
- For arrays, use the GDML **for** loop. This statement is described in Chapter 5, *Retrieving Data with GDML*. Detailed information on using arrays is presented in Chapter 7, *Using Array Fields*.

For more information on mixing SQL and GDML, refer to Chapter 17, *Mixing SQL with Other Interfaces*.

For More Information

For more information on retrieving blob, date, and array data, refer to Part II, *Programming with GDML.*

For more information on using embedded SQL, refer to:

- Chapter 13, *Getting Started with SQL*, for introductory information on using embedded SQL.
- Chapter 15, *Writing Data with SQL*, for information on storing and modifying data with embedded SQL.
- Chapter 16, *Defining Metadata with SQL*, for information on defining metadata with embedded SQL.
- Chapter 17, *Mixing SQL with Other Interfaces*, for information on mixing SQL and GDML.
- Chapter 18, *Preprocessing Your Program*, for information on using **gpre** to preprocess your SQL program.
- The chapter on SQL statements in the *Programer's Reference*, for information on specific SQL commands and statements.

Chapter 15
Writing Data with SQL

This chapter describes how to store, modify, and delete data by using embedded SQL.

Overview

You can use the following SQL statements to store, modify, and delete data:

- **insert**, which allows you to insert a new row into a table
- **update**, which allows you to modify one or more rows in a table
- **delete**, which allows you to delete one or more rows from a table

Storing Data

You use the **insert** statement to insert a new row into a table. This statement consists of a list of database columns and a **values** list that provides the values you want to store in those columns.

The following statement stores a new row in the SKI_AREAS table:

```
exec sql
   insert into ski_areas (name, city, state, type)
      values ('Flints Corner', 'Tyngsboro', 'MA', 'B');
```

Instructions for assigning column values, storing rows with missing column values, storing special types of columns, and storing through a view are presented below.

Assigning Column Values

The source of values for an assignment can be any combination of the following:

- Scalar expressions
- Host variables
- Values from a subquery

Using Scalar Expressions

You can assign values to a new row by using one or more scalar expressions. These can be quoted literal expressions or numeric statements.

For example, the following **insert** statement stores a new row into the SKI_AREAS table by using constant values for assignments:

```
exec sql
   insert into ski_areas (name, city, state, type)
      values ('Rte. 16', 'N. Conway', 'NH', 'N');
```

Using Host Variables

You can assign values to a new row by using host variables. To use host variables:

1. Assign values to the host variables.
2. Assign those variables to database columns.

For example, the following **insert** statement acquires all of its values through host variables:

```
char  skiname [26], skicity [26];
char  skistate [3];
char  skitype;

printf ("Enter name of ski area: ");
gets (skiname);
printf ("Enter municipality: ");
gets (skicity);
printf ("Enter state or commonwealth code: ");
gets (skistate);
printf ("Enter type: ");
getchar (skitype);
exec sql
  insert into ski_areas (name, city, state, type)
      values (:skiname, :skicity, :skistate, :skitype);
```

Using Values from a Subquery

You can assign values to a new row by using values retrieved from a subquery. To assign values:

1. Use a **select** statement in a subquery to choose the row you want.
2. Copy the values from the selected row into the row you are storing.

For example, suppose you want to store a row for a city that is very close to another city. You choose to duplicate all of the values except for the city's name. The following example finds a specific row in CITIES and stores a new CITIES row by using some of the values from the row identified in the subquery:

```
/* sql program */

exec sql
  include sqlca;

char villeancienne [26],
  villenouvelle [26];

main ()
{
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;

if ( (SQLCODE) && (SQLCODE != 100)) {
  printf ("Encountered SQLCODE %d\n", SQLCODE);
  printf ("  expanded error message -\n");
  gds_$print_status (gds_$status);
  exec sql
      rollback release;
  }
else
  exec sql
      commit release;
}
```

The assignments in the subquery can include arithmetic operations. For example, suppose you have an application that keeps track of people by using an employee number. When you hire a new employee, you take the highest current employee number and increase it by 1.

You can use a subquery to find the highest customer number. When you make the assignment, read that value and add 1.

The following assignment assigns an employee number and reads in values for LAST_NAME and FIRST_NAME from host variables:

```
exec sql insert into employees
   (employee_number, last_name, first_name)
   select (max employee_number + 1, :lastname, :firstname)
      from employees;
```

Storing Rows with Missing Column Values

There are three ways to assign a missing value to a column when you store a row:

- Assign the value **null**
- Ignore the column
- Use indicator parameters

The missing value options are discussed below.

Note

Not all database management systems support three-state logic (true, false, and missing). If you plan to migrate your SQL programs to systems other than InterBase and you use missing values, be sure the target system supports this type of logic.

Assigning a Null Value

You can set a column to missing by specifying **null** when you store the row. For example, the following statement stores a row into the CITIES table, assigns the values of host variables to some columns, and assigns a null value to other columns:

```
exec sql
  insert into cities
      (city, state, population, altitude,
      latitude_degrees, latitude_minutes, latitude_compass,
      longitude_degrees, longitude_minutes, longitude_compass)
  values (:newcity, :newstate, :newpopulation, :newaltitude,
      null, null, null, null, null, null);
```

The **null** assignment is the standard SQL approach to assigning missing values.

Ignoring the Column

When you store a row, you can assign a missing value to a column by not referencing that column (and thus not providing a value). In this case, InterBase sets a flag for that column indicating that its value is missing.

Consider the following **insert** statement:

```
exec sql
  insert into cities
      (city, state, population)
  values (:newcity, :newstate, :newpopulation);
```

This **insert** statement does not include assignments to the LATITUDE, LONGITUDE, POPULATION, or ALTITUDE columns, so InterBase sets the missing flag for each of them. (LATITUDE and LONGITUDE are computed fields, so there is no way to assign values directly to them.) Similarly, if you add a column to a table, InterBase sets the missing value flag for all existing rows until you provide a value.

Using Indicator Parameters

If you want to store values from existing data, you need to know if any of the values are missing. If you don't check for missing values when you store existing data, InterBase stores zeros for missing numeric data and spaces for missing character data. Therefore, the missing data is indistinguishable from "true" zeros and spaces.

You can use indicator parameters to check for missing values before you store them. If the value retrieved is:

- Missing, InterBase sets the indicator parameter to -1.
- Not missing, InterBase sets the indicator parameter to 0.

The program below shows how to store values for the CITY and POPULATION columns. It sets the indicator parameter to 0 when the POPULATION column contains a value. It sets the indicator parameter to -1 when the POPULATION column contains missing values:

```
exec sql begin declare section;
exec sql end declare section;
main ()
{
char      city [20], pop [20];
long      population;
short     pi;

/* here's an insert with value present */

population = 2300;
pi = 0;
exec sql
    insert into cities (city, state, population)
    values ("Hanson", "MA", :population :pi)
if (SQLCODE)
    gds_$print_status (gds_$status);

/* here's an insert with value missing */
pi = -1;
exec sql
```

```
    insert into cities (city, state, population)
    values ("Wenham", "MA", :population :pi)
if (SQLCODE)
    gds_$print_status (gds_$status);

exec sql commit release;
}
```

Storing Special Types of Columns

Because host languages do not directly support InterBase's date datatype, and SQL does not support the blob or array datatypes, you must do the following if you intend to store data into columns that have these datatypes:

- For dates use the following **gds** call interface routines:
  - — The **gds_$decode_date** routine, to convert the InterBase internal date format to the UNIX time structure.
  - — The **gds_$encode_date**, to convert the UNIX time structure to the internal InterBase date format.

  These routines are described in Chapter 9, *Using Date Fields*.
- For blobs, use GDML blob handling statements, GDML library routines, or the **gds** call interface routines. These statements and routines are described in Chapter 8, *Using Blob Fields*.
- For arrays, use the GDML **store** statement. This statement is described in Chapter 6, *Writing Data*.

For more information on mixing SQL and GDML, refer to Chapter 17, *Mixing SQL with Other Interfaces*.

Storing Data Through a View

You can store rows through a view if either of the following criteria are met:

- The view references only a single table.
- The appropriate triggers are defined in the metadata.

For example, suppose you have the following view definition:

```
create view city.pops
  (city, state, population)
  as select city, state, population
  from cities;
```

Because this view references only one table, CITIES, you can store a new CITIES row by storing data into this view:

```
exec sql
   insert into pop_cities (city, state, population)
   values('Montgomery','AL', '177857);
```

When you store rows through a view, InterBase stores missing values for unreferenced fields.

For information on defining triggers, refer to the chapter on preserving data integrity in the *Data Definition Guide*.

Modifying Data

You use the **set** substatement of the **update** statement to modify a row in a table. For example, the following statement changes the value of the TYPE column in a particular SKI_AREAS row:

```
exec sql
  update ski_areas
     set type = 'N'
     where name = 'Flints Corner';
```

The value assigned in this statement can be either a quoted string, a number, or a host variable.

Instructions for the following topics are presented below:

- Modifying multiple rows
- Modifying rows with missing column values
- Modifying special types of columns
- Modifying through a view

Modifying Multiple Rows

There are two ways to modify multiple rows:

- You can modify the rows in a single step, as a mass update operation. For example, the following program changes the population of all cities identified by the host variable STATECODE:

```
exec sql begin declare section;
exec sql end declare section;
main ()
{
char statecode [5];
int  multiplier;
char mult [8];

printf ("Enter state with population needing adjustment: ");
gets (statecode);
printf ("Percent change (e.g. 5 = 5%% increase, \
            -5 = 5%% decrease: ");
gets (mult);
multiplier = atoi (mult);
exec sql
    update cities
```

```
        set population  = population * (1 + :multiplier /
             100)
        where state = :statecode;

if (SQLCODE && (SQLCODE != 100))
    {
    printf ("Encountered SQLCODE %d\n", SQLCODE);
    printf ("    expanded error message -\n");
  gds_$print_status (gds_$status);
    exec sql
    rollback release;
    }
else
    exec sql commit release;
}
```

- You can use a cursor to select each row before modifying it. For example, the following update of the POPULATION column in the CITIES table takes place through a cursor:

```
exec sql begin declare section;
exec sql end declare section;
main ()
{
char statecode [5], state[5];
char city [26];
int  multiplier, pop;
char mult [10];

printf ("Enter state with population needing adjustment: ");
gets (statecode);

exec sql
    declare pop_mod cursor for
   select city, state, population from cities
            where state = :statecode
               and population is not null
            for update of population;
exec sql
    open pop_mod;
exec sql
    fetch pop_mod into :city, :state, :pop;

while (!SQLCODE)
```

```
        {
        printf ("City is %s, %s population %d\n", city, state,
                 pop);

        printf ("Percent change (e.g. 5 = 5%% increase,\
                 -5 = 5%% decrease: ");
       gets (mult);
        pop *=  (atoi (mult) + 100);
        pop /= 100;
        printf ("New population will be %d\n", pop);
        exec sql
            update cities
                set population  = :pop
                where current of pop_mod;
        exec sql
            fetch pop_mod into :city, :state, :pop;
        }

    if (SQLCODE != 100)
        {
        printf ("Encountered SQLCODE %d\n", SQLCODE);
        printf ("    expanded error message - \n");
        gds_$print_status (gds_$status);
        exec sql
        rollback release;
        }
    else
        exec sql commit release;
    }
```

In general, the cursor approach is more flexible than the one involving mass updates. There is no way to display the pre- or post-modification value of a column in the mass update approach, because the row selection is performed in the **update** statement itself. Nor can you prompt the user for a specific value for each row.

Modifying Rows with Missing Column Values

You set a column to missing by specifying **null** when you modify a row. For example, the following statement changes the value of the SEATING column in a particular BASEBALL_TEAMS table to null values:

```
/* sql program */

exec sql
```

```
    include sqlca;

char teamname [26];

main()
{

printf ("Enter team name: ");
gets (teamname);

exec sql update baseball_teams
   set seating = null
   where team_name = :teamname;

if ( (SQLCODE) && (SQLCODE != 100)) {
   printf ("Encountered SQLCODE %d\n", SQLCODE);
   printf ("  expanded error message -\n");
   gds_$print_status (gds_$status);
   exec sql
       rollback release;
   }
else
   exec sql
       commit release;
}
```

Modifying Special Types of Columns

Because host languages do not directly support InterBase's date datatype, and SQL does not support the blob or array datatypes, you must do the following if you intend to modify columns that have these datatypes:

- For dates, use the following **gds** call interface routines:
  - — The **gds_$decode_date** routine, to convert the InterBase internal date format to the UNIX time structure.
  - — The **gds_$encode_date** routine, to convert the UNIX time structure to the internal InterBase date format.

  These routines are described in Chapter 9, *Using Date Fields*.
- For blobs, use GDML blob handling statements, GDML library routines, or the **gds** call interface routines. These statements and routines are described in Chapter 8, *Using Blob Fields*.

- For arrays, use the GDML **modify** statement within a GDML **for** loop. The **modify** statement is described in Chapter 6, *Writing Data with GDML*. The **for** loop is described in Chapter 5, *Retrieving Data with GDML*.

For more information on mixing SQL and GDML, refer to Chapter 17, *Mixing SQL with Other Interfaces*.

Modifying Through a View

You can modify rows through a view if either of the following criteria are met:

- The view references only a single table.
- The appropriate triggers are defined in the metadata.

For example, suppose you have the following view definition:

```
create view city_pops
   (city, state, population)
   as select city, state, population
   from cities;
```

Because this view references only one table, CITIES, you can modify a CITIES row by modifying data in this view:

```
exec sql
   update pop_cities
   set population = '183453'
      where city = 'Montgomery';
```

For information on defining triggers, refer to the chapter on preserving data integrity in the *Data Definition Guide*.

Deleting Data

You use the **delete** statement to delete a row from a table. For example, the following statement deletes a particular SKI_AREAS row:

```
exec sql
  delete ski_areas
      where name = 'Flints Corner';
```

You can use this statement to delete rows that contain any type of columns, including blob, date, and array columns.

Instructions for deleting multiple rows and deleting through a view are presented below.

Deleting Multiple Rows

There are two ways to delete multiple rows:

- You can delete the rows in a single step, as a mass delete operation. For example, the following program erases all qualifying rows from the CITIES table:

```
/* sql program */

exec sql
   include sqlca;

char  statecode [5];

main()
{
printf ("Enter code of state to raze: ");
gets (statecode);
exec sql
   delete from cities
   where state = :statecode;

if ( (SQLCODE != 0) && (SQLCODE != 100)) {
   printf ("Encountered SQLCODE %d\n", SQLCODE);
   printf ("  expanded error message -\n");
   gds_$print_status (gds_$status);
   exec sql
      rollback release;
   }
else
```

```
   exec sql
      commit release;
}
```

- You can use a cursor to select each row before deleting it. For example, the following program erases all qualifying rows from the CITIES table by using a cursor:

```
/* sql program */

exec sql
   include sqlca;

char  statecode [5], st [5];
char  cityname [26];
int   pop;
char  option [4];

main()
{

printf ("Enter code of state to raze: ");
gets (statecode);

exec sql
   declare raze_cities cursor for
      select city, state, population
      from cities where state = :statecode;
exec sql
   open raze_cities;
exec sql
   fetch raze_cities into :cityname, :st, :pop;

while (!SQLCODE) {
   printf ("Eliminate %s, %s: %d? ", cityname, st, pop);
   gets (option);
   if ((option[0] == "Y") || (option[0] == "y")) {
      exec sql
         delete from cities
         where current of raze_cities;
      }
   exec sql
      fetch raze_cities into :cityname, :st, :pop;
   }
```

```
if ( (SQLCODE) && (SQLCODE != 100)) {
   printf ("Encountered SQLCODE %d\n", SQLCODE);
   printf ("  expanded error message -\n");
   gds_$print_status (gds_$status);
   exec sql
      rollback release;
   }
else
   exec sql
      commit release;
}
```

Deleting Through a View

You can delete rows through a view if either of the following criteria are met:

- The view references only a single table.
- The appropriate triggers are defined in the metadata.

For example, suppose you have the following view definition:

```
create view pop_cities
   (city, state, population)
   as select city, state, population
   from cities;
```

Because this view references only one table, CITIES, you can delete a CITIES row by deleting a row in this view:

```
exec sql
   delete from pop_cities
      where city = 'Montgomery';
```

For information on defining triggers, refer to the chapter on preserving data integrity in the *Data Definition Guide.*

For More Information

For more information on storing and modifying blob, date, and array data, refer to Part II, *Programming with GDML*.

For more information on using embedded SQL, refer to:

- Chapter 13, *Getting Started with SQL*, for introductory information on using embedded SQL.
- Chapter 14, *Retrieving Data with SQL*, for information on retrieving data with embedded SQL.
- Chapter 16, *Defining Metadata with SQL*, for information on defining metadata with embedded SQL.
- Chapter 17, *Mixing SQL with Other Interfaces*, for information on mixing SQL and GDML.
- Chapter 18, *Preprocessing Your Program*, for information on using **gpre** to preprocess your SQL program.
- The chapter on SQL statements in the *Programmer's Reference*, for information on specific SQL commands and statements.

Chapter 16
Defining Metadata with SQL

This chapter discusses how to define, modify, and delete metadata by using embedded SQL.

Overview

You can use the following SQL commands to define, modify, and delete metadata:

- **create database**, which allows you to create databases
- **create table**, **alter table**, and **drop table,** which allow you to create, modify, and delete tables
- **create index** and **drop index**, which allow you to create and delete indexes
- **create view** and **drop view**, which allow you to create and delete views
- **grant** and **revoke**, which allow you to give access to and remove access from tables

The following considerations apply to defining metadata with SQL:

- You can't define global columns with SQL. You must always define columns locally within tables.
- You can't use SQL to define the following column attributes:
  - Query headers, query names, and edit strings
  - Comments for each column
  - Column validation (except **not null**)
  - Computed columns
  - Columns with blob, date, and array datatypes
  - Alternate missing values

  You *can* define these attributes with **gdef**. For information, refer to the chapter on defining fields in the *Data Definition Guide*.

Unsupported Data Definition Statements

Some metadata operations associated with other implementations of SQL are not supported, because they are not part of the ANSI SQL standard:.

- **Gpre** returns a warning if your program includes any of the following statements:
  - — Statements that deal with database **storage groups**. **Gdef** provides options on its **define database** and **modify database** statements to specify or change database storage characteristics.
  - — Statements that deal with **tablespaces. Gdef** provides options on its **define database** and **modify database** statements to specify or change database storage characteristics.
  - — The **alter index** command. **Gdef** provides the **modify index** statement to change the active status or uniqueness of an index.

  For more information on **gdef** statements, refer to the *Data Definition Guide*.
- **Gpre** returns an error if your program includes any of the following commands:
  - — comment on
  - — create synonym
  - — drop synonym
  - — lock table

Defining and Deleting a Database

You define a database by using the **create database** command, followed by the definition of one or more tables, views, and indexes. At the very least, a database definition consists of one statement that creates the database and one that defines a table.

A sample database definition containing two tables is shown below:

```
exec sql
  create database "my_very_own.gdb";

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),
      composer_first_name varchar(10),
      type char(2));

exec sql
  create table performers (
      last_name varchar(20),
      first_name varchar(10),
      nationality varchar(10));
```

Defining and Deleting Tables and Columns

You define tables and columns by using the **create table** command. For example, the following statement defines a table called SKI_AREAS:

```
exec sql
  create table ski_areas (
      name varchar(20) not null,
      type char(1),
      city varchar(25),
      state varchar(4));
```

The following considerations apply to defining tables and columns:

- Tables created with the **create table** command are automatically secured against unauthorized access. You can grant access to a table you created by using the SQL **grant** and **revoke** commands. You cannot use security classes to secure these tables.
- Columns defined through the **create table** command are local columns. To define global columns and include them in a table definition, you must use either **gdef**, or the GDML version of **qli.**

You can modify a table by adding and dropping columns. You do this by using the **alter table** command. For example, the following command changes the datatype and length of the STATE column in the SKI_AREAS table:

```
exec sql
  alter table ski_areas
      drop state varchar(4),
      add state char(2);
```

To delete a table, use the **drop table** command. For example, the following command deletes the SKI_AREAS table:

```
exec sql
  drop table ski_areas;
```

Defining and Deleting Indexes

An *index* is an internal structure that optimizes row retrieval. You define an index by using the **create index** command. For example, the following command defines a single-segment index for the STATE column in the STATES table:

```
exec sql
  create index states_idx1 on states (state);
```

This command defines a multi-segment index (that is, an index with multiple columns) for the STATE and RIVER columns in the RIVER_STATES table:

```
exec sql
  create index rivstate_idx1 on river_states (state, river);
```

You can use indexes to eliminate duplicates and make a descending sort more efficient:

- If you want to eliminate duplicates, include the **unique** option when you create the index. This causes InterBase to disallow users from storing duplicate values in the indexed column. For example:

```
exec sql
   create unique index states_idx1 on states
      (state);

exec sql
   create unique index states_idx2 on states
      (city, state);
```

- By default, SQL stores an index in ascending order. If you want to make a descending sort on a column or group of columns more efficient, include the **descending** option when you create the index. For example:

```
exec sql
   create unique desc index states_idx3 on states
      (state);
```

  This directs InterBase to build the index in descending order. To retrieve the indexed data in descending order, you must use the **order by** clause in the **select** statement.

To delete an index, use the **drop index** command:

```
exec sql
  drop index states_idx3;
```

Defining and Deleting Views

A *view* is a virtual table that comprises a vertical or horizontal subset of one or more tables.

You define a view by using the **create view** command. Within this command, you need to specify:

- A view name
- One or more column names
- A **select** statement that names the columns and rows you want to include in the view

For example, the following view definition contains several columns from all rows in the CITIES table:

```
exec sql
  create view map_cities as
      select city, state, latitude, longitude from cities;
```

This view definition accesses information about cities that are located above 40 degrees latitude:

```
exec sql
  create view ice_belt as
      select city, state, population, altitude, latitude,
          longitude from cities where latitude_degrees >= 40;
```

This view definition joins the CITIES table to the STATES table:

```
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;
```

When you define a view using embedded SQL, InterBase doesn't save the source code for the view. It saves the compiled version only. This has the following implications:

- You can use the view through **qli**. However, if you issue a **show** statement for the view, **qli** can't display the view source.
- If you do an extract through **gdef**, you get a message telling you that the view must be re-created.

To delete a view, use the **drop view** statement:

```
exec sql
  drop view capital_cities;
```

Controlling Access to Tables

You can use embedded SQL to control access to tables. You do this by using the **grant** and **revoke** security commands.

In accordance with the SQL security model, a table created with the **create table** command is automatically secured from unauthorized access. Only the table owner (that is, the creator of the table) can initially grant privileges using **grant** commands. Additionally, only the owner of a table can modify the table definition.

Granting Privileges

To grant a user access to a table, you use a **grant** command that specifies the privilege, the table name, and the user name or names. For example, the following command grants read-only privileges for the states table to two users:

```
exec sql
   grant select on states to juliec, dana;
```

The privileges you can grant are listed in Table 16-1.

Table 16-1. Grant Privileges

Privilege	Access provided
All	Read, write, update, and delete data.
Select	Read data.
Delete	Delete data.
Insert	Write values to the database.
Update	Modify existing data.

To grant a privilege to all users of the database, you can specify **public** rather than list all user names:

```
exec sql
  grant all on tourism to public;
```

You can assign more than one privilege in a command. For example, to provide read and write privileges to all users, type:

```
exec sql
  grant select, insert on tourism to public;
```

Finally, you can use a **grant** command to assign grant authority to users. For example, you may want all supervisor-level users to grant access privileges at their discretion to their department members. Assigning a **with grant option** privilege to a user enables the user to grant privileges.

The following example illustrates the **with grant option** privilege:

```
exec sql
  grant select on states to charles with grant option;
```

Charles is now authorized to grant select privileges to other users. A grantor can only grant the privileges for which he or she is authorized. Thus, Charles can grant only select privileges to other users.

Revoking Privileges

You can remove privileges from a user by using the **revoke** command. For example, to revoke all user access to the states table, type:

```
exec sql
  revoke all on states from public;
```

Revoking privileges can have a cascading effect. One revoke action can cause other revoke actions. For example, suppose that Charles, having been assigned granting privileges, grants select authority to Diane. If you revoke Charles' privileges, Diane's privileges are also revoked:

```
exec sql
  revoke select on states from charles;
```

If Diane has also been granted select privileges from another source, she retains that privilege when Charles loses his. For example, suppose Charles, when he had the authority to do so, granted Diane select and insert privileges for the states table:

```
exec sql
  grant select,insert on states to diane;
```

Then, Elizabeth grants Diane select and update privileges:

```
exec sql
  grant select,update on states to diane;
```

Diane now possesses the authority to read, write, and update data. Now, however, Charles changes jobs and his privileges are revoked:

```
exec sql
  revoke all on states from charles;
```

Charles' authority to grant privileges is also revoked. Thus, the select and insert privileges should be taken from Diane. However, since she is authorized to read and update data through Elizabeth, Diane only loses her insert privilege.

For more information on the syntax for the **grant** and **revoke** commands, refer to the chapter on SQL statements in the *Programmer's Reference*.

Securing a Database

InterBase provides another security model that enables you to secure databases, tables, views, and columns. This security model, which uses security classes to control user access to database structures, is described in detail in the chapter on securing data in the *Data Definition Guide*.

The following considerations apply to mixing the **grant/revoke** security model with the security class model:

- Once you create a table using the **create table** command, you can't use security classes for that table. You can, however, use them for securing columns within that table.
- If you create a table through **gdef** by using the **define relation** statement, the table is not automatically secured from unauthorized access. You can secure the table by using either of InterBase's security mechanisms:
  - — You can assign a security class to the table. Once you assign a security class to a table, you can't use an SQL **grant** command to give access to the table.
  - — You can use the SQL **grant** command to grant specific access to the table. Once you use this statement, the table is automatically secured from unauthorized access, and you must explicitly grant access to it.

Since the SQL **grant/revoke** security model only applies to tables and views, we suggest that you follow these steps to secure your database:

1. Secure the database itself using an InterBase security class.
2. Secure the RDB$SECURITY_CLASSES system table, which controls the security-class relation. This prevents unauthorized users from subverting SQL security.
3. Use either security classes *or* **grant** commands to limit access to tables and views. *Mixing the two models of security can lead to unpredictable results.*

Sample Database Definition

The following database is a subset of the definition of the atlas database listed in Appendix A, *Sample Database Definition*. SQL does not support all of the data definition capabilities of **gdef**, including computed columns and blob columns.

```
exec sql
  create database "atlas.gdb";

exec sql
  create table CITIES (
      CITY varchar(25),
      STATE char(2),
      POPULATION integer,
      ALTITUDE integer,
      LATITUDE_DEGREES varchar(3),
      LATITUDE_MINUTES char(2),
      LATITUDE_COMPASS char(1),
      LONGITUDE_DEGREES varchar(3),
      LONGITUDE_MINUTES char(2),
      LONGITUDE_COMPASS char(1));

exec sql
  create table POPULATIONS (
      STATE char(2),
      CENSUS_1950 integer,
      CENSUS_1960 integer,
      CENSUS_1970 integer,
      CENSUS_1980 integer);

exec sql
  create table PROVINCES (
      PROVINCE varchar(4),
      PROVINCE_NAME char(3),
      AREA integer,
      CAPITAL varchar(4));

exec sql
  create table RIVERS (
      RIVER varchar(20),
      LENGTH integer,
      HEADWATER_STATE char(2));

exec sql
  create table RIVER_STATES (
```

```
        STATE char(2),
        RIVER varchar(20));

exec sql
  create table SKI_AREAS (
        NAME varchar(20),
        TYPE char(1),
        CITY varch(25),
        STATE char(2));

exec sql
  create table STATES (
        STATE char(2),
        STATE_NAME varchar(25),
        AREA integer,
        STATEHOOD date,
        CAPITAL varchar(25));

exec sql
  create table TOURISM (
        STATE char(2),
        ZIP char(9),
        CITY varchar(25));

exec sql
  create view ski_cities as
        select ski.ski.name, ski.ski.city, s.state_name
            from ski_areas ski, states s
            where ski.state = s.state;

exec sql
  create view smaller_cities as
        select city, state, population
        where population < 50000;

exec sql
  create unique index cities_1 on cities
        (city, state);

exec sql
  create unique index state_1 on states
        (state);
```

For More Information

For a discussion of relational data definition, database design, and security classes, refer to the *Data Definition Guide*.

For more information on programming with embedded SQL, refer to:

- Chapter 14, *Retrieving Data with SQL*, for information on retrieving data with embedded SQL.
- Chapter 15, *Writing Data with SQL*, for information on storing, modifying, and deleting data with embedded SQL.
- Chapter 17, *Mixing SQL with Other Interfaces*, for information on mixing SQL with GDML and OSRI calls.
- Chapter 18, *Preprocessing Your Program*, for information on using **gpre** to preprocess your SQL program.
- The chapter on SQL statements in the *Programmer's Reference*, for information on specific SQL statements and commands.
- The appendix on error handling in the *Programmer's Reference*, for a list of InterBase error messages.

Chapter 17
Mixing SQL With Other Interfaces

This chapter describes how to mix SQL with GDML statements and **gds** calls.

Overview

InterBase offers database capabilities that do not exist in SQL. You can take advantage of these capabilities by using GDML statements and **gds** calls in some of your SQL programs.

By using these statements and calls in your SQL program, you gain:

- The ability to check for specific InterBase errors
- The ability to access multiple databases
- Greater control over transactions
- Access to blob and date fields

Using GDML statements to check for specific InterBase errors is discussed in Chapter 13, *Getting Started*. Using GDML statements and **gds** calls to access multiple data-

bases, gain greater control over transactions, and access blob and date fields is discussed in the remainder of this chapter.

Note

If you mix SQL and GDML statements within a program and you have a GDML **database** statement, it must occur before the first **exec sql** statement. If you have the **exec sql** statement first, **gpre** will generate code that fails at runtime.

Accessing Multiple Databases

You can also use GDML statements to access multiple local and remote databases in a single transaction. This capability enables you to distribute the workload and storage of your databases without losing access to data.

There are two ways to access multiple databases in an SQL program:

- You can ready multiple databases serially by using GDML **ready** commands. You must ready each database by using the SQL default database handle, **gds_$database**.

  Gpre automatically generates the SQL default database handle when you preprocess your program. **Gpre** won't generate this handle if your program uses GDML **database** declarations to specify other database handles.

- You can declare multiple databases by using GDML **database** declarations. Then, you can access each database by using the appropriate database handle as an SQL qualifier.

  For example, suppose you declare one database as *db1*. You can access the STATES table from this particular database by specifying DB1.STATES as the target table.

Note

> If you use GDML database declarations to declare multiple databases, don't use the **database** option when you preprocess your program. You should also not use this option when pre-processing a single database statement.

The SQL program below accesses multiple databases serially by using the **ready** command to change the file associated with the default database handle. You must use an SQL **commit release** command to detach one database before opening the next:

```
exec sql begin declare section;
exec sql end declare section;
main ()

{
long  state_count;

state_count = 0;

ready "atlas.gdb" as gds_$database;
exec sql select count (*) into :state_count from states;
printf ("%d states in database 1\n", state_count);
exec sql commit release;
```

```
ready "coastal_guide.gdb" as gds_$database;
exec sql select count (*) into :state_count from states;
printf ("%d states in database 2\n", state_count);
exec sql commit release;

ready "north_coast_guide.gdb" as gds_$database;
exec sql select count (*) into :state_count from states;
printf ("%d states in database 3\n", state_count);
exec sql commit release;
}
```

The SQL program below accesses multiple databases by declaring them with GDML **database** declarations and using their database handles as table qualifiers:

```
database db1 = filename "atlas.gdb";
database db2 = filename "coastal_guide.gdb";
database db3 = filename "north_coast_guide.gdb";

main ()
{
long  state_count;
state_count = 0;

exec sql select count (*) into :state_count from db1.states;

printf ("%d states in database 1\n", state_count);

exec sql select count (*) into :state_count from db2.states;

printf ("%d states in database 2\n", state_count);

exec sql select count (*) into :state_count from db3.states;

printf ("%d states in database 3\n", state_count);

exec sql commit release;

}
```

Controlling Transactions

GDML gives you more control over your transaction environment than SQL. By using GDML, you can explicitly start and end a transaction.

You use the GDML **start_transaction** command to specify when a transaction begins. This capability does the following:

- Gives you more control over a transaction's scope of operations
- Lets you use all available GDML transaction options, including the **reserving** clause
- Lets you coordinate transaction activity with non-database activity

If you don't use the **start_transaction** command, InterBase starts a transaction for you automatically when it encounters the first SQL statement or command in the program, unless that statement is a **commit** command. InterBase ends the transaction when you **commit** or **commit release** your work. The next SQL statement other than **commit** starts a new transaction.

The SQL program below uses a GDML **start_transaction** command to start a transaction explicitly and override the default transaction options:

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

main ()
{

start_transaction READ_WRITE CONSISTENCY RESERVING
    CITIES FOR PROTECTED WRITE;

exec sql
    update cities
      set population = 100
      where population is null;

exec sql
    commit release;

}
```

For more information on GMDL transaction options, refer to Chapter 2, *InterBase Transaction Management*.

Accessing Blob and Date Fields

You can use GDML statements and **gds** calls to access columns defined with the blob (basic large object), and date datatypes. You can access date fields from SQL directly, but converting the data from its numeric format to a readable representation requires aditional processing. Blob fields are entirely outside the scope of SQL.

Accessing Blobs

A *blob* is a field that looks like a stream or sequential file, but behaves like a column in a table. Blobs can hold text, graphics, images, digitized voice, arrays, or any other large unstructured data. Blobs operate under transaction and concurrency control.

You can use the following blob library routines to retrieve and store blob information:

- **blob_$load** (**BLOB_load** in C)
- **blob_$dump** (**BLOB_dump** in C)
- **blob_$display** (**BLOB_display** in C)
- **blob_$edit** (**BLOB_edit** in C)

You can also use the following **gds** statements to retrieve and store blob information:

- gds_$create_blob
- gds_$open_blob
- gds_$get_segment
- gds_$put_segment
- gds_$close_blob
- gds_$cancel_blob

To use blob fields from SQL programs, you must store or retrieve the blob identifier with the record. Both the lower-level **gds** calls and the higher-level BLOB calls give you access to the data in the blob.

For example, to load a blob from a file:

1. Declare a variable of type **gds_$quad** (**GDS_$QUAD** in C) to hold the blob identifier created by the load routine.
2. Start a transaction before you call **blob_$load**. You can do this in one of two ways:
   — Execute an SQL statement other than **commit** before you call **blob_$load**.
   — Use the GDML **start_transaction** command.
3. Pass the variable declared in Step 1 plus the file name to the **blob_$load** call.

4. When the call completes, assign the value of the variable to the database field using standard SQL syntax for insertions or updates, as appropriate.

You can use any of the blob calls as documented, replacing the blob field reference with the name of your variable.

The SQL program below shows how to load blob data by using the **blob_$load** call:

```
exec sql begin declare section;
exec sql end declare section;
main ()
{
GDS_$QUAD  blob_id;
char  filename [30];
int   c;

exec sql
    select count (*) from sources into :c;

printf ("Please enter the name of a source file to store: ");
gets (filename);

BLOB_load (&blob_id, gds_$database, gds_$trans, filename);
exec sql
    insert into sources (filename, source) values (:filename,
  :blob_id);

if (SQLCODE && (SQLCODE != 100))
    {
    printf ("Encountered SQLCODE %d\n", SQLCODE);
    printf ("    expanded error message -\n");
    gds_$print_status (gds_$status);
    exec sql
    rollback release;
    }
else
    exec sql commit release;
}
```

In this program:

- **GDS_$QUAD** is a macro that defines a 64-bit quantity, which represents the blob identifier. This identifier points to the blob data. **GDS_$QUAD** is defined in a header file that's automatically included by **gpre**.
- **gds_$database** is the default database handle for SQL programs.

- **gds_$trans** is the default transaction handle. This is the required transaction handle for SQL statements.
- **gds_$status** is the default status vector.

On systems that don't support the dollar sign in identifiers, the dollar sign is replaced by two underscores (_ _). For example, **gds_$quad** becomes **gds_ _quad** on these systems.

The SQL program below shows how to retrieve and print the contents of the blob field GUIDEBOOK:

```
/*
 * This short program selects a single record
 * and prints the contents of a blob field in
 * the record.
 */
exec sql begin declare section;
exec sql end declare section;

main ()
{
GDS_$QUAD         blob_id;
long              blob_handle;
char              segment [60], string [61];
unsigned short    length;

 exec sql select t.guidebook into :blob_id from tourism t with
   t.state = "ME";
if (SQLCODE)
    {
    gds_$print_status (gds_$status);
    exit (1);
    }

/* the blob handle must be initialized to zero before the blob
open call */

blob_handle = 0;

gds_$open_blob (*gds_$null, GDS_REF(gds_$database),
   GDS_REF(gds_$trans), GDS_REF(blob_handle), GDS_REF(blob_id);
for (;;)
    {
    gds_$status[1] = gds_$get_segment (*gds_$null,
```

```
        GDS_REF(blob_handle), GDS_REF(length), sizeof (segment),
            segment);
    if (gds_$status[1] && (gds_$status[1] != gds_$segment))
                break;
    strncpy (string, segment, length);
    string[length] = 0;
    printf ("%s\n", string);
    }
if (gds_$status[1] != gds_$segstr_eof)
    gds_$print_status (gds_$status);

exec sql
    commit release;
}
```

In this program:

- **GDS_$QUAD** is a macro that defines a 64-bit quantity, which represents the blob identifier. This identifier points to the blob data. **GDS_$QUAD** is defined in a header file that's automatically included by **gpre**.
- **gds_$null** is the Apollo version of the null status vector. This causes InterBase to print the status of a call and to terminate if the call fails. On other systems, supplying a zero in this position has the same result.
- **gds_$database** is the default database handle for SQL programs.
- **gds_$trans** is the default transaction handle. This is the required transaction handle for SQL statements.
- **gds_$status** is the default status vector.
- **gds_$segment** is a status code that indicates that an incomplete segment was returned to avoid overflowing the buffer provided.
- **gds_$segstr_eof** is the normal end-of-file for a blob string loop.

The SQL program below shows how to use the C-style blob calls from SQL:

```
#include <stdio.h>
exec sql begin declare section;
exec sql end declare section;

main ()
{
BSTREAM     *g_in;
GDS_$QUAD   g_id;
int         c;

exec sql declare c cursor for
```

```
    select guidebook from tourism;
exec sql open c;

exec sql fetch c into :g_id;

while (!SQLCODE)
    {
    g_in = Bopen (&g_id, gds_$database, gds_$trans, "r");
    while ((c = getb (g_in)) != EOF)
   putchar (c);
    BLOB_close (g_in);
    exec sql fetch c into :g_id;
    }
if (SQLCODE != 100)
    gds_$print_status (gds_$status);
exec sql
    commit release;

}
```

For more information on using GDML blob routines and **gds** statements to access and write to a blob, refer to Chapter 8, *Using Blob Fields*.

Accessing Dates

You can retrieve and store date information in an SQL program as follows:

- You can retrieve date information by using the **gds_$decode_date** routine. This routine converts the InterBase internal date format to the UNIX time structure.
- You can store date information by using the **gds_$encode_date** routine. This routine converts the UNIX time structure to the InterBase internal date format.

The SQL program below uses the **gds_decode_date** routine to access statehood information:

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

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

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

main ()
{
GDS_$QUAD     date;
char          state[26];
struct tm     time;
short         year;

exec sql declare c cursor for
  select s.state_name, s.statehood from states s
      where s.statehood is not null
      order by s.statehood;


exec sql open c;

if (!SQLCODE)
    exec sql fetch c into :state, :date;

while (!SQLCODE)
    {
    gds_$decode_date (&date, &time);
    year = 1900 + time.tm_year;
    printf ("%s entered the union on %s %d, %4d\n",
      state, months[time.tm_mon],
      time.tm_mday, year);
    exec sql fetch c into :state, :date;
    }
if (SQLCODE != 100)
    gds_$print_status (gds_$status);
exec sql
    commit_release;
}
```

For more information on using the **gds_$decode_date** routine and the **gds_$encode_date** routine to access and write to a date field, refer to Chapter 10, *Using Date Fields*.

For More Information

For more information on:

- The GDML **declare** and **ready**commands, refer to the chapter on GDML statements in the *Programmer's Reference.*
- InterBase transaction handling, refer to Chapter 2, *InterBase Transaction Management.*
- Blob library routines and **gds** blob calls, refer to Chapter 8, *Using Blob Fields.*
- **Gds** date routines, refer to Chapter 10, *Using Date Fields.*

Part IV
Preparing Your Program

Chapter 18
Preprocessing Your Program

This chapter describes how to preprocess your program by using **gpre**.

Overview

After you code your GDML, SQL, or DSQL program, you must use **gpre** to preprocess the program. **Gpre** translates GDML, SQL, and DSQL statements and commands into statements the host language compiler accepts. **Gpre** does this by generating InterBase library function calls.

Gpre also translates GDML, SQL, and DSQL database variables into variables the host language compiler accepts. **Gpre** then declares these variables in host language format.

Using Gpre

The syntax for **gpre** follows:

Operating System	Command
Apollo AEGIS	**gpre** [*-language*] [*-options*] *infile* [*outfile*]
UNIX	**gpre** [*-language*] [*-options*] *infile* [*outfile*]
VMS	**gpre** [*/language*] [*/options*] *infile* [*outfile*]

The following operating-specific considerations apply to this syntax:

- Under Apollo AEGIS, you can specify the **language** and **option** syntax options either before you specify the input and output files or after you specify these files. The syntax options must include at least a hyphen and a unique character.
- Under UNIX, you can also specify the **language** and **option** syntax options either before you specify the input and output files or after you specify these files. The syntax options must include at least a hyphen and a unique character.
- Under VMS, you can use either a slash or a hyphen to prefix a syntax option. You can specify the language and option syntax options either before you specify the input and output files or after you specify these files. The syntax option must include at least a slash (or hyphen) and a unique character.

  Also, you must have a space between the end of a filename and the beginning slash or hyphen of a switch. For example, this command fails:

  ```
  gpre census_report/e
  ```

  But this command works:

  ```
  gpre census_report /e
  ```

Detailed information on specifying a language and specifying other preprocessing options is presented below.

Specifying a Language

There are two ways to specify the language of your source program. You can:

- Use the language option.
- Name an input file that contains a language-specific file extension.

If you don't specify a language option or name an input file that contains a file extension, **gpre** searches for the file as described later in this chapter.

Using the Language Option

When you use **gpre** to preprocess a program, you can optionally use the **language** option to specify the language of the source program. Table 18-1 lists the available language options:

Table 18-1. Gpre Language Options

Language	Option
Ada (Alsys)	**al[sys]**
Ada (VERDIX, VMS, Telesoft)	**a[da]**
BASIC	**b[asic]**
C	**c**
COBOL	**co[bol]**
FORTRAN	**f[ortran]**
Pascal	**pa[scal]**
PL/I	**pl[i]**

For example, to preprocess a VERDIX Ada program called *census_report* on a UNIX system, type:

```
gpre -ada census_report
```

To preprocess a C program with the same name, type:

```
gpre -c census_report
```

Using a File Extension

You can also indicate the language of the source file by using an appropriate file extension when you name the source file. Table 18-2 lists the file extensions you can use to indicate the source file language. It also shows the output defaults that **gpre** produces for the file.

Table 18-2. File Extensions

Language	Input File Extension	Output Default
Ada (VERDIX)	ea	a
Ada (Alsys, VMS, Telesoft)	eada	ada
BASIC	ebas	bas
C	e	c
COBOL	ecob	cob
FORTRAN (VMS)	efor	for
FORTRAN (Apollo)	eftn	ftn
Pascal	epas	pas
PL/I	epli	pli

For example, to preprocess a VERDIX Ada program called *census_report*, type:

```
gpre census_report.ea
```

This generates an output file called *census_report.a*

When you use a file extension, you can still optionally use a language option as well:

```
gpre -ada census_report.ea
```

What Gpre Looks For

Because both the language switch and the file extension are optional, **gpre** can encounter three different situations:

- A language option with no file extension
- A file extension with no language option
- Neither a language option, nor a file extension

Each of these options is discussed below.

Specifying a Language Option with No File Extension

If you specify a language option, but name a file with no extension, **gpre** does the following:

1. Looks for the named file without an extension. If **gpre** finds the file, it processes it and generates an output file with the appropriate default extension.

   If **gpre** doesn't find the file, it looks for the named file with the extension that corresponds to the requested language. If it finds such a file, it generates an output file with the appropriate extension.

2. If **gpre** can't find either the named file or the named file with the appropriate extension, it returns the following error:

   ```
   gpre: can't open filename or filename.extension
   ```

   Filename is the file you specified in the **gpre** command. *Extension* is the language-specific file extension for the specified program.

For example, suppose you have the following command:

```
gpre -c census_report
```

Gpre:

1. Looks for a file called *census_report* without an extension. If it finds the file, it processes it and generates an output file called *census_report.c*.
2. If **gpre** doesn't find this file, it looks for a file called *census_report.e*. If **gpre** finds this file, it processes the file and generates an output file called *census_report.c*.
3. If **gpre** can't find a file named *census_report* with an appropriate extension, it returns this error:

   ```
   gpre: can't open census_report or census_report.e
   ```

Specifying a File Extension with No Language Option

If you specify a file extension, but do not specify a language option, **gpre** looks for the specified file and assumes the language is based on the extension.

For example, suppose you have the following command:

```
gpre census_report.e
```

Gpre looks for a file called *census_report.e*. If **gpre** finds this file, it processes it as a C program and generates an output file called *census_report.c*.

If **gpre** doesn't find this file, it returns the following error:

```
gpre: can't open census_report.e
```

Not Specifying a Language Option or a File Extension

If you don't specify either a language option or a file extension, **gpre** looks for a file in the following order:

1. *filename*.e (C)
2. *filename*.epas (Pascal)
3. *filename*.efor (VAX FORTRAN)
4. *filename*.eftn (APOLLO FORTRAN)
5. *filename*.ecob (COBOL)
6. *filename*.ebas (BASIC)
7. *filename*.epli (PL/I)
8. *filename*.ea (VERDIX Ada)
9. *filename*.eada (Alsys,VMS, and Telesoft Ada)

If **gpre** finds such a file, it generates an output file with the appropriate extension. If **gpre** doesn't find the file, it returns the folowing error:

```
gpre: can't find filename with any known extension.
   Giving up.
```

Filename is the file name you specified in the **gpre** command.

Specifying Preprocessing Options

When you use **gpre**, you can specify one or more preprocessing options. There are two types of preprocessing options:

- Generic options, which apply to any host language program.
- Specific options, which apply to specific situations. These include options for programs written in C and programs written in COBOL.

Specifying Generic Options

You can specify the following options for any host language program:

- **d**[**atabase**] *infile* declares a database for programs. This option takes the filename of the database to access as its argument.

  You must use this option if your program contains SQL statements only. You should not use this option if your program includes a GDML **database** declaration.

- **x** gives the database handle identified with the **database** option an external declaration. This option directs the program to pick up a global declaration from another linked module.
- **m**[**anual**] suppresses the automatic generation of GDML **ready** and **start_transaction** commands. This reduces code size and enables you to control the transactions that your program uses.

Note

Do not use the manual option for SQL programs unless you have included GDML transaction-handling statements in your program.

- **o**[**utput**] directs **gpre**'s output to standard out, rather than to a file.
- **r**[**aw**] prints BLR as raw numbers, rather than as their mnemonic equivalents. This option may be useful for making your host-language source file smaller.
- **z** prints the version number of **gpre** and the version number of all declared databases. These databases can be declared either in the program or with the **database** option.

Specifying Specific Options

You can specify options that apply exclusively to C, COBOL, and Ada programs:

e[**ither_case**] makes **gpre** work on both uppercase and lowercase characters in C programs. You should use the **either_case** option whenever you use lower case letters in GDML or SQL keywords. If you mix cases and fail to preprocess the program with this option, **gpre** won't be able to process your input file.

Since the other languages InterBase supports are case-insensitive, you don't need to use this option when you preprocess programs written in those languages.

- **n**[**o_lines**] suppresses line numbers for C programs.
- **s**[**tring**] generates fixed-length strings for C programs.
- **h**[**andles**] specifies an Ada handle package.
- **ansi** indicates that the source file is in ANSI COBOL format.

Gpre Examples

Examples of **gpre** preprocessing commands are shown below for the following types of programs:

- Programs with a **database** declaration
- Programs with no **database** declaration

Example 1 — Programs with a Database Declaration

Table 18-3 shows examples of **gpre** commands that preprocess programs containing GDML **database** declarations. When you preprocess these programs, don't use the **database** option.

Table 18-3. Programs with a Database Declaration

Operating System	Language	Sample Command
Apollo AEGIS	Ada (Alsys)	gpre -al -h handles.eada emp.eada
	C	gpre -c -e ursus
	Pascal	gpre -pa ours.epas
	FORTRAN	gpre -f medved
UNIX	Ada	gpre -a -h handles.eada emp.eada
	C	gpre -c ursus
	Pascal	gpre -pascal ours
	FORTRAN	gpre -fortran medved.ftn
VMS	Ada	gpre /a /h handles.eada emp.eada
	BASIC	gpre -b samp.ebas
	C	gpre /c /either ursus
	Pascal	gpre /pascal ours.epas
	FORTRAN	gpre /fortran medved

Example 2 — Programs with no Database Declaration

The table below shows examples of **gpre** commands that preprocess programs that don't contain GDML **database** declarations. When you preprocess these programs, you must use the **database** option.

Table 18-4. Programs with no Database Declaration

Operating System	Language	Sample Command
Apollo AEGIS	C	gpre -c -e -database bears.gdb ursus
	Pascal	gpre -pa -d bears.gdb ours.epas
	FORTRAN	gpre -f -d bears.gdb medved
UNIX	C	gpre -c -d bears.gdb ursus
	Pascal	gpre -pascal -database bears.gdb ours
	FORTRAN	gpre -fortran -d bears.gdb medved.ftn
VMS	C	gpre /c /either /d bears.gdb ursus
	Pascal	gpre /pascal /d bears.gdb ours.epas

For More Information

For information on compiling and linking your program, refer to Chapter 19, *Compiling and Linking Your Program*.

For information on **gpre** error handling, refer to the Appendix in the *Programmer's Reference*.

Chapter 19
Compiling and Linking Your Program

This chapter describes how to compile and link your preprocessed program.

Overview

After you preprocess your program, you must compile and link it. The compile process creates an object module from the preprocessed source file. You use your host language compiler to compile your program.

The linking process resolves external references and creates an executable object. You use whatever link tools your environment provides to link your program's object module to other object modules and libraries. This is usually based on the operating system and hardware you use. It can also be based on the host language in which you write your program.

Considerations for Compiling Your Program

Special considerations for compiling Ada and C programs are described below.

Ada Considerations

Before you compile an Ada program, be sure the Ada library contains the package *interbase.ada* (or *interbase.a* for VERDIX Ada). This package resides in the InterBase *include* directory.

If you want to use the programs in the InterBase examples directory, you must also use the package *basic_io.ada* (or *basic_io.a* for VERDIX Ada). This package resides in the InterBase *examples* directory.

C Considerations

When compiling C programs under VMS, you must specify the **/g_float** option.

Considerations for Linking Your Program

Special considerations for linking your object modules are presented below.

Apollo Considerations

You don't need to link to a library in order to resolve InterBase calls. Instead, you can do one of the following:

- Inlib **gdslib** to your current process.
- Use the **bind inlib** option to bring InterBase into the bound program.
- Use the **mergbss** option on the bind under SR10.

Under SR10, the **gdslib** library is permanently installed on your node through *sys.conf*. Therefore, you don't need to inlib **gdslib** to your current process or use the **bind inlib** option.

UNIX Considerations

You can link your program to one of the following UNIX libraries:

- A library that uses pipes, obtained with the **lgds** option. This library yields faster links and smaller images. It also lets your application work with new versions of InterBase automatically when they are installed.
- A library that does not use pipes, obtained with the **lgds_b** option. This library has faster execution, but binds your application to a specific version of Interase. When you install a new version, you must relink your program to use the new features.

Under SunOS-4, you can link your programs to a shareable library by using the **lgdslib** option. This creates a dynamic link at runtime and yields smaller images with the execution speed of the full library. It also has the ability to upgrade InterBase versions automatically.

VMS Considerations

When you link a C program under VMS, you must specify a link options file to direct the linker to the correct version of the VAX C runtime library. Because InterBase C programs are compiled to use g-float format large precision floating point numbers, they require a special version of the runtime library under VMS.

To avoid accidentally including the wrong runtime library, you must also specify the InterBase library in this order:

```
sys$library!vaxcrtlg.exe/share
sys$library!gdsshr/share
```

If you specify these files in the wrong order, or omit the *vaxcrtlg* file, your program appears to link correctly, but will fail when you attempt to run it.

When you link a non-C program under VMS, you don't need to reference a library in order to resolve InterBase calls. This is because the file *gdsshr.exe* is placed in *sys$library* and registered in *sys$library:imagelib.olb* by the InterBase installation process.

For More Information

For information on preprocessing your program, refer to Chapter 18, *Preprocessing Your Program*.

For more information on compiling and linking your program, refer to your language- and platform-specific documentation.

Appendix

Appendix A
Sample Database Definition

The definition of the atlas database, which is used in many documentation examples, is shown below:

```
define database "atlas.gdb";

        {The atlas database is the sample database used
throughout the documentation set. It is based on a North
American atlas and gazeteer. Type "show relations" at the QLI
prompt for a listing of the relations in the database.}
  page_size 1024;


/* Global Field Definitions */

define field ALTITUDE           long;
define field AREA               long;
define field AREA_CODE          char [3];
define field AREA_NAME          varying [20];
define field CAPITAL            varying [25];
```

```
define field CENSUS_1950          long;
define field CENSUS_1960          long;
define field CENSUS_1970          long;
define field CENSUS_1980          long;
define field CENTER_FIELD         long;
define field CITY                 varying [25];
define field CODE                 varying [4];
define field COMMENTS             blob
                                   segment_length 60;
define field ELECTED_APPT         char [1]
  valid if (elect_appt = 'E'
    or elected_appt = 'A'
    or elected_appt missing);
ddefine field F1                  blob;
define field F2                   blob;
define field F3                   blob;
define field FIRST_NAME           varying [10];
define field FLAG                 char [1]
  valid if (flag = 'Y'
    or flag = 'N'
    or flag missing);
define field GUIDEBOOK            blob
                                   segment_length 60;
define field HOME_STADIUM         varying [30];
define field INCORPORATION        date
define field INIT_TERM            date;
define field LAST_NAME            varying [20];
define field LATITUDE             long;
define field LATITUDE_COMPASS     char [1]
  missing_value is "x";
define field LATITUDE_DEGREES     varying [3]
  missing_value is -1;
define field LATITUDE_MINUTES     char [2]
  missing_value is -1;
define field LEAGUE               char [1];
define field LEFT_FIELD           long;
define field LENGTH               long;
define field LOCATION             blob
                                   segment_length 60;
define field LONGITUDE            long;
define field MIDDLE_INITIAL       char [1];
define field NAME varying         [20];
define field NUM_TRAILS           long;
```

```
define field OFFICE                blob
                                   segment_length 40;
define field OUTFLOW               varying [30];
define field PARTY_AFFILIATION     char [1];
define field PHONE                 char [10]
  edit_string "(xxx)Bxxx-xxxx";
define field POL_TYPE              char [1];
define field POPULATION            long;
define field POSTAL_CODE           char [10];
define field PROVINCE              varying [4];
define field RIGHT_FIELD           long;
define field RIVER                 varying [30];
define field SEATING               long;
define field STATE                 varying [4];
define field STATEHOOD             date;
define field STATE_NAME            varying [25];
define field SURFACE               char [1];
define field TEAM_NAME             varying [15];
define field TRAILS_LIGHTED        long
  query_name LIT;
define field TRAILS_SET            long;
define field TYPE                  char [1]
  valid if (type = 'N' or
    type = 'A' or
    type = 'B');
define field YEAR                  char [4];
define field YEAR_FOUNDED          char [4];
define field ZIP                   varying [10];

/* Relation Definitions */

define relation BASEBALL_TEAMS
   TEAM_NAME                       position 0,
   CITY                            position 1,
   STATE                           position 2,
   HOME_STADIUM                    position 3,
   LEAGUE                          position 4,
   LEFT_FIELD                      position 5,
   CENTER_FIELD                    position 6,
   RIGHT_FIELD                     position 7,
   SEATING                         position 8,
   SURFACE                         position 9;
```

```
define relation CITIES
   CITY                                          position 0,
   STATE                                         position 1,
   POPULATION                                    position 2,
   ALTITUDE                                      position 3,
   LATITUDE_DEGREES                              position 6
     query_name LATD,
   LATITUDE_MINUTES                              position 7
     query_name LATM,
   LATITUDE_COMPASS                              position 8
     query_name LATC,
   LONGITUDE_DEGREES
     based on LATITUDE_DEGREES                   position 9
     query_name LONGD,
   LONGITUDE_MINUTES
     based on LATITUDE_MINUTES                   position 10
     query_name LONGM,
   LONGITUDE_COMPASS
     based on LATITUDE_COMPASS                   position 11
     query_name LONGC,
   LATITUDE
     computed by (latitude_degrees | ' ' |
     latitude_minutes | latitude_compass)        position 4
    LONGITUDE
     computed by (longitude_degrees | ' ' |
     longitude_minutes | longitude_compass)      position 5;

define relation CROSS_COUNTRY
    AREA_NAME                                    position 0,
    CITY                                         position 1,
    STATE                                        position 2,
    PHONE                                        position 3
  edit_string "(xxx)Bxxx-xxxx",
    NUM_TRAILS                                   position 4,
    TRAILS_SET                                   position 5,
    TRAILS_LIGHTED                               position 6,
    INSTRUCTION
  based on FLAG                                  position 7
  query_header "INST",
    RENTALS
  based on FLAG                                  position 8
  query_header "RENT",
    REPAIRS
```

```
    based on FLAG                                   position 9
    query_header "REP",
      FOOD
    based on FLAG                                   position 10,
      LODGE
    based on FLAG                                   position 11
    query_header "BEDS",
      PACKAGES
    based on FLAG                                   position 12
    query_header "PKG",
      GUIDED_TOURS
    based on FLAG                                   position 13
    query_header "TOUR",
      COMMENTS                                      position 14;

define relation MAYORS
      CITY                                          position 0,
      STATE                                         position 1,
      PARTY_AFFILIATION                             position 3
    query_name PARTY
    query_header "party",
      INIT_TERM                                     position 4,
      ELECT_APPT                                    position 5,
      FIRST_NAME                                    position 6,
      MIDDLE_INITIAL                                position 7,
      LAST_NAME                                     position 8,
      MAYOR_NAME
    computed by (first_name | ' ' |
    last_name)                                      position 2;

define relation POLITICAL_SUBDIVISIONS
    CODE                                            position 0,
    NAME                                            position 1,
    AREA                                            position 3,
    INCORPORATION                                   position 4,
    CAPITAL                                         position 5,
    POL_TYPE;

define relation POPULATIONS
      STATE                                         position 0,
      CENSUS_1950                                   position 1,
      CENSUS_1960                                   position 2,
      CENSUS_1970                                   position 3,
      CENSUS_1980                                   position 4;
```

```
define relation POPULATION_CENTER
    YEAR                                        position 0,
    LATITUDE_DEGREES                            position 3,
    LATITUDE_MINUTES                            position 4,
    LATITUDE_COMPASS                            position 5,
    LONGITUDE_DEGREES
  based on LATITUDE_DEGREES                     position 6,
    LONGITUDE_MINUTES
  based on LATITUDE_MINUTES                     position 7,
    LONGITUDE_COMPASS
  based on LATITUDE_COMPASS                     position 8,
    LOCATION                                    position 9,
    LATITUDE
  computed by (latitude_degrees | ' ' |
  latitude_minutes | latitude_compass),
    LONGITUDE
   computed by (longitude_degrees | ' ' |
   longitude_minutes | longitude_compass);

define relation PROVINCES
    PROVINCE                                    position 0,
    PROVINCE_NAME
  based on STATE_NAME                           position 1,
    AREA                                        position 2,
    CAPITAL
   based on CITY                                position 3;


define relation RIVERS
    RIVER                                       position 0,
    SOURCE
   based on PROVINCE                            position 1,
    OUTFLOW                                     position 2,
    LENGTH                                      position 3;

define relation RIVER_STATES
    STATE                                       position 0,
    RIVER                                       position 1;

define relation SKI_AREAS
    NAME                                        position 0,
    TYPE                                        position 1,
    CITY                                        position 2,
    STATE                                       position 3;
```

```
define relation STATES
    STATE                                          position 0,
    STATE_NAME                                     position 2,
    AREA                                           position 3,
    STATEHOOD                                      position 4,
    CAPITAL
  based on CITY                                    position 5;

define relation TOURISM
    STATE                                          position 0,
    ZIP                                            position 1,
    CITY                                           position 2,
    OFFICE                                         position 3,
    GUIDEBOOK                                      position 4;

/* View  Definitions */

define view CITY_TON of c in cities
with c.city matching '*ton*'
  C.CITY                                           postion 0,
  C.STATE                                          postion 1,
  C.POPULATION                                     position 2;

define view LARGE_NON_CAPITALS of s in states
cross c in cities over state
cross cs in cities with cs.state = c.state and
cs.city = s.capital and cs.population < c.population
  C.CITY                                           position 0,
  S.STATE_NAME                                     position 1,
  S.CAPITAL                                        position 2;

define view LT_AVG_CITIES of c in cities
with c.population < average c1.population of c1 in cities
  C.CITY                                           position 0,
  C.STATE                                          position 1;

define view MIDDLE_AMERICA of c in cities
with c.longitude_degrees between 79 and 104
and c.latitude_degrees between 33 and 42
  C.CITY                                           position 0,
  C.STATE                                          position 1,
  C.ALTITUDE                                       position 2;
```

```
define view POPULATION_DENSITY of p in populations
cross s in states over state
    P.STATE                                    position 0,
    DENSITY_1950
  computed by (p.census_1950/s.area)           position 1,
    DENSITY_1960
  computed by (p.census_1960/s.area)           position 2,
    DENSITY_1970
  computed by (p.census_1970/s.area)           position 3,
    DENSITY_1980
  computed by (p.census_1980/s.area)           position 4;

define view PROVINCE_VIEW of p in political_subdivisions
with p.pol_type = 'P'
  PROVINCE FROM P.CODE                         position 0,
  PROVINCE_NAME FROM P.NAME                    position 1,
  P.AREA                                       position 2,
  P.CAPITAL                                    position 3;

define view SKI_CITIES of s in states
cross ski in ski_areas with s.state = ski.state
  SKI.NAME                                     position 0,
  SKI.CITY                                     position 1,
  S.STATE_NAME                                 position 2;


define view SKI_STATES of c in cross_country
reduced to c.state
  C.STATE                                      position 0;

define view SMALLER_CITIES of c in cities
with c.population < 500000
  C.CITY                                       position 0,
  C.STATE                                      position 1,
  C.POPULATION                                 position 2;

define view SMALL_CAPITAL_CITY of s in states
cross c in cities over state
cross cs in cities with cs.state = c.state and
cs.city = s.capital and cs.population < c.population
reduced to s.state, s.caital
  S.STATE_NAME                                 position 0,
  S.CAPITAL                                    position 1;
```

```
define view SMALL_CITY_TEAMS of b in baseball_team
cross c in cities with b.city = c.city and
b.state = c.state and b.seating > c.population / 10
   C.CITY                                          position 0,
   C.STATE                                         position 1,
   B.SEATING                                       position 2,
   C.POPULATION                                    position 3;

define view STATE_VIEW of p in political_subdivisions
with p.pol_type = 'S'
   STATE FROM P.CODE                               position 0,
   STATE-NAME FROM P.NAME                          postion 1,
   P.AREA                                          position 2,
   STATEHOOD FROM P.INCORPORATION                  position 3,
   P.CAPITAL                                       position 4;

define view VARIED_XC of c in cross_country
with c.comments containing 'varied'
   C.AREA_NAME                                     position 0,
   C.STATE                                         position 1,
   C.COMMENTS                                      position 2;

define view VILLES of c in cities
with c.city containing 'ville'
   C.CITY                                          position 0,
   C.STATE                                         position 1,
   C.POPULATION                                    position 2;

/* Index  Definitions */

define index BBT1 for BASEBALL_TEAMS unique
   TEAM_NAME,
   CITY;

define index DUPE_CITY for CITIES
   STATE;

define index CITIES_1 for CITIES unique
   CITY,
   STATE;

define index MAYORS_1 for MAYORS unique
   CITY,
   STATE;
```

```
define index RIV1 for RIVERS unique
   RIVER,
   SOURCE;

define index STATE_1 for STATES unique
   STATE;

define index XXX for TOURISM unique
   STATE;

/* Trigger Definitions*/

define trigger cascading_store for CROSS_COUNTRY
   pre store 0:
   begin
      if not any c in cities
         with c.city = new.city and c.state = new.state
      store x in cities
         x.city = new.city;
         x.state = new.state;
      end_store;
   end;
end_trigger;
```

Appendix B
Using InterBase with Ada

This section describes how to write an Ada program which uses InterBase. A sample Ada package is used to illustrate the process. The package is based on "sql.ea", supplied as a sample program in */usr/interbase/examples*. To write your own Ada program, follow these steps:

1. To make all InterBase-supplied symbols available for use within your program, declare the InterBase package to the Ada compiler:

   ```
   WITH interbase;
   ```

   If your program has only one compilation unit or you do not plan to share databases and transactions with other compilation units, you may skip steps 2, 3, and 4.

2. Create an Ada package which declares handles for all transactions and databases to be shared with other compilation units.

   For example, the package POP_HANDLES declares one transaction handle and one database handle to be shared between compilation units:

   ```
   PACKAGE pop_handles IS
   gds_trans: interbase.transaction_handle := 0;
   ```

```
DB     : interbase.database_handle := 0;
END pop_handles;
```

If you have not declared an explicit database handle on the DATABASE statement, you should use the default, "gds_database". If you have not declared an explicit transaction handle on the START_TRANSACTION statement, you should use the default, "gds_trans".

3. Declare any packages created in step 2 to the Ada compiler:

```
WITH pop_handles;
```

4. Identify any packages created in step 2 to **gpre**, the InterBase preprocessor. Use the HANDLES clause on one database statement within each compilation unit:

```
DATABASE DB = "atlas.gdb" HANDLES "pop_handles";
```

Alternatively, you may use the "handles" switch when invoking **gpre**:

On VMS:

```
% gpre sql.eada /handles=pop_handles
```

On other systems:

```
% gpre sql.ea -handles pop_handles
```

Gpre recognizes the name of only one handle declaration package per Ada compilation unit. **Gpre** prints out a warning if you:

— Specify more than one handle package on the DATABASE statement or the "handles" switch

— Use both the "handles" switch and a DATABASE statement with the HANDLES option

— Specify more than one handle package on DATABASE statements within a compilation unit

Sample Program

The following is a shortened version of "sql.ea", which has been divided into separate compilation units to demonstrate how an Ada program shares database and transaction handles. The program consists of POP1 (the main program), and POP2 (a separate compilation unit which extrapolates missing data).

Below is the code for POP2. Note the reference to the package POP_HANDLES in both the WITH statement and the HANDLES clause of the database statement.

```
WITH basic_io, interbase, pop_handles;


PACKAGE pop2 IS


FUNCTION SETUP RETURN integer;
END pop2;
PACKAGE BODY pop2 IS


DATABASE DB = "atlas.gdb" HANDLES "pop_handles";


---Set up to catch SQL errors.
EXEC SQL
WHENEVER SQLERROR GO TO Error_processing;


--- "setup" function returns 1 if successful, 0 otherwise
FUNCTION SETUP RETURN integer IS
   success: integer;

BEGIN


---Replace missing 1960 census data with estimates
   success := 1;
   EXEC SQL
   UPDATE POPULATIONS
   SET CENSUS_1960 = 0.3 * (CENSUS_1980 - CENSUS_1950)
   WHERE CENSUS_1960 IS NULL;


---Replace missing 1970 census data with estimates
   EXEC SQL
      UPDATE POPULATIONS
   SET CENSUS_1970 = 0.65 * (CENSUS_1980 - CENSUS_1950)
   WHERE CENSUS_1970 IS NULL;
   RETURN success;


<<Error_processing>>
   success := 0;
   IF SQLCODE /= 0 THEN
```

```
    basic_io.put ("Data base error, SQLCODE = ");
    basic_io.put (SQLCODE);
    basic_io.new_line;
    END IF;
    RETURN success;

END SETUP;
END pop2;
```

The code for the main module, POP1 is:

```
WITH basic_io, interbase, pop_handles, pop2;
PROCEDURE pop1 IS
DATABASE DB = "atlas.gdb" HANDLES "pop_handles";

EXEC SQL
WHENEVER SQLERROR GO TO Error_processing;

--- generic routine to print SQLCODE and database status
vector
PROCEDURE print_error IS
BEGIN
basic_io.put ("Data base error, SQLCODE = ");
basic_io.put (SQLCODE);
basic_io.new_line;
interbase.print_status (gds_status);
END print_error;

--- routine to retrieve and print all census data
PROCEDURE census_report IS
pop_50, pop_60, pop_70, pop_80: INTEGER;
stname: BASED_ON STATES.STATE_NAME;
BEGIN

--- Declare cursor for use in retrieving state/census data

EXEC SQL
DECLARE C CURSOR FOR
SELECT STATE_NAME, CENSUS_1950, CENSUS_1960,
```

```
CENSUS_1970, CENSUS_1980
FROM STATES S, POPULATIONS P
WHERE S.STATE = P.STATE
ORDER S.STATE_NAME;


EXEC SQL
OPEN C;


---Until end of data stream is reached, print out current
---set of data and then fetch next set
WHILE SQLCODE = 0 LOOP
EXEC SQL
FETCH C INTO :stname, :pop_50, :pop_60, :pop_70, :pop_80;


basic_io.put (stname);
basic_io.put (pop_50);
basic_io.put (pop_60);
basic_io.put (pop_70);
basic_io.put (pop_80);
basic_io.new_line;
END LOOP;


EXEC SQL
CLOSE C;


<<Error_processing>>
IF SQLCODE /= 0 THEN
print_error;
END IF;
END census_report;


BEGIN --- pop1


---Set up estimated data; if successful then do report
IF pop2.setup = 1 THEN
census_report;
END IF;
```

```
---Undo estimated census data
EXEC SQL
ROLLBACK;


<<Error_processing>>
IF SQLCODE /= 0 THEN
print_error;
END IF;


END pop1;
```

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W