HP 9000 Computer Systems

ALLBASE/SQL Pascal
Application Programming Guide

Printing History

The following table lists the printings of this document, together with the respective release dates for each edition. The software version indicates the version of the software product at the time this document was issued. Many product releases do not require changes to the document. Therefore, do not expect a one-to-one correspondence between product releases and document editions.

Edition	Date	Product Number	Hardware Series	HP-UX Release
First Edition	June 1987	36217-02A.01.00	800	1.2
Update 1	June 1987	36217-02A.01.03	800	1.2
First Edition	February 1988	79725-1.01	300	2.0
Second Edition	August 1987	36217-02A.01.03	800	2.0
Second Edition	February 1989	79725-2.00	300	6.5
Third Edition	February 1988	36217-02A.B2.00.03	800	3.0
Fourth Edition	June 1991	36217-02A.07.00	800	7.0
		HP79725 Rev.7.0	300	7.0
Fifth Edition	September 1992	36216-02A.E1.00	800	8.0
		36216-02A.E1.00	700	8.0
		HP79725A.E1.00	300	8.0
		HP79725A.E1.00	400	8.0

Note   The E.1 release of ALLBASE/SQL is compatible with all 8.0 releases of the HP-UX operating system. The exact correspondence of ALLBASE/SQL release, hardware platform, and HP-UX release is shown above.

ALLBASE/SQL HP-UX Documents

General Reference

Up and Running with ALLBASE/SQL (36389-90011)	ALLBASE/SQL Reference Manual (36217-90001)	ALLBASE/ISQL Reference Manual (36217-90004)
ALLBASE/SQL Quick Reference Guide (36217-90012)	ALLBASE/SQL Message Manual (36217-90009)	ALLBASE/Query Manuals (92534-64001)

Database And Network Administration

ALLBASE/SQL Database Administration Guide (36217-90005)	ALLBASE/NET User's Guide (36217-90093)	ALLBASE/SQL Performance Guidelines (36389-90001)

LG200145_003a

ALLBASE/SQL HP-UX Documents

Embedded SQL Programming Guides

ALLBASE/SQL C Application Programming Guide (36217-90014)	ALLBASE/SQL FORTRAN Application Programming Guide (36217-90013)	ALLBASE/SQL Pascal Application Programming Guide (36217-90007)	ALLBASE/SQL COBOL Application Programming Guide (36217-90058)

ALLBASE/SQL Release F.0 Application Programming Bulletin for HP-UX (36217-90157)

4GL Programming Guides

HP ALLBASE/4GL Installation Guide (30662-90002)	HP ALLBASE/4GL Self-Paced Training Guide (30662-60004)	HP ALLBASE/4GL Developer Reference Manual, V.1 (30662-60002)	HP ALLBASE/4GL Developer Reference Manual, V.2 (30662-60003)
HP ALLBASE/4GL Run-Time Administration Manual (92441-64050)	HP ALLBASE/4GL Developer Administration Manual (30662-60001)	HP ALLBASE/4GL Developer Quick Reference Guide (30662-90005)	HP ALLBASE/4GL Software Update Notice of B.04 (30662-90003)

LG200145_003b

Preface

ALLBASE/SQL is a relational database management system for use on the HP 9000 Series
800 computer. ALLBASE/SQL (Structured Query Language) is the language you use to
define and maintain data in an ALLBASE/SQL DBEnvironment. This manual presents the
techniques of embedding ALLBASE/SQL within Pascal language source code. When Series
300 and 400 implementation differs from that of Series 700 and 800, Series 700 and 800
information is given in the body of the text, followed by a note in this format:

300/400   *Series 300 and Series 400 information appears here.*

This manual is intended as a learning tool and a reference guide for Pascal programmers. It
presumes the reader has a working knowledge of Pascal, the HP-UX operating system, and
ALLBASE/SQL relational database concepts.

This manual contains both basic and in-depth information about embedding ALLBASE/SQL.
Topics are discussed in separate chapters, as follows:

- Chapter 1, "Getting Started with ALLBASE/SQL Pascal Programming," is an introduction
  to ALLBASE/SQL programming which includes information on developing, using, and
  maintaining programs on the HP-UX operating system.
- Chapter 2, "Using the ALLBASE/SQL Pascal Preprocessor," explains the ALLBASE/SQL
  preprocessor and how to invoke it.
- Chapter 3, "Embedding SQL Commands," gives rules for how and where to embed SQL
  commands.
- Chapter 4, "Host Variables," describes how to define and use variables to transfer data
  between your Pascal program and an ALLBASE/SQL DBEnvironment.
- Chapter 5, "Runtime Status Checking and the SQLCA," defines ways to monitor and
  handle successful and unsuccessful SQL command execution.

Chapters 6 through 13 address the various ways to manipulate data in an ALLBASE/SQL
Pascal program.

- Chapter 6, "Overview of Data Manipulation," is an overview of data manipulation and the
  technniques for executing data manipulation commands.
- Chapter 7, "Simple Data Manipulation," explains how to operate on one row at a time.
- Chapter 8, "Processing with Cursors," shows how to process a multiple row query result one
  row at a time.
- Chapter 9, "BULK Table Processing," examines the processing of multiple rows at a time.
- Chapter 10, "Using Dynamic Operations," describes the use of ALLBASE/SQL commands
  that are preprocessed at runtime.
- Chapter 11, "Programming with Constraints," compares the use of statement level integrity
  and row level integrity and discusses the use of integrity constraints.
- Chapter 12, "Programming with LONG Columns," discusses the LONG BINARY and
  LONG VARBINARY data types.
- Chapter 13, "Programming with ALLBASE/SQL Functions," contains descriptions of SQL
  functions, including date/time functions and the TID function.

Most of the examples are based on the tables, views, and other objects in the sample
database, PartsDBE. For complete information about PartsDBE, refer to the *ALLBASE/SQL
Reference Manual*, appendix C.

Conventions

UPPERCASE   In a syntax statement, commands and keywords are shown in
             uppercase characters. The characters must be entered in the order
             shown; however, you can enter the characters in either uppercase or
             lowercase. For example:

    `COMMAND`

can be entered as any of the following:

    `command`   `Command`   `COMMAND`

It cannot, however, be entered as:

    `comm`   `com_mand`   `comamnd`

italics   In a syntax statement or an example, a word in italics represents a
             parameter or argument that you must replace with the actual value.
             In the following example, you must replace *filename* with the name
             of the file:

    `COMMAND` *filename*

punctuation In a syntax statement, punctuation characters (other than brackets,
             braces, vertical bars, and ellipses) must be entered exactly as shown.
             In the following example, the parentheses and colon must be entered:

    (*filename*):(*filename*)

underlining Within an example that contains interactive dialog, user input and
             user responses to prompts are indicated by underlining. In the
             following example, yes is the user's response to the prompt:

    `Do you want to continue? >>` <u>`yes`</u>

{   }   In a syntax statement, braces enclose required elements. When
             several elements are stacked within braces, you must select one. In
             the following example, you must select either `ON` or `OFF`:

    `COMMAND` $\left\{ \begin{array}{l} \mathtt{ON} \\ \mathtt{OFF} \end{array} \right\}$

[   ]   In a syntax statement, brackets enclose optional elements. In the
             following example, `OPTION` can be omitted:

    `COMMAND` *filename* `[OPTION]`

When several elements are stacked within brackets, you can select
one or none of the elements. In the following example, you can select
`OPTION` or *parameter* or neither. The elements cannot be repeated.

    `COMMAND` *filename* $\left[\begin{array}{l} \mathtt{OPTION} \\ \mathit{parameter} \end{array} \right]$

Conventions (continued)

[...]   In a syntax statement, horizontal ellipses enclosed in brackets
indicate that you can repeatedly select the element(s) that appear
within the immediately preceding pair of brackets or braces. In the
example below, you can select *parameter* zero or more times. Each
instance of *parameter* must be preceded by a comma:

    [, *parameter*] [...]

In the example below, you only use the comma as a delimiter if
parameter is repeated; no comma is used before the first occurrence
of *parameter*:

    [*parameter*] [, ...]

| ... | In a syntax statement, horizontal ellipses enclosed in vertical bars
indicate that you can select more than one element within the
immediately preceding pair of brackets or braces. However, each
particular element can only be selected once. In the following
example, you must select **A, AB, BA**, or **B**. The elements cannot be
repeated.

$$\left\{ \begin{array}{c} \texttt{A} \\ \texttt{B} \end{array} \right\} |\ \ ...\ \ |$$

. . .   In an example, horizontal or vertical ellipses indicate where portions
of an example have been omitted.

Δ   In a syntax statement, the space symbol Δ shows a required blank.
In the following example, *parameter* and *parameter* must be
separated with a blank:

    (*parameter*)Δ(*parameter*)

⬭   The symbol ⬭ indicates a key on the keyboard. For example,
RETURN represents the carriage return key or Shift represents the
shift key.

CTRL*character* CTRL*character* indicates a control character. For example, CTRLY
means that you press the control key and the Y key simultaneously.

Contents

Index

Figures

Tables

Getting Started with ALLBASE/SQL Pascal Programming

The steps in creating a Pascal application program that accesses an ALLBASE/SQL relational database environment (**DBEnvironment**) are summarized in Figure 1-1.
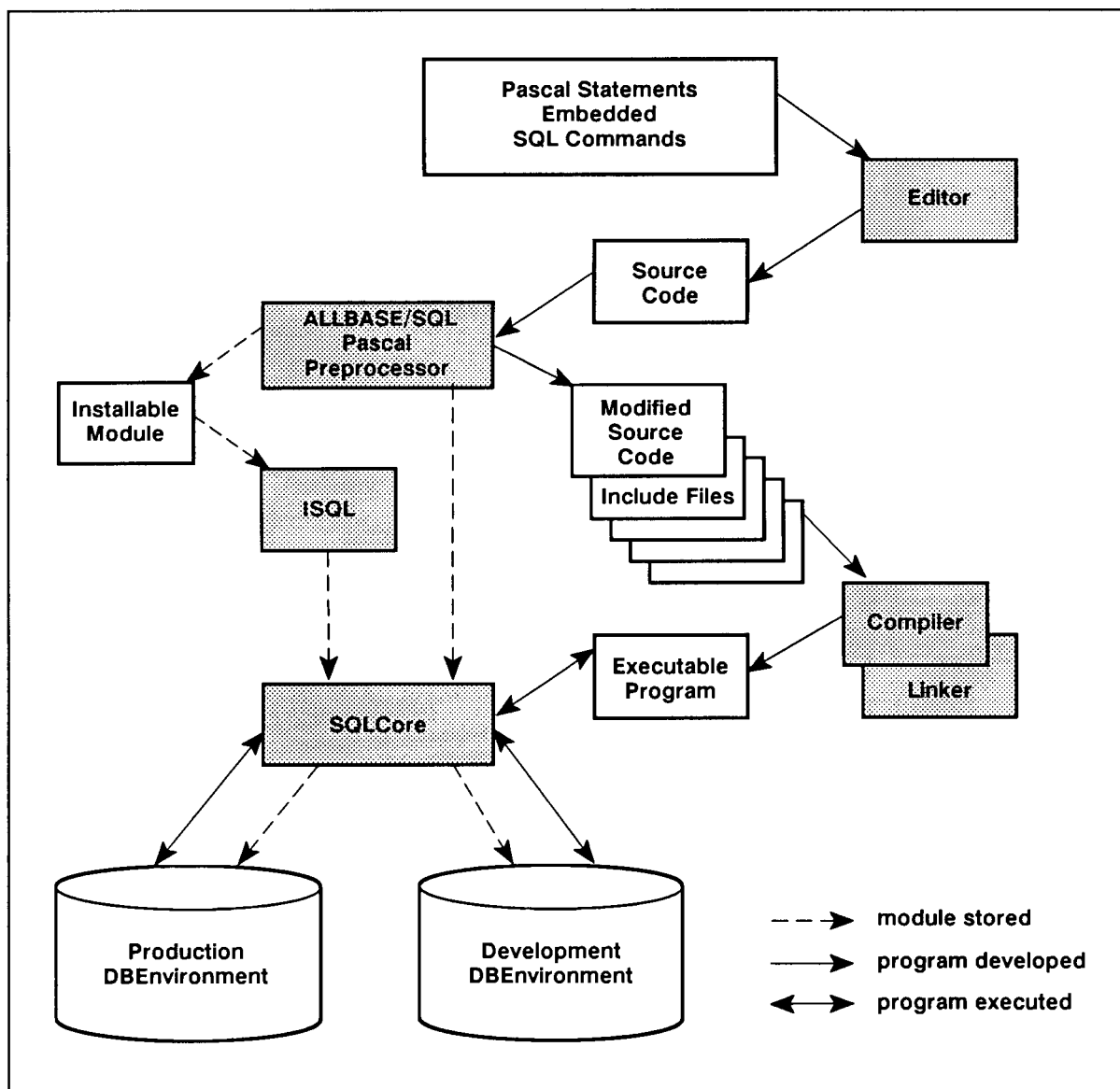
LG200128_001a

Figure 1-1. Creating an ALLBASE/SQL Pascal Application Program

Using your favorite editor, you create Pascal **source code**. The source code is a compilable Pascal program or subprogram that contains SQL commands. The SQL commands contained within the Pascal program are said to be **embedded**. Refer to the *ALLBASE/SQL Reference Manual* for SQL terminology and usage rules.

Before compiling the source code, you **preprocess** it with the ALLBASE/SQL Pascal **preprocessor**. Preprocessing performs the following tasks:

- Checks the syntax of the SQL commands.

- Stores a **module** in the system catalog of the DBEnvironment to be accessed at run time. A module consists of ALLBASE/SQL instructions for executing SQL commands in your program.

- Creates an installable module file. This file contains a copy of the module stored in the DBEnvironment at preprocessing time. You can use this file to install the module into another DBEnvironment in order to run the application program in that DBEnvironment.

- Generates Pascal constructs for executing the SQL commands and comments out the SQL commands. Non-SQL constructs are ignored. This modified version of your source code is placed into a file created by the preprocessor, referred to as a **modified source code** file.

- Creates four **include** files, which contain variable declarations, constant declarations, type declarations, and external procedure declarations that the preprocessor-generated Pascal constructs use.

You use the Pascal **compiler** and **system linker** to create the **executable** program from the modified source code file and the four include files. The executable program automatically makes the appropriate database accesses at run time in the DBEnvironment where the related module is stored.

ALLBASE/SQL Pascal Programs

To write a Pascal application that uses an ALLBASE/SQL database, you embed SQL commands in the Pascal source wherever you want the program to do the following tasks:

- Start or terminate a DBEnvironment session, either in single user mode or multi-user mode.

- Start or terminate a transaction.

- Retrieve rows from or change data in tables in a database.

- Create or drop objects, such as indexes or views.

You also embed special SQL commands known as **preprocessor directives**. The Pascal preprocessor uses these directives to do the following tasks:

- Identify Pascal variables referenced in SQL commands, known as **host variables**.

- Set up a data structure known as the SQL Communications Area (**SQLCA**) in the main program, for communicating the status of executed SQL commands to your program.

- Optionally automate program flow based on SQLCA information.

- Generate error handling code.

- Set up a special variable known as the SQL Description Area (**SQLDA**) in the main program or subprogram, for handling dynamically preprocessed SELECT commands.

- Identify cursor declarations.

Program Structure

The following skeleton program illustrates the relationship between Pascal constructs and embedded SQL commands in an application program. SQL commands may appear in a program at locations indicated by shading.

```
(* PROGRAM HEADING *)
program ProgramName (input, output);
    ⋮
(* PROGRAM DECLARATION PART *)
var
   SQLCA Declaration
   SQLDA Declaration

   Host Variable Declarations
    ⋮
(* PROGRAM STATEMENT PART *)
begin
    ⋮
Pascal statements and  SQL Commands
    ⋮
end.
```

The global area of a subprogram cannot contain host variable declarations. However, any level of a procedure in a subprogram can contain host variable declarations, and you can use the host variables in procedures at other levels.

The ALLBASE/SQL preprocessor does not allow the Pascal language MODULE keyword, or the associated IMPORT and EXPORT keywords.

300/400   Series 300 and 400 systems do not allow subprograms.

To delimit SQL commands for the preprocessor, you begin each command with EXEC SQL and end each command with a semicolon, as follows:

```
EXEC SQL BEGIN WORK;
```

SQL commands can appear within any Pascal procedure where you establish DBEnvironment access and manipulate data in a database.

DBEnvironment Access

You must always specify a DBEnvironment at preprocessing time. The preprocessor needs to access the DBEnvironment you specify in order to store a module containing permanent sections used by your application program at run time. In this example, the DBEnvironment is PartsDBE which is in the sampledb directory.

```
$ psqlpas ../sampledb/PartsDBE -i mysource.sql -p mysource.p -d
```

Your application program needs to access the DBEnvironment to perform its work. The CONNECT command starts a DBEnvironment session for a specific environment. The RELEASE statement terminates that session.

```
EXEC SQL CONNECT TO '../sampledb/PartsDBE';
.
.
.
EXEC SQL RELEASE;
```

At run time, the program starts a DBE session in the DBEnvironment PartsDBE, where one or more modules for the program have been stored.

In some cases an ALLBASE/SQL program is used with one or more DBEnvironments in addition to the DBEnvironment accessed at preprocessing time. In these cases, you use ISQL to install the installable module created by the preprocessor into each additional DBEnvironment accessed by your program. You can also preprocess the same application repeatedly with different DBEnvironments. See Chapter 2 for information on the installable module.

An alternative method of accessing more than one DBEnvironment from the same program would be to separate the program into separate compilable files. Each source file would access a DBEnvironment. In each file you start and terminate a DBE session for the DBEnvironment accessed. You then preprocess and compile each file separately. When you invoke the preprocessor, you identify the DBEnvironment accessed by the source file being preprocessed. After each separate source file is preprocessed, it must be compiled without linking. When all source files have been preprocessed and compiled, you link them to create the executable program.

Note that a program which accesses more than one DBEnvironment must access them one after another. Such program design may adversely affect performance and requires special consideration.

To preprocess or to use an already preprocessed ALLBASE/SQL application program, you must satisfy the authorization requirements for each DBEnvironment accessed.

During preprocessing, the preprocessor needs to access the same DBEnvironment your program will access at run time. Therefore, when you invoke the preprocessor, the DBEnvironment your program accesses must exist and you must identify that DBEnvironment.

```
$ psqlpas mydbe -i mysource.sql -p mysource.p -d
```

In addition, you define a DBEnvironment session for the **same** DBEnvironment with the following source code syntax:

```
   .
   .
   .
   EXEC SQL CONNECT TO 'mydbe';
   .
   .
   .
   EXEC SQL RELEASE;
```

At run time, the program starts a DBE session in mydbe, where a module for the program is stored.

In some cases an ALLBASE/SQL program is used with a DBEnvironment named differently from the DBEnvironment accessed at preprocessing time. In these cases, you can accept a DBEnvironment name into a host variable from the program user and execute the SQL command that starts a DBE session, using the host variable. No dynamic preprocessing is involved. You can also accept the whole CONNECT command and dynamically preprocess the command that starts the DBE session. Dynamic preprocessing is covered in the chapter, "Using Dynamic Operations."

File Referencing

When you create a DBEnvironment, a DataBase Environment Configuration (DBECon) file is created. The absolute pathname of this DBECon file is stored in the DBECon file itself. Once the DBECon file is created, the user, in all subsequent references to DBEnvironment files, may use either an absolute or a relative pathname. By default, all pathnames of files and directories are relative to the pathname stored in the DBECon file. Any file reference which does not begin with a slash (/) assumes a pathname relative to that of the DBECon file.

For example, a DBEnvironment is created with the following command:

```
   START DBE 'PartsDBE' NEW
```

If the user was currently in the directory /users/dbsupport/sql, the pathname /users/dbsupport/sql would be stored in the DBECon file. Suppose the user were subsequently to create a DBEFile with the command:

```
   CREATE DBEFILE Orders WITH PAGES=50, NAME='OrdersDF'
```

The actual pathname of the file OrdersDF would be relative to the pathname stored in the DBECon file and would be /users/dbsupport/sql/OrdersDF. Relative pathnames are restricted to a maximum length of 128 characters. If however, the user were to create a DBEFile with the command:

```
   CREATE DBEFILE Orders WITH PAGES=50, NAME='/users/shipping/OrdersDF'
```

the default pathname stored in the DBECon file would not be used while creating this file. The user would need to fully qualify this pathname each time this file is referenced. Absolute pathnames are restricted to a maximum length of 128 characters.

In addition, if the DBEnvironment you want the preprocessor to access resides in a directory other than your current working directory, you will have to either absolutely or relatively qualify the pathname of that DBEnvironment.

For example, if the DBEnvironment you want the preprocessor to access resides in a subdirectory of yours, you would invoke the preprocessor as follows:

```
$ psqlpas sampledb/SomeDBE -i SourceFileName.sql -d
```

Authorization

ALLBASE/SQL authorization governs who can preprocess, execute, and maintain a program that accesses an ALLBASE/SQL DBEnvironment.

To preprocess a program for the first time, you need CONNECT or DBA authority in your DBEnvironment. When you preprocess a program, ALLBASE/SQL stores a module for that program in the DBEnvironment's system catalog and identifies your userid as the owner of that module. Subsequently, if you have OWNER or DBA authority, you can preprocess the program again.

To run a program accessing an ALLBASE/SQL DBEnvironment, you need the following authorities to start the DBE session in the program:

- If the program uses a CONNECT command to start a DBE session, you need CONNECT authority and RUN or module OWNER authority to run the program.

- If the program uses a START DBE command to start the DBE session, you need DBA authority to run the program.

At run time, any SQL command in the program is executed only if the OWNER of the module has the authorization to execute the command at run time, and the individual running the program has RUN authority to the program. However, any dynamic command is executed only if the userid of the user running the program has the authority to execute the entered command at run time. A **dynamic** command is an SQL command entered by the user at run time.

Maintaining an ALLBASE/SQL program includes such activities as modifying a program in production use and keeping runtime authorizations current as program users change. For these activities, you need OWNER authority for the module or DBA authority. More on this topic appears later in this chapter under "Maintaining ALLBASE/SQL Programs."

Native Language Support

ALLBASE/SQL lets you manipulate databases in a variety of native languages in addition to the default language, known as **n-computer**. You can use either 8-bit or 16-bit character data, as appropriate for the language you select. In addition, you can always include ASCII data in any database, since ASCII is a subset of each supported character set. The collating sequence for sorting and comparisons is that of the native language selected. A list of supported languages is in /usr/lib/nls/config.

You can use native language characters in the following places, including the following places:

- Character literals.
- Host variables for CHAR or VARCHAR data (but not variable names).
- ALLBASE/SQL names.
- WHERE and VALUES clauses.
- Filenames.

If your system has the proper message files installed, ALLBASE/SQL displays prompts, messages and banners in the language you select, and it displays dates and time according to local customs. In addition, ISQL accepts responses to its prompts in the native language selected. However, regardless of the native language used, the syntax of ISQL and SQL commands—including punctuation—remains in ASCII.

In order to use a native language other than the default, you must do the following:

1. Make sure your I/O devices support the character set you wish to use.

2. Set the HP-UX environment variable LANG to the native language (LanguageName) you wish to use.

   For the C shell, use the following command syntax:

       setenv LANG *LanguageName*

   For the K shell, use the following command (no spaces before or after the equal sign):

       typeset -x LANG=*LanguageName*

   For the Bourne shell, use the following commands:

       LANG = *LanguageName*
       export LANG

   This language then becomes the **current** language. (If LANG is not set, the current language is n-computer.)

3. Use the LANG = *LanguageName* option of the START DBE NEW command to specify the language when you create a DBEnvironment.

You can use native language characters in the DBEnvironment name. If you do so, you must set the LANG environment variable to the same language before you can connect to the DBEnvironment. To avoid confusion, it is advised that you always use the same language for the DBEnvironment that you use in the LANG variable.

Resetting the LANG variable while you are connected to a DBEnvironment has no effect on the current DBE session. You must connect to the DBEnvironment again with the new setting.

The ALLBASE/SQL Pascal Preprocessor

The preprocessor generates Pascal constructs, based on embedded SQL commands.

<xref EVENTS>: undefined summarizes these four main preprocess-time events:

- Syntax checking of SQL commands and host variable declarations.

- Creation of compilable files: one modified source code file and four include files.

- Creation of an installable module.

- Storage of a module in the system catalog.
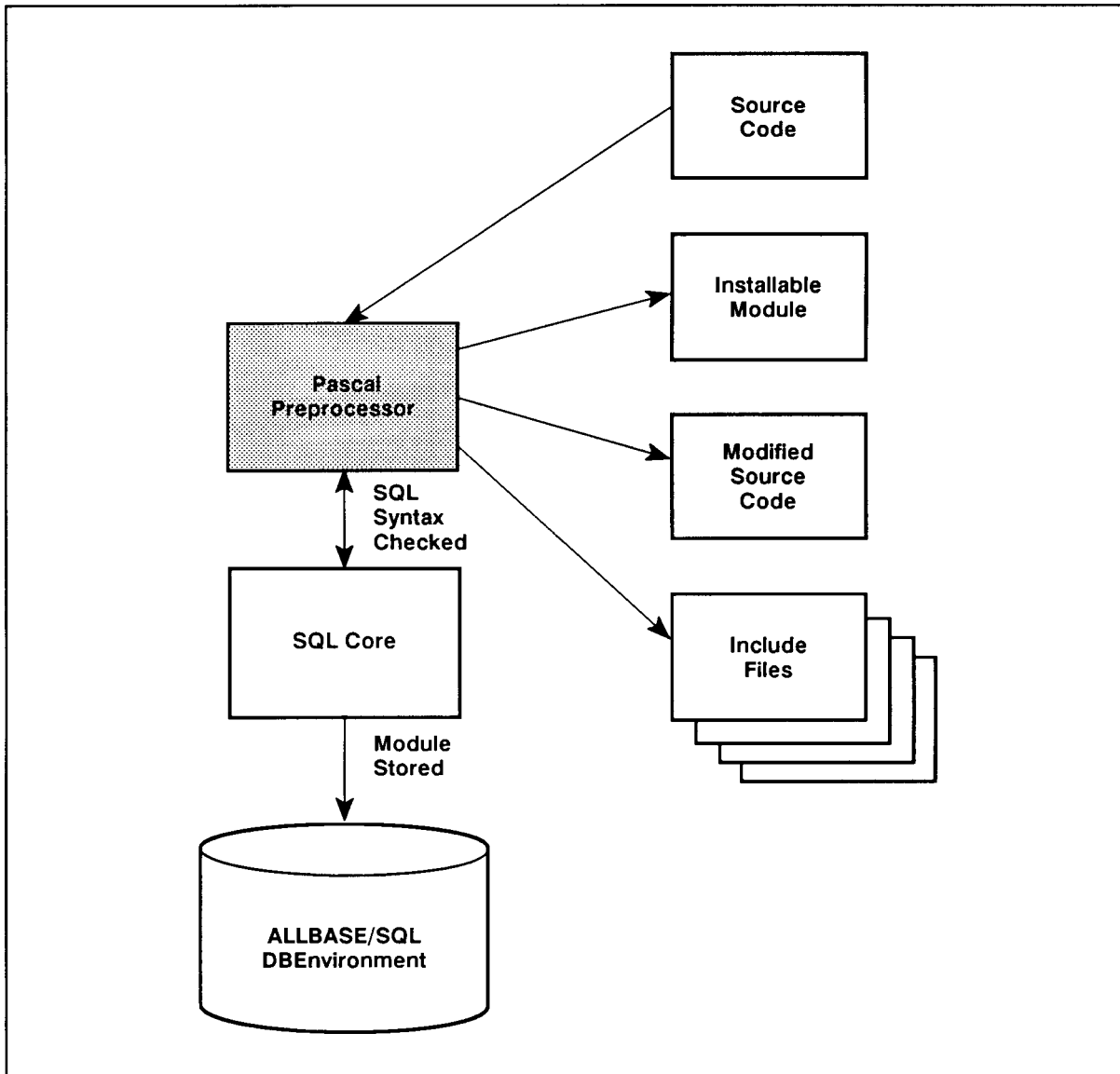
LG200128_002a

Figure 1-2. Preprocess-Time Events

Effect of Preprocessing on Source Code

The Pascal preprocessor scans the source code for SQL commands. If the syntax of an SQL command is valid, the preprocessor converts the command into compilable Pascal constructs that call ALLBASE/SQL external procedures at run time. During preprocessing, for example, ALLBASE/SQL converts the following SQL command:

```
EXEC SQL SELECT PartNumber, PartName, SalesPrice
   INTO :PartNumber,
        :PartName,
        :SalesPrice :SalesPriceInd
   FROM  PurchDB.Parts
  WHERE  PartNumber = :PartNumber;
```

The preprocessor produces the following converted modified source code constructs:

```
begin
SQLTEMPV.REC1.PartNumber1 := PartNumber;
SQLXFET(waddress(SQLCA),SQLOWNER,SQLMODNAME,1,waddress(SQLTEMPV),
        16,64,TRUE);
if SQLCA.SQLCODE = 0 then
  begin
  PartNumber := SQLTEMPV.REC2.PartNumber1;
  PartName := SQLTEMPV.REC2.PartName2;
  if SQLTEMPV.REC2.SalesPriceInd4 >= 0 then
  SalesPrice := SQLTEMPV.REC2.SalesPrice3;
  SalesPriceInd := SQLTEMPV.REC2.SalesPriceInd4;
  end
else
  begin
  end;
end;
```

The preprocessor converts the embedded SELECT command into a Pascal comment and Pascal constructs that enable ALLBASE/SQL to execute the command at run time. The names that appear in the inserted Pascal code identify variables ALLBASE/SQL external procedures uses; in this example, the names identify variables used by the SQLXFET external procedure. Some of these variables are derived from host variables. As shown in the embedded SELECT command above, you precede a host variable with a colon when you use it in SQL commands like this:

```
:SalesPrice
```

Refer to the chapter, "Host Variables," for more information.

Declarations used by preprocessor-generated code are defined in the four include files the preprocessor creates in the following syntax:

- *ModifiedSourceFileName*.sqlc: File defining ALLBASE/SQL constants.

- *ModifiedSourceFileName*.sqlt: File defining ALLBASE/SQL type declarations.

- *ModifiedSourceFileName*.sqlv: File defining variables in a program.

- *ModifiedSourceFileName*.sqle: File defining ALLBASE/SQL's external procedure declarations.

From one to 100 files of the syntax *ModifiedSourceFile*.sv*nn* are generated, one for each declare section; for example, ModifiedSourceFileName.sv01.

The preprocessor inserts INCLUDE statements of the following syntax that reference these files in the declaration section of the modified source code:

```
$include ModifiedSourceFileName.sqlc$
$include ModifiedSourceFileName.sqlt$
$include ModifiedSourceFileName.sqlv$
$include ModifiedSourceFileName.sqle$
```

In place of the *ModifiedFileName*.sqlv include statement syntax, a *ModifiedSourceFile*.sv*nn* include statement is used for multiple declare sections when subprograms are used.

The chapter, "Using the ALLBASE/SQL Pascal Preprocessor," explains how ModifiedSourceFileName is derived.

Caution   Never modify either the constructs inserted by the preprocessor or the include files the preprocessor creates. Changes to preprocessor-generated information could damage your DBEnvironment or your system.

Effect of Preprocessing on DBEnvironments

When you invoke the preprocessor, you name an ALLBASE/SQL DBEnvironment. The preprocessor starts a DBE session for that DBEnvironment when preprocessing begins and terminates that session when preprocessing is completed.

When the preprocessor encounters a syntactically correct SQL command, it usually creates a **section** and stores it in the system catalog of the DBEnvironment being accessed. (Some SQL commands do not create sections.) An ALLBASE/SQL section is a group of stored ALLBASE/SQL instructions for executing one SQL command.

All sections created during the preprocessing of a single unit (main program or subprogram) constitute a module. The preprocessor derives the name of the module from the program heading unless you supply a different name when you invoke the preprocessor. In the following example, mymodule is a different name than mysource:

```
$ psqlpas DBEnvironmentName -m mymodule -i mysource.sql -p mysource.p
```

300/400   Series 300 and 400 systems do not allow subprograms. The main program and the subprograms that comprise an application must each have a unique name. No two modules are allowed to have the same name.

When the preprocessor terminates its DBEnvironment session, it issues a COMMIT WORK command if it encountered no errors. Created sections are stored in the system catalog and associated with the module name.

The Stored Section

A section consists of ALLBASE/SQL instructions for executing an SQL command. The SQL commands that do not generate stored sections are listed in the "Stored Sections" paragraph of the "Using the ALLBASE/SQL Pascal Preprocessor." The preprocessor creates a section and assigns a unique section number for all embedded ALLBASE/SQL commands except:

```
BEGIN DECLARE SECTION          INCLUDE
BEGIN WORK                      OPEN
CLOSE                          PREPARE
COMMIT WORK                    RELEASE
CONNECT                        ROLLBACK WORK
DECLARE CURSOR                 SAVEPOINT
DELETE WHERE CURRENT           START DBE
DESCRIBE                       STOP DBE
END DECLARE SECTION            SQLEXPLAIN
EXECUTE                        TERMINATE USER
EXECUTE IMMEDIATE              UPDATE WHERE CURRENT
                               WHENEVER
```

The preprocessor assigns a number to each section and references the appropriate stored section in the generated code:

```
   .
   .
   .
SQLXFET(Waddress(SQLCA),SQLOWNER,SQLMODNAME,1,Waddress(SQLTEMPV),
   16,64,true);
```

Purpose of Sections

A section serves the following two purposes:

1. Access validation: Before executing a stored section at run time, ALLBASE/SQL ensures that any objects referenced exist and that runtime authorization criteria are satisfied.

2. Access optimization: If ALLBASE/SQL has more than one way to access data, it determines the most efficient method and creates the section based on that method. Indexes, for example, can expedite the performance of some queries.

By creating and storing sections at preprocessing time rather than at run time, runtime performance is improved.

Section Validity

A section is assigned one of two states at preprocessing time: **valid** or **invalid**. A section is valid when access validation criteria are satisfied. If the SQL command references objects that exist at preprocessing time and the individual doing the preprocessing is authorized to issue the command, the stored section is marked as valid. A section is invalid when access validation criteria are not satisfied. If the SQL command references an object that does not exist at preprocessing time or if the individual doing the preprocessing is not authorized to issue the command, the stored section is marked as invalid. After being stored by the preprocessor, a valid section is marked as invalid when such activities as the following occur:

■ Changes in authorities of the module's owner.

■ Alterations to tables accessed by the program.

■ Deletions or creations of indexes.

■ Updating a table's statistics.

At run time, ALLBASE/SQL executes valid sections and attempts to validate any section marked as invalid. If an invalid section can be validated, as when an altered table does not affect the results of a query, ALLBASE/SQL marks the section as valid and executes it. If an invalid section cannot be validated, as when a table reference is invalid because the table owner name has changed, ALLBASE/SQL returns an error indication to the application program.

When a section is validated at run time, it remains in the valid state until an event that invalidates it occurs. The program execution during which validation occurs is slightly slower than program executions following validation.

Compiling and Linking the Program

Figure 1-3 summarizes the steps in creating an executable ALLBASE/SQL Pascal program from the files created by the Pascal preprocessor.
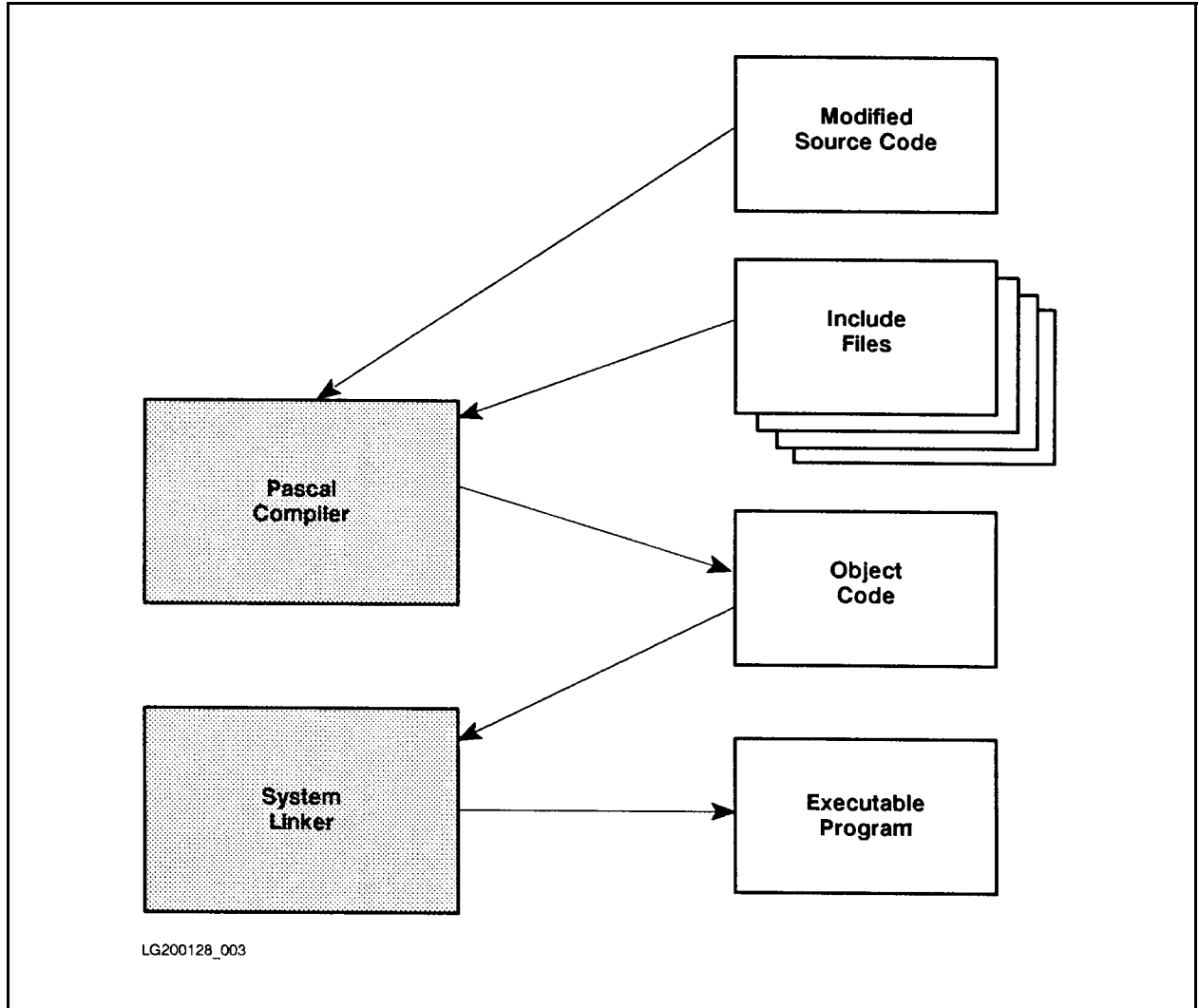
Figure 1-3. Compile-Time and Linking-Time Events

You submit to the Pascal compiler a modified source code file and related include files created by the preprocessor. The compiler then generates an object code file. To convert one or more object code files into an executable program, you link them by invoking the HP-UX Linker. This step creates an executable program file.

In the following example, an executable program named someprog.r is created after a module named somemod is stored by the Pascal preprocessor in a DBEnvironment named somedbe:

```
login: pgmr1

$ psqlpas somedbe -m somemod -i mysource.sql -p mysource.p -d
    .
    .
$ pc mysource.p -lsql -lportnls -o someprog.r
```

300/400   For Series 300 and 400 systems, use the following command, to create the executable module:

```
$ pc mysource.p -lsql -lheap2 -lpc -lportnls -o someprog.r
```

Refer to "Full Preprocessing Mode" in the chapter, "Using the ALLBASE/SQL Pascal Preprocessor" for more information on the psqlpas command.

The -lsql option tells the compiler to link in the runtime library /usr/lib/libsql.a. The -lportnls option tells the compiler to link in the native language routines. The -o option tells the compiler to create the executable output file someprog.r.

ALLBASE/SQL Program Execution

When an ALLBASE/SQL program is first created, it can only be executed by the module OWNER or a DBA. In addition, it can only operate on the DBEnvironment used at preprocessing time if a module was generated. If no module was generated because the SQL commands embedded in the program are only commands for which no sections are created, the program can be run against any DBEnvironment.

The program created in the previous example can be executed as follows by pgmr1:

```
$ someprog.r
```

To make the program executable by other users in other DBEnvironments, you must perform the following steps:

■ Load the executable program file onto the machine where the production DBEnvironment resides.

■ Install any related module in the production DBEnvironment.

■ Ensure necessary module owner authorities exist.

■ Grant required authorities to program users.

Installing the Program Module

When the preprocessor stores a module in a DBEnvironment, it also creates a file containing a copy of the module, which can be installed into another DBEnvironment. You use the INSTALL command in ISQL to install the module in another DBEnvironment:

```
isql=> CONNECT TO 'somedbe';
isql=> INSTALL mysource.sqlm;

Name of module in this file:  PGMR1.SOMEMOD
Number of sections installed:  6
COMMIT WORK to save to DBEnvironment.

isql=> COMMIT WORK;
```

ISQL copies the module from the installable module file named mysource.sqlm into a DBEnvironment named somedbe. During installation, ALLBASE/SQL marks each section in the module valid or invalid, depending on the current objects and authorities in somedbe. To use the INSTALL command, you need the authority to start a DBE session in the DBEnvironment that is to contain the new module. If you are replacing a module with a new one of the same name, make sure no other users are accessing the module. To avoid problems, install modules while connected to the DBEnvironment in single user mode.

Granting Required Owner Authorization

At run time, embedded SQL commands are executed only if the original module owner has the authority to execute them. Therefore, you need to grant required authorities to the module owner in the production DBEnvironment.

If module pgmr1.somemod contains a SELECT command for table PurchDB.Parts, the following grant would ensure valid owner authorization:

```
isql=> GRANT SELECT ON PurchDB.Parts TO pgmr1;
```

If pgmr1 had DBA authority, he could have assigned ownership of the module to another owner at preprocessing time by using the -o option:

```
$ psqlpas somedbe -o PurchDB -m somemod -i mysource.sql -p mysource.p -d
```

In this case, ownership belongs to a class, PurchDB. Only an individual with DBA authority can maintain this program, and runtime authorization would be established as follows:

```
isql=> GRANT SELECT ON PurchDB.Parts TO PurchDB;
```

Granting Program User Authorization

In order to execute an ALLBASE/SQL application program you must be able to start any DBE session initiated in the program. You must also have one of the following authorities in the DBEnvironment accessed by the program:

```
RUN
module OWNER
DBA
```

A DBA must grant the authority to start a DBE session. In most cases, application programs start a DBE session with the CONNECT command, so CONNECT authorization is sufficient:

```
isql=> CONNECT TO 'somedbe';
isql=> GRANT CONNECT TO somelogin;
isql=> COMMIT WORK;
```

If you have module OWNER or DBA authority, you can grant RUN authority:

```
isql=> CONNECT TO 'somedbe';
isql=> GRANT RUN ON someprog TO somelogin;
isql=> COMMIT WORK;
```

Now somelogin can run program someprog.r:

```
login: someuserid
       .
       .
       .
$ someprog.r
```

Running the Program

If the program contains the SQLEXPLAIN command, the ALLBASE/SQL message catalog must be available at run time. SQLEXPLAIN obtains warning and error messages from this catalog. The default message catalog is /usr/lib/nls/n-computer/hpsqlcat. For native language users, the catalog is /usr/lib/nls/$LANG/hpsqlcat, where $LANG is the current language. If this catalog is not available, ALLBASE/SQL issues a warning and uses the default catalog instead.

Once the ALLBASE/SQL message catalog and the appropriate DBEnvironment are identified, the program can be run as follows:

$ someprog.r

At run time, an ALLBASE/SQL program interacts with the DBEnvironment as illustrated in Figure 1-4.
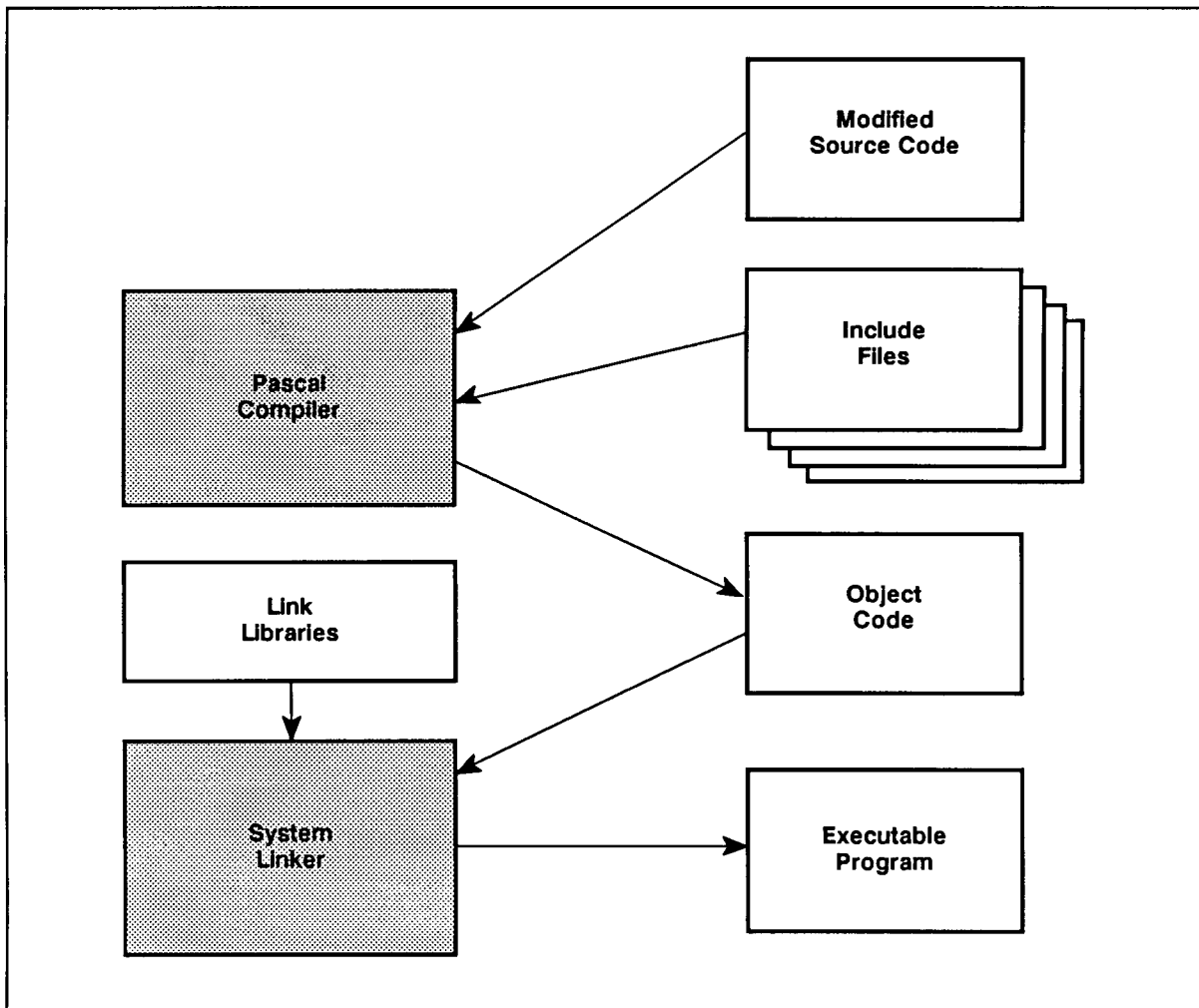
LG200128_003a

Figure 1-4. Runtime Events

All the Pascal constructs inserted by the preprocessor and the stored sections automatically handle database operations, including providing the application program with status information after each SQL command is executed. SQL commands that have a stored section are executed if the section is valid at run time or can be validated by ALLBASE/SQL at run time.

SQL commands that are not known until run time can also be processed by an application program. These SQL commands, known as **dynamic commands**, are entered by the user at run time rather than embedded in the source code at programming time. ALLBASE/SQL converts these commands into executable ALLBASE/SQL instructions at run time rather than at preprocessing time. Sections and other instructions created for dynamic data manipulation commands are deleted at the end of the transaction.

Maintaining ALLBASE/SQL Programs

After ALLBASE/SQL Pascal programs are in production use, changes in applications, personnel, or databases may necessitate doing the following tasks:

- Updating application programs.
- Changing program-related authorizations.
- Obsoleting application programs.

Updating Application Programs

Minor modifications to programs in use can often be made right on the production machine and production DBEnvironment, during hours in which the production DBEnvironment use is minimal. Major program modifications, because they are more time-consuming, are usually made on a development machine and development DBEnvironment.

In either case, the OWNER of the program's module or a DBA preprocesses the revised program and replaces the old module with a new one. Existing RUN authorities can be either preserved or revoked. Dropping old modules and preserving or revoking RUN authorities can be done either by using the DROP MODULE command in ISQL or when you invoke the preprocessor.

The PRESERVE option of the DROP MODULE command retains any existing RUN authorities for the module when it is deleted from the system catalog as in the following example:

```
isql=> DROP MODULE myprog PRESERVE;
```

To delete a module and any existing RUN authorities in ISQL, simply omit the PRESERVE option.

You can also drop a module at preprocessing time:

```
$ psqlpas somedbe -m modulename -i mysource.sql -d
```

The -d command tells the preprocessor to drop any existing module named modulename; if you omit the -d command, a new module is created only if a module named modulename does not exist. This invocation line drops any existing module named modulename, but retains any related RUN authorities. To revoke the RUN authorities, you would specify the -r (revoke) option in the command string as follows:

```
$ psqlpas somedbe -m modulename -i mysource.sql -d -r
```

The DROP MODULE command is also useful in conjunction with revised programs whose modules must be installed in a DBEnvironment different from that on which preprocessing occurred. Before using the INSTALL command to store the new module, you drop the existing module using the DROP MODULE command, preserving or dropping related RUN authorization as required.

Changing Program-Related Authorization

Once a program is in production use, you may need to grant and revoke RUN and CONNECT authority as program users revoking CONNECT authority in the following example requires DBA authorization:

```
isql=> REVOKE CONNECT FROM olduser;
```

Revoking RUN authority as shown below requires either module OWNER or DBA authority:

```
isql=> REVOKE RUN ON pgmr1.somepgm FROM olduser;
```

Obsoleting Programs

When an application program becomes obsolete, you use the DROP MODULE command to both remove the module from any DBEnvironment where it is stored and revoke any related RUN authorities:

```
isql=> DROP MODULE myprog;
```

Related RUN authorities are automatically revoked when you do not use the PRESERVE option of this command.

Use of Shared Memory

ALLBASE/SQL uses two types of shared memory, **DBCore shared memory** and **inter-process communication (IPC) shared memory**.

When you start a DBEnvironment with the START DBE command, a block of DBCORE shared memory is obtained for the current DBEnvironment session. Until the session ends, all users and programs accessing the DBEnvironment share this allocated memory, which includes:

- Transaction block buffer.
- Log buffers.
- Data buffers.
- Runtime control block buffers.

The use of DBCore shared memory is illustrated in Figure 1-5.
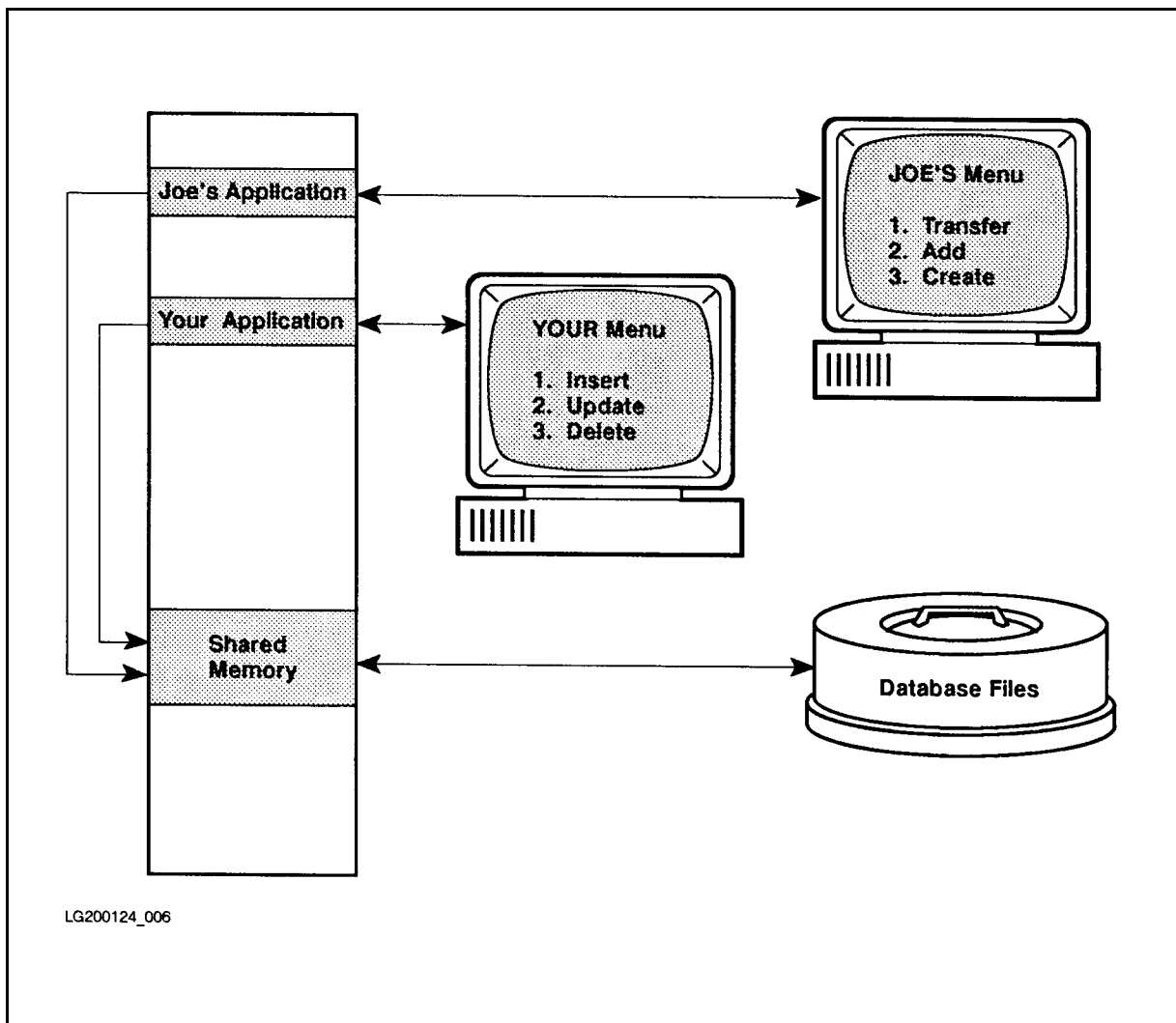
Figure 1-5. Use of DBCore Shared Memory in a DBEnvironment

Since these are shared resources, your program cannot manipulate their size directly. However, a DBA can alter them with the SQLUtil ALTDBE command. Refer to "Shared Memory Requirements" in the "Physical Design" chapter of the *ALLBASE/SQL Database Administration Guide*.

An IPC shared memory segment is allocated for each local access application accessing the DBEnvironment. IPC shared memory is application accessing the DBEnvironment. IPC shared memory is mapped to the user's process.

300/400   The following shared memory considerations apply only to Series 300 and 400 systems. Systems using SQL must be configured with a minimum shmmaxaddr of 13MB. The default shmmaxaddr is 16MB and is recommended. (See the *HP-UX System Administrator Manual* for information on shmmaxaddr, an operating system parameter.)

ALLBASE/SQL application programmers using their own shared memory must consider the following:

- Do not map your shared memory between the 12MB to 16MB address space used by ALLBASE/SQL.

- It is recommended that you avoid mapping shared memory within 1MB of the top of the data segment (brk).

- CONNECT to your DBEnvironment prior to mapping in your own shared memory. This is to satisfy ALLBASE/SQL requirements for dynamic data storage.

Choosing an Interprocess Communication Option

ALLBASE/SQL offers two interprocess communication options. Fast ipc is the default and is available for the E.1 release. The previously existing interprocess communication method, signal ipc, remains a viable alternative depending on your application requirements.

Any applications developed with ALLBASE/SQL release E.1 will by default use fast ipc. Any existing applications developed on a prior release will by default use signal ipc unless they are relinked. Once an existing application has been relinked, fast ipc is in effect. You have the option of setting an environment variable to indicate which form of interprocess communication you wish to use.

The following sections describe each option, how to set the environment variable, and debugging considerations:

- Using Fast IPC
- Using Signal IPC
- Setting the HPSQLfast_ipc Environment Variable
- Setting Signal Handling for Debugging

Using Fast IPC

Fast ipc offers the following features:

Performance is improved for applications with many, simple ALLBASE/SQL statements because overhead is reduced in communicating a request between an application and ALLBASE/SQL. It is the complexity of the statement that determines whether or not

improved performance results. The size of a transaction or the size of a query result do not affect performance in regard to fast ipc.

Only signal 18 (SIGCLD) is used by ALLBASE/SQL when fast ipc is enabled. Therefore, if your application uses signals, you should avoid using signal 18.

Note   If you want to use fast ipc and trap signals, note that once a signal has been trapped and processed, your code *must* return from the signal handling routine to the trap point before further processing continues.

Using Signal IPC

When **signal ipc** is in effect, the ALLBASE/SQL software uses the following system signals:

signal 2 (SIGINTR)
signal 14 (SIGALRM)
signal 17 (SIGUSR2)
signal 18 (SIGCLD)

If your application also uses system signals, the fast ipc option might offer a solution to signal contention problems. However, if you choose to use the signal ipc option, it remains available.

Setting the HPSQLfast_ipc Environment Variable

The following example illustrates how to set the HPSQLfast_ipc environment variable.

Suppose you already have applications that must use signals and which were written prior to this release. Signal ipc is the default unless you relink your applications. In this case, you reset the signal ipc option by setting the environment variable, HPSQLfast_ipc to n or N.

For the Bourne shell or Korn shell:

```
HPSQLfast_ipc=n; export HPSQLfast_ipc
```

For the C shell:

```
setenv HPSQLfast_ipc n
```

Perhaps your existing applications do not use signals, and you decide to take advantage of the new option. To reset to the fast ipc option you must relink your applications.

Note   The appropriate HPSQLfast_ipc environment variable must be in effect at run time.

Setting Signal Handling for Debugging

Be aware that when you enter a debugging session, you must set the signal handling table for the **xdb** debugger, for example:

z 18 rs

300/400   Series 300 and 400 systems use the **cdb** debugger with a different format, as follows:

18 z rs

Using Semaphores

Dynamically allocated and freed **semaphores** coordinate access to resources. Each user session allocates a set of two semaphores. The HP-UX ipcs command displays information on semaphores. If you need more semaphores, explicitly increase the semaphore parameters in the kernel configuration file to a larger value (or twice the number of expected concurrent user sessions) and regenerate the kernel.

For more information on semaphores, refer to the *HP-UX System Administrator Manual*.

2

Using the ALLBASE/SQL Pascal Preprocessor

You use the preprocessor to develop Pascal application programs that access an ALLBASE/SQL DBEnvironment.

The Preprocessor and Program Development

Pascal ALLBASE/SQL application programs originate as Pascal source files that are subsequently compiled with the Pascal compiler and linked by the linker to create an executable program file. The development of ALLBASE/SQL programs, however, requires that you preprocess those portions of the program that contain SQL commands *before* compilation.

In the case illustrated in Figure 2-1, the ALLBASE/SQL Pascal program consists of one source file and, optionally, one or more include files. The preprocessor merges any user include file into the source program, and preprocesses it. The result is a modified source code file and several preprocessor-generated include files. These preprocessor include files contain all of the definitions of variables used by any Pascal statements in the modified source code file. These two files are then compiled to produce an object code file, and linked to produce an executable program file, in the same manner as any other Pascal program.
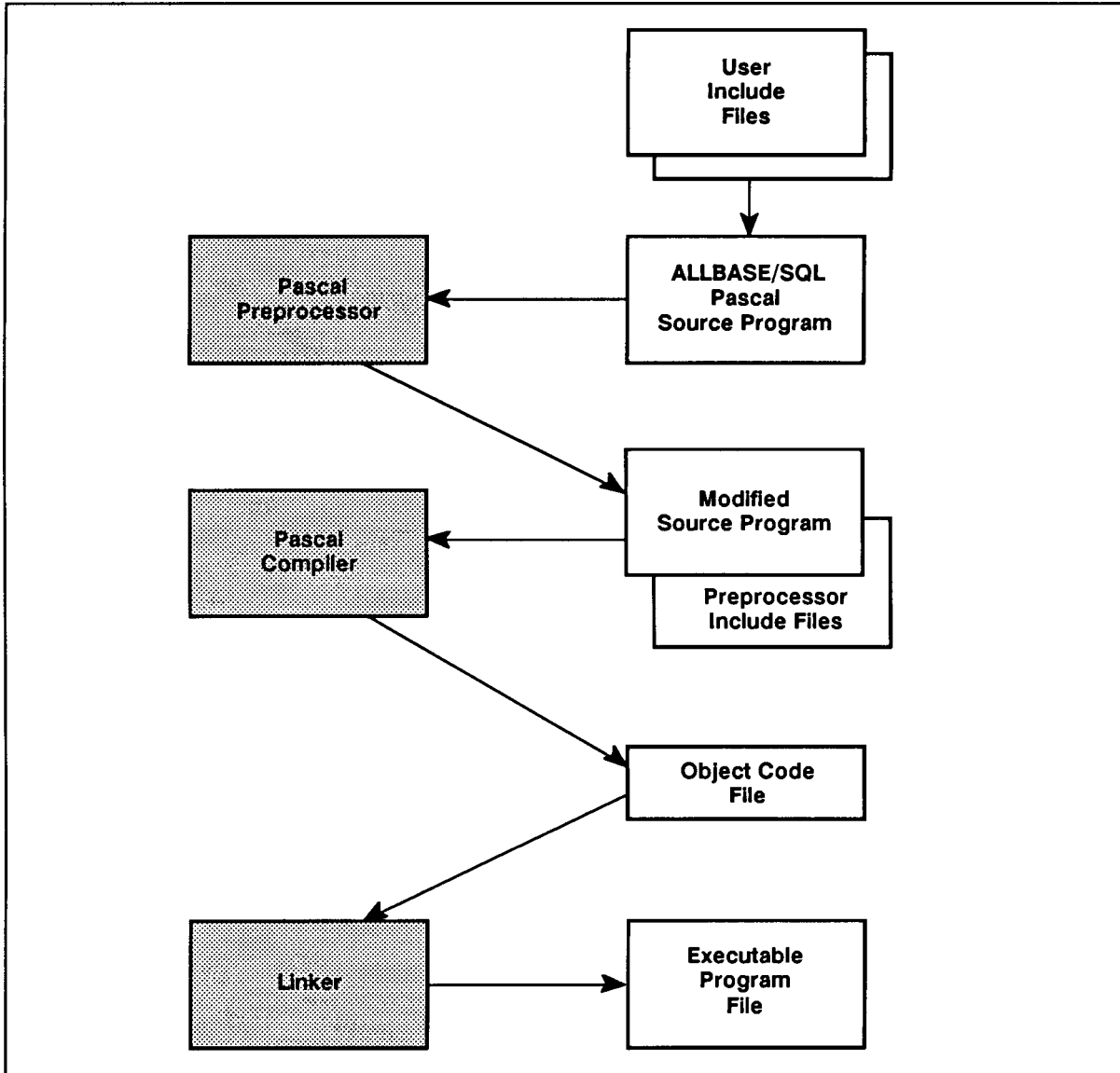
LG200128_004b

Figure 2-1. Developing a Pascal ALLBASE/SQL Program

In other cases, the ALLBASE/SQL application program might consist of a main program
and one or more subprograms in separate files. In these cases, only source files containing
SQL commands need to be preprocessed, as illustrated in Figure 2-2. You invoke the Pascal
compiler as many times as necessary to create the desired number of object code files. The
files output by the Pascal preprocessor are treated just as any other compiler input files at
compile time.

300/400   Series 300 and 400 systems do not allow subprograms. The parts of Figure 2-2 shown with diagonal lines do not apply to Series 300 and 400 systems.
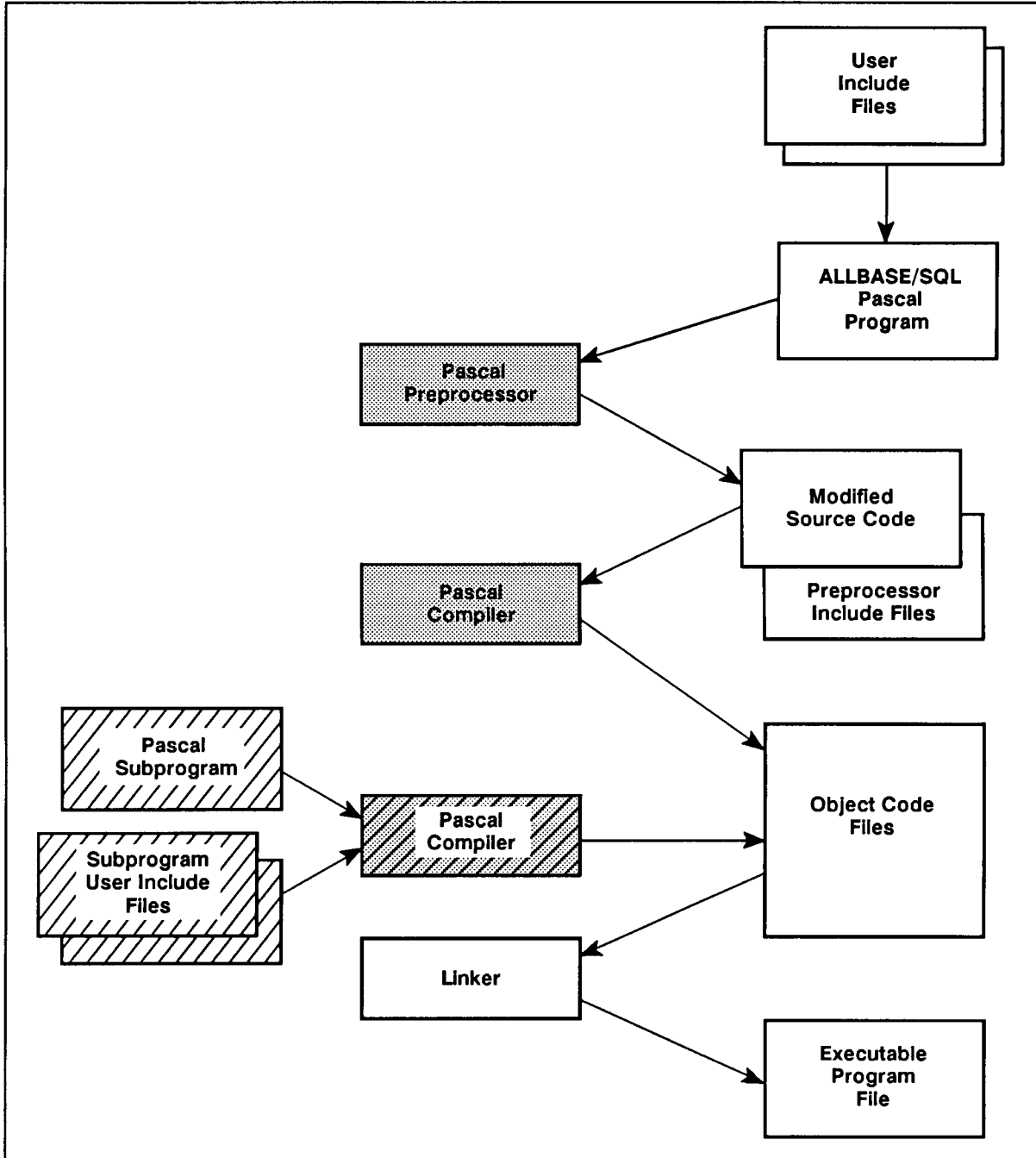
LG200128_005b

Figure 2-2. Developing a Pascal ALLBASE/SQL Program with Subprograms

During preprocessing, the Pascal preprocessor actually accesses one of the DBEnvironments to be used by your program at run time. It can be the same DBEnvironment your program uses at run time, or a DBEnvironment into which your program is installed. The preprocessor stores a module in the DBEnvironment which is executed at run time. The module is used at run time to optimize and validate DBEnvironment operations.

During any invocation, the Pascal preprocessor can access only *one* DBEnvironment. You can create separate subprograms that all access the *same* DBEnvironment. Each program is separately preprocessed and compiled. In this case, the preprocessor stores *multiple* modules in *one* DBEnvironment.

The criteria governing the division of an application program into subprograms is very application-dependent. As in the development of any application program, factors such as program size, program complexity, expected recompilation frequency, and number of programmers affect how a program is subdivided. In the case of ALLBASE/SQL Pascal application programs, the only additional factors are as follows:

- All code containing embedded SQL commands must be preprocessed.

- The preprocessor can access only one DBEnvironment at a time.

- Each program or subprogram (preprocessed unit) that accesses the same DBEnvironment must have a unique OwnerName.ModuleName.

- The preprocessor can process only one program or subprogram per invocation.

Preprocessor Modes

You can use the preprocessor in the following two modes:

1. To only check your SQL syntax.

2. To perform full preprocessing, which includes SQL syntax checking, creating compilable output, storing a module in a DBEnvironment, and creating a file that contains an installable copy of the stored module.

As you develop the SQL portions of your Pascal programs, syntax checking mode is quite useful. Preprocessing is quicker in this mode than in full preprocessing mode. In addition, you can start debugging your SQL commands before the DBEnvironment itself is in place.

How to run the preprocessor in both modes is described later in this chapter under "Invoking the Pascal Preprocessor."

Preprocessor Input and Output

Regardless of the mode you use, the following files must be available when you invoke the Pascal preprocessor, as shown in Figure 2-3:

- **Source file**: A file containing the source code of the Pascal ALLBASE/SQL program with embedded SQL commands for the DBEnvironments used. The name is specified by using the -i option as explained later in this chapter.

- **ALLBASE/SQL Message Catalog**: The ALLBASE/SQL message catalog, which contains preprocessor messages and ALLBASE/SQL error and warning messages. The message catalog can actually be accessed through two different pathnames which each point to the same file. The fully qualified names that point to the default catalog are:

    `/usr/lib/hpsqlcat`

    `or`

    `/usr/lib/nls/n-computer/hpsqlcat`

    For native language users, the name of the catalog is:

    `/usr/lib/nls/$LANG/hpsqlcat`

    where $LANG is the name of the current language. If this catalog is not available, ALLBASE/SQL issues a warning and uses the default catalog instead.

When you run the preprocessor in full preprocessing mode, also ensure that the DBEnvironment accessed by the program is available.

- **ALLBASE/SQL Message File**: File containing the preprocessor banner, error and warning messages, and other messages. The file name for this file is as follows:

    `sqlmsg`

- **Installable module file**: File containing a copy of the module created by the preprocessor. The file name syntax for this file is as follows:

    ModifiedSourceFileName`.sqlm`

    When you run the preprocessor in full preprocessing mode, the preprocessor also stores a module in the DBEnvironment accessed by your program. The module is used at run time to execute DBEnvironment operations.

    As Figure 2-4 points out, the Pascal preprocessor creates the following output files:

- **Modified source file**: File containing the modified version of the source code in the source file. The default filename syntax for this file is as follows:

    ModifiedSourceFileName`.p`

    An alternative name can be specified by using the -p option as described later in this chapter.

- **Include files**: After you use the preprocessor in full preprocessing mode, you use the modified source code file and the following four include files as input files for the Pascal compiler, as shown in Figure 2-4. These include files contain declarations and definitions used by Pascal constructs the preprocessor inserts into the modified source code file. The syntax is as follows:

☐ **Constant include file:** File containing constant definitions, is as follows:

ModifiedSourceFileName.`sqlc`

☐ **Type include file:** File containing type declarations, is as follows:

ModifiedSourceFileName.`sqlt`

☐ **Variable include file:** File containing variable declarations, is as follows:

ModifiedSourceFileName.`sqlv` (or *ModifiedSourceFileName*.`sv`*nn*)

☐ **External include file:** File containing external procedure declarations, is as follows:

ModifiedSourceFileName.`sqle`

Any extension on the ModifiedSourceFileName is ignored when the preprocessor assigns names to the include files.

If SourceFileName is in a language other than n-computer (ASCII), the ModifiedSourceFileName and all generated files will have names in the native language and extensions in ASCII.
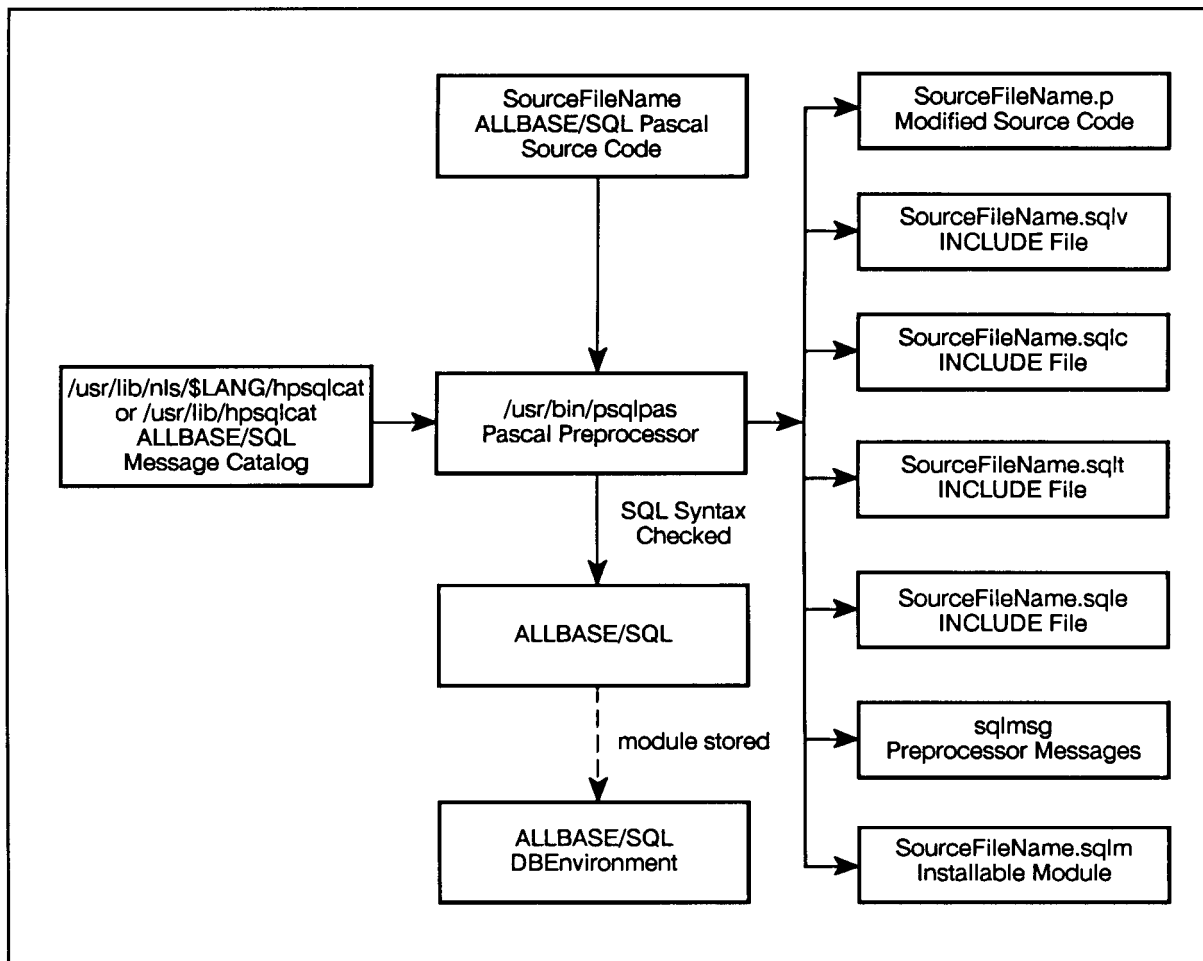
LG200128_010

Figure 2-3. Pascal Preprocessor Input and Output
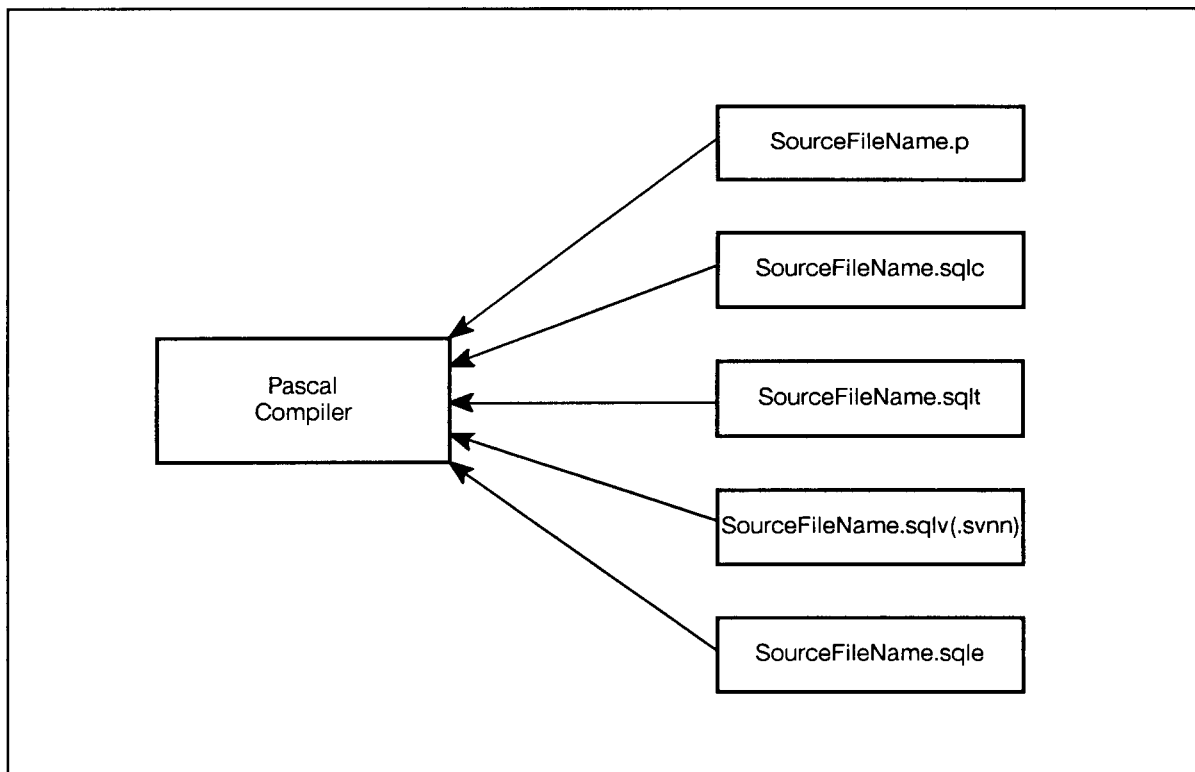
LG200128_011

Figure 2-4. Compiling Pascal Preprocessor Output

Source File

The source file must be a file that contains at a minimum the following constructs:

```
(* PROGRAM HEADING *)
Program  ProgramName(input, output);

begin
AnyStatement;
end.
```

When parsing the source file, the Pascal preprocessor ignores Pascal statements and most Pascal compiler directives in it. Only the following information is parsed by the Pascal preprocessor:

■ The Pascal compiler directives $Skip_Text ON$, $Skip_Text OFF$, $Set, $If, $Else, $Endif, and $Include.

300/400   Series 300 and 400 systems comment out SQL commands, rather than using a $Skip_text compiler directive.

■ The program name. Unless you specify a module name in the preprocessor invocation line, the preprocessor uses the program name as the name for the module it stores. The name may optionally have the suffix .sql to distinguish it from non-SQL programs. A module

name can contain as many as 20 bytes and must follow the rules governing ALLBASE/SQL basic names (given in the *ALLBASE/SQL Reference Manual*).

■ Constructs found between the prefix EXEC SQL and the suffix ;. These constructs follow the rules given in Chapter 3 for how and where to embed these constructs.

■ Constructs found between the BEGIN DECLARE SECTION and END DECLARE SECTION commands. These commands delimit a **declare section**, which contains Pascal data declarations for the host variables used in the program. Host variables are described in the chapter, "Host Variables."

Figure 2-6 illustrates a source file containing a sample program using the following SQL commands, highlighted in shading in Figure 2-7:

```
INCLUDE SQLCA
BEGIN DECLARE SECTION
END DECLARE SECTION
WHENEVER
CONNECT
BEGIN WORK
SELECT
COMMIT WORK
SQLEXPLAIN
```

As the sample dialog in Figure 2-5 illustrates, the program begins a DBE session for PartsDBE, the sample DBEnvironment. It prompts the user for a part number, then displays information about the part from the table PurchDB.Parts. Warning and error conditions are handled with WHENEVER and SQLEXPLAIN commands with the exception of explicit error checking after the SELECT command. The program continues to prompt for a part number until a serious error is encountered or until the user enters a slash (/).

```
      Program to SELECT specified rows from the Parts Table - PASEX2

   Event List:
    Connect to PartsDBE
    Begin Work
    SELECT specified row from Parts Table
      until user enters "/"
    Commit Work
    Disconnect from PartsDBE

   Connect to PartsDBE

   Enter Part Number within Parts Table or "/" to STOP>
   1243-P-01

   Begin Work
   SELECT PartNumber, PartName, SalesPrice

   Row not found!

   Commit Work

   Enter Part Number within Parts Table or "/" to STOP>
   1323-D-01

   Begin Work
   SELECT PartNumber, PartName, SalesPrice

   Part Number:  1323-D-01
   Part Name:    Floppy Diskette Drive
   Sales Price:        200.00

   Commit Work

   Enter Part Number within Parts Table or "/" to STOP> /

   Release PartsDBE

   Terminating Program
```

Figure 2-5. Runtime Dialog of Program pasex2

```
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* This program illustrates the use of SQL's SELECT command to      *)
(* retrieve one row or tuple at a time.                             *)
(* BEGIN WORK is executed before the SELECT and a COMMIT WORK       *)
(* after the SELECT.  An indicator variable is also used for        *)
(* SalesPrice.                                                      *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

Program pasex2(input, output);

label
    1000,
    9999;

const
    OK            =        0;
    NotFound      =      100;
    MultipleRows  =  -10002;
    DeadLock      =  -14024;

var

    EXEC SQL INCLUDE SQLCA;

  (* Begin Host Variable Declarations *)
    EXEC SQL  Begin Declare Section;
    PartNumber      : packed array[1..16] of char;
    PartName        : packed array[1..30] of char;
    SalesPrice      : longreal;
    SalesPriceInd   : SQLIND;
    SQLMessage      : packed array[1..132] of char;
    EXEC SQL  End Declare Section;
  (* End Host Variable Declarations *)

    Abort            : boolean;

procedure SQLStatusCheck;  (* Procedure to Display Error Messages *)
    Forward;

$PAGE $

(* Directive to set SQL Whenever error checking *)
EXEC SQL Whenever SQLError GOTO 1000;
```

Figure 2-6. Program pasex2: Using Simple Select

```
Procedure ConnectDBE;  (* Procedure to Connect to PartsDBE *)
begin

writeln('Connect to PartsDBE');
EXEC SQL CONNECT TO '../sampledb/PartsDBE';

end;  (* End of ConnectDBE Procedure *)


Procedure BeginTransaction;  (* Procedure to Begin Work *)
begin

writeln;
writeln('Begin Work');
EXEC SQL BEGIN WORK;

end;  (* End BeginTransaction Procedure *)


procedure EndTransaction;  (* Procedure to Commit Work *)
begin

writeln;
writeln('Commit Work');
EXEC SQL COMMIT WORK;

end;  (* End EndTransaction Procedure *)


(* Directive to reset SQL Whenever error checking *)
EXEC SQL WHENEVER SQLERROR CONTINUE;


procedure TerminateProgram;   (* Procedure to Release PartsDBE *)
begin

writeln('Release PartsDBE');
EXEC SQL COMMIT WORK RELEASE;

writeln;
writeln('Terminating Program');
Goto 9999;  (* Goto exit point of main program *)

end;  (* End TerminateProgram Procedure *)
$PAGE $
```

Figure 2-6. Program pasex2: Using Simple Select (page 2 of 5)

```
procedure DisplayRow;   (* Procedure to Display Parts Table Rows *)
begin

writeln;
writeln('Part Number: ', PartNumber);
writeln('Part Name:   ', PartName);
if SalesPriceInd < 0 then
   writeln('Sales Price is NULL')
else
   writeln('Sales Price: ', SalesPrice:10:2);

end;  (* End of DisplayRow *)
$PAGE $

procedure SelectData; (* Procedure to Query Parts Table *)
begin

repeat

writeln;
prompt('Enter Part Number within Parts Table or "/" to STOP> ');
readln(PartNumber);
writeln;

if PartNumber[1] <> '/' then
  begin

  BeginTransaction;

  writeln('SELECT PartNumber, PartName, SalesPrice');
  EXEC SQL SELECT PartNumber, PartName, SalesPrice
     INTO :PartNumber,
    :PartName,
    :SalesPrice  :SalesPriceInd
       FROM PurchDB.Parts
      WHERE PartNumber = :PartNumber;

  if SQLCA.SQLWARN[0] in ['W','w'] then
     begin
     write('SQL WARNING has occurred. The following row');
     writeln('of data may not be valid.');
     end;
```

Figure 2-6. Program pasex2: Using Simple Select (page 3 of 5)

```
   case SQLCA.SQLCODE of
   OK            : DisplayRow;
   NotFound      : begin
      writeln;
      writeln('Row not found!');
    end;
   MultipleRows : begin
      writeln;
      writeln('WARNING: More than one row qualifies.');
    end;
   otherwise      begin
      SQLStatusCheck;
      end;

   end;  (* case *)

   EndTransaction;

end;  (* End if *)
until PartNumber[1] = '/';

end;      (* End of SelectData Procedure *)

procedure SQLStatusCheck;  (* Procedure to Display Error Messages *)
begin

Abort := FALSE;
if SQLCA.SQLCODE < DeadLock then Abort := TRUE;

repeat
EXEC SQL SQLEXPLAIN :SQLMessage;
writeln(SQLMessage);
until SQLCA.SQLCODE = 0;

if Abort then
  begin

  TerminateProgram;

  end;

end;  (* End SQLStatusCheck Procedure *)
$PAGE $
```

Figure 2-6. Program pasex2: Using Simple Select (page 4 of 5)

```
begin  (* Beginning of Program *)

write('Program to SELECT specified rows from ');
writeln('the Parts Table - pasex2');
writeln;
writeln('Event List:');
writeln('  Connect to PartsDBE');
writeln('  Begin Work');
writeln('  SELECT specified row from Parts Table');
writeln('   until user enters "/" ');
writeln('  Commit Work');
writeln('  Disconnect from PartsDBE');
writeln;

ConnectDBE;
SelectData;
TerminateProgram;

(* Whenever Routine - Serious DBE Error *)
(* SQL Whenever SQLError Entry Point *)
1000:

  (* Begin *)
  SQLStatusCheck;
  TerminateProgram;
  (* End *)

(* Exit Point for the main program *)
9999:

end.   (* End of Program *)
```

Figure 2-6. Program pasex2: Using Simple Select (page 5 of 5)

Output File Attributes

When the source file illustrated in Figure 2-6 is preprocessed, the attributes of the output files created are as follows:

```
$ ll pasex2*

-r--r--r--   1 thomas      dbsupport   4714 Sep 16 11:25 pasex2
-rwxrwxr-x   1 thomas      dbsupport   6450 Sep 18 11:02 pasex2.p
-rwxrwxr-x   1 thomas      dbsupport    149 Sep 18 11:02 pasex2.sqlc
-rwxrwxr-x   1 thomas      dbsupport   3504 Sep 18 11:02 pasex2.sqle
-rwxrwxr-x   1 thomas      dbsupport   1500 Sep 18 11:02 pasex2.sqlm
-rwxrwxr-x   1 thomas      dbsupport   1536 Sep 18 11:02 pasex2.sqlt
-rwxrwxr-x   1 thomas      dbsupport    156 Sep 18 11:02 pasex2.sqlv

$ ll sqlmsg

-rwxrwxr-x   1 thomas      dbsupport    403 Sep 18 11:02 sqlmsg
```

Modified Source File

As the Pascal preprocessor parses the source file, it copies lines from it and any include file(s) into the modified source file, comments out embedded SQL commands, and inserts information around each embedded SQL command. Figures 2-7 and 2-8 illustrate the modified source file generated for the source file pictured in Figure 2-6. The shaded lines highlight areas of original SQL commands commented out and boundaries of begin blocks of preprocessor-generated code.

In both preprocessing modes (which are discussed later in this chapter), the Pascal preprocessor inserts a $Skip_Text ON$ and $Skip_Text OFF$ compiler directive around the embedded SQL command to comment out the SQL command.

300/400   In Series 300 and 400 systems, it comments out all SQL commands and delimits any comments within SQL commands with the braces characters ({ and }).

■ Inserts four include Pascal compiler directives within the declaration part. These directives reference the four preprocessor-generated include files. The constant and type include files are included after the program header. The variable and externals include files are included at the end of the global declaration part.

■ Places any comment you placed after an embedded command on the line following the last line generated for the embedded command. Note, for example, that the comment following the INCLUDE SQLCA command in the source file is in the same column, but on a different line, in the modified source file.

300/400   To ensure portability between Series 300 or 400 and 700 or 800 systems, the
               Series 300 and 400 Pascal preprocessors additionally insert the following three
               compiler directives at the beginning of the program:

```
$SYSPROG ON$
$LONGSTRINGS ON$
$UNDERSCORE ON$
```

If you are preprocessing code written for a previous version of Series 300 SQL,
the following information may need to be considered.

The $UNDERSCORE ON$ directive was not available in previous compiler
versions. Thus, $ALIAS$ references to the symbol table which were coded
with an underscore as the first character are preceded by two underscores after
preprocessing, causing an unresolved symbol table error at link time. To avoid
this problem, remove the original underscore in the $ALIAS$ directive.

Use of the $LONGSTRINGS ON$ directive requires that all linked code use
the same directive. If any of your programs that are to be linked together
use the default string size, the program should be changed to include the
$LONGSTRINGS ON$ directive.

In full preprocessing mode, the preprocessor also generates the following:

■ Pascal declaration of the SQLCA and the SQLDA in the type include file.

■ Pascal statements providing conditional instructions following SQL commands encountered
after one of the following SQL commands: WHENEVER SQLERROR, WHENEVER
SQLWARNING, and WHENEVER NOT FOUND.

■ Pascal statements that call ALLBASE/SQL external procedures at runtime. These calls
reference the module stored by the preprocessor in the DBEnvironment for execution at run
time. Parameters used by these external calls are defined in the variable, constant, type,
and external include files.

Caution   Although you can access the preprocessor output files with an editor, you
               should *never* change the information generated by the Pascal preprocessor.
               Your DBEnvironment could be damaged at run time if preprocessor-generated
               constructs are altered.

If you need to change nonpreprocessor-generated constructs in the modified source file, make
the changes to the source file, preprocess it again, and re-compile the output files before
putting the application program into production.

```
$set 'XOPEN_SQLCA=false'$
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* This program illustrates the use of SQL's SELECT command to    *)
(* retrieve one row or tuple at a time.                           *)
(* BEGIN WORK is executed before the                             *)
(* SELECT and a COMMIT WORK after the SELECT.  An indicator        *)
(* variable is also used for SalesPrice.                          *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

Program pasex2(input, output);

label
    1000,
    9999;

$include 'pasex2.sqlc'$
$include 'pasex2.sqlt'$
const
    OK           =       0;
    NotFound     =     100;
    MultipleRows = -10002;
    DeadLock     = -14024;

var

$SKIP_TEXT ON$
    EXEC SQL INCLUDE SQLCA;
$SKIP_TEXT OFF$
SQLCA : SQLCA_TYPE;

        (* Begin Host Variable Declarations *)

$SKIP_TEXT ON$
    EXEC SQL  Begin Declare Section;
$SKIP_TEXT OFF$

    PartNumber       : packed array[1..16] of char;
    PartName         : packed array[1..30] of char;
    SalesPrice       : longreal;
    SalesPriceInd    : SQLIND;
    SQLMessage       : packed array[1..132] of char;


$SKIP_TEXT ON$
    EXEC SQL  End Declare Section;
$SKIP_TEXT OFF$

        (* End Host Variable Declarations *)
```

Figure 2-7. Sample Modified Source File

```
      Abort               : boolean;

$include 'pasex2.sqlv'$
$include 'pasex2.sqle'$
procedure SQLStatusCheck;  (* Procedure to Display Error Messages *)
     Forward;

$PAGE $

(* Directive to set SQL Whenever error checking *)


$SKIP_TEXT ON$
EXEC SQL Whenever SqlError goto 1000;
$SKIP_TEXT OFF$

Procedure ConnectDBE;  (* Procedure to Connect to PartsDBE *)
begin

writeln('Connect to PartsDBE');

$SKIP_TEXT ON$
EXEC SQL CONNECT TO '../sampledb/PartsDBE';
$SKIP_TEXT OFF$
begin
SQLVAR1 := '00AE00002E2E2F73616D706C6564622F50617274473444245202020202020' +
           '2020202020202020202020202020202020202020202020202020202020202020' +
           '2020202020202020202020202020202020202020202020202020202020202020' +
           '2020202020202020202020202020202020202020202020202020202020202020' +
           '20202020202020202020202020';
SQLXCON(waddress(SQLCA), SQLVAR1);
if SQLCA.SQLCODE < 0 then
  goto 1000;
end;

end;  (* End of ConnectDBE Procedure *)

Procedure BeginTransaction;  (* Procedure to Begin Work *)
begin

writeln;
writeln('Begin Work');

$SKIP_TEXT ON$
EXEC SQL BEGIN WORK;
$SKIP_TEXT OFF$
```

Figure 2-7. Sample Modified Source File (page 2 of 6)

```
begin
SQLVAR2 := '00A6007F00110061';
SQLXCON(waddress(SQLCA), SQLVAR2);
if SQLCA.SQLCODE < 0 then
  goto 1000;
end;
end;  (* End BeginTransaction Procedure *)

procedure EndTransaction;  (* Procedure to Commit Work *)
begin
writeln;
writeln('Commit Work');

$SKIP_TEXT ON$
EXEC SQL COMMIT WORK;
$SKIP_TEXT OFF$
begin
SQLVAR3 := '00A10000';
SQLXCON(waddress(SQLCA), SQLVAR3);
if SQLCA.SQLCODE < 0 then
  goto 1000;
end;

end;  (* End EndTransaction Procedure *)

(* Directive to reset SQL Whenever error checking *)

$SKIP_TEXT ON$
EXEC SQL WHENEVER SQLERROR CONTINUE;
$SKIP_TEXT OFF$

procedure TerminateProgram;   (* Procedure to Release PartsDBE *)
begin
writeln('Release PartsDBE');

$SKIP_TEXT ON$
EXEC SQL COMMIT WORK RELEASE;
$SKIP_TEXT OFF$
begin
begin
SQLVAR4 := '00A10000';
SQLXCON(waddress(SQLCA), SQLVAR4);
end;
begin
SQLVAR5 := '00B200002020202020202020202020202020202020202020FFFFFFFF';
SQLXCON(waddress(SQLCA), SQLVAR5);
```

Figure 2-7. Sample Modified Source File (page 3 of 6)

```
   end;
  end;

  writeln;
  writeln('Terminating Program');
  Goto 9999;  (* Goto exit point of main program *)

  end;  (* End TerminateProgram Procedure *)
  $PAGE $

procedure DisplayRow;    (* Procedure to Display Parts Table Rows *)
begin

  writeln;
  writeln('Part Number: ', PartNumber);
  writeln('Part Name:   ', PartName);
  if SalesPriceInd < 0 then
     writeln('Sales Price is NULL')
  else
     writeln('Sales Price: ', SalesPrice:10:2);

  end;  (* End of DisplayRow *)
  $PAGE $

procedure SelectData; (* Procedure to Query Parts Table *)
begin

  repeat

  writeln;
  prompt('Enter Part Number within Parts Table or "/" to STOP> ');
  readln(PartNumber);
  writeln;

  if PartNumber[1]  '/' then
    begin
    BeginTransaction;

    writeln('SELECT PartNumber, PartName, SalesPrice');

  $SKIP_TEXT ON$
    EXEC SQL SELECT PartNumber, PartName, SalesPrice
             INTO :PartNumber,
                  :PartName,
                  :SalesPrice  :SalesPriceInd
             FROM PurchDB.Parts
             WHERE PartNumber = :PartNumber;
  $SKIP_TEXT OFF$
```

Figure 2-7. Sample Modified Source File (page 4 of 6)

```
begin
SQLTEMPV.REC1.PartNumber1 := PartNumber;
SQLXFET(waddress(SQLCA),SQLOWNER,SQLMODNAME,1,waddress(SQLTEMPV),
        16,64,TRUE);
if SQLCA.SQLCODE = 0 then
  begin
  PartNumber := SQLTEMPV.REC2.PartNumber1;
  PartName := SQLTEMPV.REC2.PartName2;
  if SQLTEMPV.REC2.SalesPriceInd4 >= 0 then
  SalesPrice := SQLTEMPV.REC2.SalesPrice3;
  SalesPriceInd := SQLTEMPV.REC2.SalesPriceInd4;
  end
else
  begin
  end;
end;

  if SQLCA.SQLWARN[0] in ['W','w'] then
     begin
     write('SQL WARNING has occured. The following row');
     writeln('of data may not be valid.');
     end;

  case SQLCA.SQLCODE of
  OK            : DisplayRow;
  NotFound      : begin
                    writeln;
                    writeln('Row not found!');
                  end;
  MultipleRows : begin
                    writeln;
                    writeln('WARNING: More than one row qualifies.');
                  end;
  otherwise       begin
                    SQLStatusCheck;
                    end;
  end;  (* case *)
  EndTransaction;

end;  (* End if *)
until PartNumber[1] = '/';
end;        (* End of SelectData Procedure *)

procedure SQLStatusCheck;  (* Procedure to Display Error Messages *)
begin

Abort := FALSE;
if SQLCA.SQLCODE < DeadLock then Abort := TRUE;
```

Figure 2-7. Sample Modified Source File (page 5 of 6)

```
  repeat

  $SKIP_TEXT ON$
  EXEC SQL SQLEXPLAIN :SQLMessage;
  $SKIP_TEXT OFF$
  begin
  SQLXPLN(waddress(SQLCA),waddress(SQLTEMPV.REC4),132,0);
  SQLMessage := '';
  strmove(132,SQLTEMPV.REC4,1,SQLMessage, 1);
  end;

  writeln(SQLMessage);
  until SQLCA.SQLCODE = 0;

  if Abort then
    begin
    TerminateProgram;
    end;
  end;  (* End SQLStatusCheck Procedure *)
  $PAGE $

  begin  (* Beginning of Program *)

  write('Program to SELECT specified rows from ');
  writeln('the Parts Table - PASEX2');
  writeln;
  writeln('Event List:');
  writeln('  Connect to PartsDBE');
  writeln('  Begin Work');
  writeln('  SELECT specified row from Parts Table');
  writeln('   until user enters "/" ');
  writeln('  Commit Work');
  writeln('  Disconnect from PartsDBE');
  writeln;
  ConnectDBE;
  SelectData;
  TerminateProgram;

  (* Whenever Routine - Serious DBE Error *)
  (* SQL Whenever SQLError Entry Point *)
  1000:
    (* Begin *)
    SQLStatusCheck;
    TerminateProgram;
    (* End *)
  (* Exit Point for the main program *)
  9999:
  end.   (* End of Program *)
```

Figure 2-7. Sample Modified Source File (page 6 of 6)

Preprocessor-Generated Include Files

The preprocessor creates a set of include files (with extensions .sqlc, .sqlt, .sqlv, .sv*nn*, and .sqle) for the constant, type, variable, and external declarations in your application. In .sv*nn*, *nn* are two numerical digits. Figure 2-8 through Figure 2-11 illustrate the generated files that correspond to the modified source files in Figure 2-7. The constant file is specific to the user and application.

The preprocessor inserts four Pascal compiler directives in the following syntax to reference the include files.

```
$include 'ModifiedSourceFileName.sqlc'$
$include 'ModifiedSourceFileName.sqlt'$
$include 'ModifiedSourceFileName.sqlv'$
$include 'ModifiedSourceFileName.sqle'$
```

These four directives are always inserted into the global declaration section of a main program. For each declare section in a subprogram, an include file of *ModifiedSourceFileName*.sv*nn* syntax is generated, and the compiler directive $include *ModifiedSourceFileName*.sv*nn*$ is inserted in the local declaration section. The value of *nn* is from 01 through 99.

```
  const
    SQLOWNER   = 'THOMAS@DARBY          ';
    SQLMODNAME = 'PASEX2                ';
```

Figure 2-8. Sample Constant Include File

```
  type
    ownername_type  = string[20];
    modulename_type = string[20];
    smallint = shortint;
    SQLIND = shortint;
    SQLREC1 = record
      PartNumber1 : packed array [1..16] of char;
      end;
    SQLREC2 = record
      PartNumber1 : packed array [1..16] of char;
      PartName2 : packed array [1..30] of char;
      SalesPrice3 : longreal;
      SalesPriceInd4 : sqlind;
      end;
    SQLREC3 = record
      DUMMY1, DUMMY2              : SQLREC2
      end;
    SQLREC4 = packed array[1..132] of char;
    SQLCASES = 0..4;
    SQLCA_TYPE = record
      SQLCAID : packed array [1..8] of char;
      SQLCABC : integer;
      SQLCODE : integer;
      SQLERRM : string[255];
      SQLERRP : packed array [1..8] of char;
      SQLERRD : array [1..6] of integer;
$if 'XOPEN_SQLCA'$
      SQLWARN0, SQLWARN1, SQLWARN2,
      SQLWARN3, SQLWARN4, SQLWARN5,
      SQLWARN6, SQLWARN7 : char;
$else$
      SQLWARN : packed array [0..7] of char;
$endif$
      SQLEXT  : packed array [1..8] of char;
      end;
    SQLFORMAT_TYPE = packed record
      SQLNTY, SQLTYPE, SQLPREC, SQLSCALE : smallint;
      SQLTOTALLEN, SQLVALLEN, SQLINDLEN  : integer;
      SQLVOF, SQLNOF : integer;
      SQLNAME : packed array [1..20] of char;
      end;
    SQLDA_TYPE = record
      SQLDAID   : packed array [1..8] of char;
      SQLDABC   : integer;
      SQLN      : integer;
      SQLD      : integer;
```

Figure 2-9. Sample Type Include File

```
      SQLFMTARR : integer;
      SQLNROW   : integer;
      SQLRROW   : integer;
      SQLROWLEN : integer;
      SQLBUFLEN : integer;
      SQLROWBUF : integer;
      end;
   SQLTEMPV_TYPE_P = @SQLTEMPV_TYPE;
   SQLTEMPV_TYPE = record case SQLCASES of
           0 : (dummy : integer);
           1: (REC1 : SQLREC1);
           2: (REC2 : SQLREC2);
           3: (REC3 : SQLREC3);
           4: (REC4 : SQLREC4);
      end;
```

Figure 2-9. Sample Type Include File (page 2 of 2)

```
 var
   SQLVAR1  : string[264];
   SQLVAR2  : string[16];
   SQLVAR3  : string[8];
   SQLVAR4  : string[8];
   SQLVAR5  : string[56];
   SQLTEMPV : SQLTEMPV_TYPE;
```

Figure 2-10. Sample Variable Include File

```
procedure SQLXBFE
    (SQLCAP         : integer;
     owner          : ownername_type;
     xmodule        : modulename_type;
     section        : integer;
     parms          : integer;
     inparms        : integer;
     outarray       : integer;
     entrysize      : integer;
     nentry         : integer;
     firstrow       : integer;
     nrow           : integer); external;
procedure SQLXBIN
    (SQLCAP         : integer;
     owner          : ownername_type;
     xmodule        : modulename_type;
     section        : integer;
     inarray        : integer;
     entrysize      : integer;
     nentry         : integer;
     firstrow       : integer;
     nrow           : integer); external;
procedure SQLXCNH
    (SQLCAP         : integer;
     msgstrp        : integer;
     xstrlen        : integer;
     isvarchar      : integer); external;
procedure SQLXCON
    (SQLCAP         : integer;
     var stmt       : string);  external;
procedure SQLXDDU
    (SQLCAP         : integer;
     xmodule        : modulename_type;
     section        : integer;
     parms          : integer;
     inparms        : integer;
     var stmt       : string);  external;
procedure SQLXDEX
    (SQLCAP         : integer;
     SQLDAP         : integer;
     xmodule        : modulename_type;
     section        : integer); external;
procedure SQLXDFE
    (SQLCAP         : integer;
     SQLDAP         : integer;
     xmodule        : modulename_type;
     section        : integer); external;
```

Figure 2-11. Sample Externals Include Files

```
procedure SQLXDOPK
    (SQLCAP       : integer;
     SQLDAP       : integer;
     owner        : ownername_type;
     xmodule      : modulename_type;
     section      : integer;
     parms        : integer;
     inparms      : integer;
     kpcval       : integer); external;
procedure SQLXDSB
    (SQLCAP       : integer;
     SQLDAP       : integer;
     xmodule      : modulename_type;
     section      : integer;
     ifinput      : integer); external;
procedure SQLXEXI
    (SQLCAP       : integer;
     queryptr     : integer;
     querysize    : integer); external;
procedure SQLXEXU
    (SQLCAP       : integer;
     xmodule      : modulename_type;
     section      : integer;
     parms        : integer;
     inparms      : integer;
     var formats  : string;
     nhv          : integer;
     nentry       : integer;
     firstrow     : integer;
     nrow         : integer); external;
procedure SQLXFET
    (SQLCAP       : integer;
     owner        : ownername_type;
     xmodule      : modulename_type;
     section      : integer;
     parms        : integer;
     inparms      : integer;
     outparms     : integer;
     isselect     : boolean); external;
procedure SQLXIDU
    (SQLCAP       : integer;
     owner        : ownername_type;
     xmodule      : modulename_type;
     section      : integer;
     parms        : integer;
     inparms      : integer;
     isbulk       : boolean); external;
```

Figure 2-11. Sample Externals Include Files (page 2 of 3)

```
procedure SQLXOPK
    (SQLCAP         : integer;
     owner          : ownername_type;
     xmodule        : modulename_type;
     section        : integer;
     parms          : integer;
     inparms        : integer;
     kpcval         : integer); external;
procedure SQLXOPU
    (SQLCAP         : integer;
     owner          : ownername_type;
     xmodule        : modulename_type;
     section        : integer;
     parms          : integer;
     inparms        : integer;
     var formats    : string;
     nhv            : integer;
     kpcval         : integer); external;
procedure SQLXPLN
    (SQLCAP         : integer;
     msgstrp        : integer;
     xstrlen        : integer;
     isvarchar      : integer); external;
procedure SQLXPRE
    (SQLCAP         : integer;
     queryptr       : integer;
     querysize      : integer;
     xmodule        : modulename_type;
     section        : integer); external;
procedure SQLXSECT
    (SQLCAP         : integer;
     owner          : ownername_type;
     modul          : modulename_type;
     section        : integer); external;
procedure SQLXSTP
    (SQLCAP         : integer); external;
procedure SQLXSVPT
    (SQLCAP         : integer;
     xstrlen        : integer;
     var hexstr     : string;
     svptrec        : integer); external;
```

Figure 2-11. Sample Externals Include Files (page 3 of 3)

ALLBASE/SQL Message File

Messages placed in the ALLBASE/SQL message file (sqlmsg) come from the ALLBASE/SQL message catalog. The default catalog is /usr/lib/nls/n-computer/hpsqlcat. For native language users, the name of the catalog is /usr/lib/nls/$LANG/hpsqlcat, where $LANG is the current language. If this catalog is not available, ALLBASE/SQL uses the default instead.

Sqlmsg messages contain the following four parts:

1. A banner:

```
                                            MON, JUL 10, 1991  4:48 PM
   HP36217-02A.E1.00      PASCAL Preprocessor/9000          ALLBASE/SQL
   (C) COPYRIGHT HEWLETT-PACKARD CO. 1982,1983,1984,1985,1986,1987,1988,
   1989,1990,1991. ALL RIGHTS RESERVED.
```

300/400   The banner for a Series 300 preprocessor is as follows:

```
                                            MON, JUL 10, 1991  4:48 PM
       HP79725A.E1.00          PASCAL Preprocessor/300          ALLBASE/SQL
       (C)COPYRIGHT HEWLETT-PACKARD CO.  1982,1983,1984,1985,1986,1987,1988,
       1989,1990,1991. ALL RIGHTS RESERVED.
```

2. A summary of the preprocessor invocation conditions:

```
   DBEnvironment        = ../sampledb/PartsDBE
   Module Name          = PASEX2
```

3. Warnings and errors encountered during preprocessing:

```
    32      SalesPriceInd    : SQLID;
            |
****** Unsupported type syntax for host variable. (DBERR 10933)


       SELECT PartNumber, PartName, SalesPrice INTO :PartNumber,
       :PartName, :SalesPrice :SalesPriceInd FROM PurchDB.Parts
       WHERE PartNumber = :PartNumber;

****** ALLBASE/SQL errors.  (DBERR 10952)
****** in SQL statement ending in line 128
*** ALLBASE/SQL alignment error on column 3 in buffer 5.  (DBERR 4200)

   There are errors.  No sections stored.
```

4. A summary of the results of preprocessing:

```
    2 ERRORS   0 WARNINGS
   END OF PREPROCESSING.
```

Both the banner and the preprocessing summary results are also echoed to the terminal.

As illustrated in Figure 2-14, a line number is often provided in sqlmsg. This line number references the line in the modified source file containing the command in question. A message accompanied by a number may also appear. You can refer to the *ALLBASE/SQL Message Manual* for additional information on the exception condition when these numbered messages appear.

```
$ more sqlmsg
                                         MON, JUL 10, 1991  4:48 PM
HP36217-02A.E1.00      PASCAL Preprocessor/9000        ALLBASE/SQL
(C)COPYRIGHT HEWLETT-PACKARD CO. 1982,1983,1984,1985,1986,1987,1988,
1989,1990,1991. ALL RIGHTS RESERVED.


DBEnvironment           = ../sampledb/PartsDBE
Module Name             = PASEX2

    32        SalesPriceInd   : SQLID
    |
******  Unsupported type syntax for host variable.  (DBERR 10933)


 SELECT PartNumber, PartName, SalesPrice INTO :PartNumber,
 :PartName, :SalesPrice :SalesPriceInd FROM PurchDB.Parts
 WHERE PartNumber= :PartNumber;

******  HP SQL query processing errors (DBERR 10952)
******  in SQL statement ending in line 128
*** HP SQL alignment error on column 3 in buffer 5.  (DBERR 4200)

There are errors.  No sections stored.
  2 ERRORS    0 WARNINGS
 END OF PROCESSING.
```

Figure 2-12. Sample sqlmsg Showing Errors

As Figure 2-15 illustrates, the preprocessor can terminate with the warning message:

```
****** HP SQL warnings. (DBWARN 10602)
```

when the name of an object in the source file does not match the name of any object in the system catalog. Although a section is stored for the semantically incorrect command, the section is marked as invalid and will not execute at run time if it cannot be validated.

```
$ more sqlmsg


                                        MON, JUL 10, 1991  4:48 PM
HP36217-02A.E1.00       PASCAL Preprocessor/9000           ALLBASE/SQL
(C)COPYRIGHT HEWLETT-PACKARD CO. 1982,1983,1984,1985,1986,1987,1988,
1989,1990,1991. ALL RIGHTS RESERVED.


DBEnvironment        = ../sampledb/PartsDBE
Module Name          = PASEX2

        SELECT PartNumber, PartName, SalesPrice INTO :PartNumber,
        :PartName, :SalesPrice :SalesPriceInd FROM PurchDB.Parts
        WHERE PartNumber = :PartNumber;

****** HP SQL warnings. (DBWARN 10602)
****** in SQL statement ending in line 128
*** Column PARTNUMBER not found. (DBERR 2211)

  1 Sections stored in DBEnvironment.

  0 ERRORS   1 WARNINGS
END OF PREPROCESSING
```

Figure 2-13. Sample sqlmsg Showing Warnings

Installable Module File

When the Pascal preprocessor stores a module in the system catalog of a DBEnvironment at preprocessing time, it places a copy of the module in an installable module file. The name of this file is ModifiedSourceFileName.sqlm. The module in this file can be installed into a DBEnvironment different from the DBEnvironment accessed at preprocessing time by using the INSTALL command in ISQL:

```
$ isql
```

*In order to install the module, you need CONNECT
or DBA authority in the target DBEnvironment:*

```
isql=> CONNECT TO '../sampledb/PartsDBE';
isql=> INSTALL;

File name> pasex2.sqlm;
Name of module in this file:  THOMAS.PASEX2
Number of sections installed:  1
COMMIT WORK to save to DBEnvironment.

isql=> COMMIT WORK;
isql=>
```

Stored Sections

In full preprocessing mode, the preprocessor stores a section for each embedded SQL
command *except*:

```
BEGIN DECLARE SECTION          INCLUDE
BEGIN WORK                     OPEN
CLOSE                          PREPARE
COMMIT WORK                    RELEASE
CONNECT                        ROLLBACK WORK
DECLARE CURSOR                 SAVEPOINT
DELETE WHERE CURRENT           START DBE
DESCRIBE                       STOP DBE
END DECLARE SECTION            SQLEXPLAIN
EXECUTE                        TERMINATE USER
EXECUTE IMMEDIATE              UPDATE WHERE CURRENT
                               WHENEVER
```

The commands listed above either require no authorization to execute or are executed
based on information contained in the compilable preprocessor output files. Note that if the
DELETE WHERE CURRENT or UPDATE WHERE CURRENT command is dynamically
preprocessed, a section *does* exist in the module.

When the preprocessor stores a section, it actually stores what is known as an input tree and
a run tree. The **input tree** consists of an uncompiled command. The **run tree** is the compiled,
executable form of the command.

If at run time a section is valid, ALLBASE/SQL executes the appropriate run tree when the SQL command is encountered in the application program. If a section is invalid, ALLBASE/SQL determines whether the objects referenced in the sections exist and whether current authorization criteria are satisfied. When an invalid section can be validated, ALLBASE/SQL dynamically recompiles the input tree to create an executable run tree and executes the command. When a section cannot be validated, the command is not executed, and an error condition is returned to the program.

ALLBASE/SQL creates the following three types of sections:

1. Sections for executing the SELECT command associated with a DECLARE CURSOR command.

2. Sections for executing the SELECT command associated with a CREATE VIEW command.

3. Sections for all other commands for which the preprocessor stores a section.

Figure 2-14 illustrates the kind of information in the system catalog that describes each type of stored section. The query result illustrated was extracted from the system view named SYSTEM.SECTION by using ISQL. The columns in Figure 2-14 have the following meanings:

■ NAME: This column contains the name of the module to which a section belongs. You specify a module name when you invoke the preprocessor; the module name is by default the program name from the Pascal program. If you are supplying a module name in a native language other than n-computer (ASCII), be sure it is in the same language as that of the DBEnvironment.

■ OWNER: This column identifies the owner of the module. You specify an owner name when you invoke the preprocessor; the owner name is by default the userid associated with the preprocessing session. If you are supplying an owner name in a native language other than n-computer (ASCII), be sure it is in the same language as that of the DBEnvironment.

■ DBEFILESET: This column indicates the DBEFileSet with which DBEFiles housing the section are associated.

■ SECTION: This column gives the section number. Each section associated with a module is assigned a number by the preprocessor as it parses the related SQL command at preprocessing time.

■ TYPE: This column identifies the type of section:

```
1 = SELECT associated with a cursor
2 = SELECT defining a view
0 = All other sections
```

■ VALID - This column identifies whether a section is valid or invalid:

```
0 = invalid
1 = valid
```

```
isql=> SELECT NAME,OWNER,DBEFILESET,SECTION,TYPE,VALID FROM SYSTEM.SECTION;

SELECT NAME,OWNER,DBEFILESET,SECTION,TYPE,VALID FROM SYSTEM.SECTION;
--------------------------------------------------------------------------
NAME              |OWNER      |DBEFILESET      |SECTION  |TYPE  |VALID
--------------------------------------------------------------------------
TABLE             |SYSTEM     |SYSTEM          |       0|     2|    0
COLUMN            |SYSTEM     |SYSTEM          |       0|     2|    0
INDEX             |SYSTEM     |SYSTEM          |       0|     2|    0
SECTION           |SYSTEM     |SYSTEM          |       0|     2|    0
DBEFILESET        |SYSTEM     |SYSTEM          |       0|     2|    0
DBEFILE           |SYSTEM     |SYSTEM          |       0|     2|    0
SPECAUTH          |SYSTEM     |SYSTEM          |       0|     2|    0
TABAUTH           |SYSTEM     |SYSTEM          |       0|     2|    0
COLAUTH           |SYSTEM     |SYSTEM          |       0|     2|    0
MODAUTH           |SYSTEM     |SYSTEM          |       0|     2|    0
GROUP             |SYSTEM     |SYSTEM          |       0|     2|    0
VIEWDEF           |SYSTEM     |SYSTEM          |       0|     2|    0
HASH              |SYSTEM     |SYSTEM          |       0|     2|    0
CONSTRAINT        |SYSTEM     |SYSTEM          |       0|     2|    0
CONSTRAINTCOL     |SYSTEM     |SYSTEM          |       0|     2|    0
CONSTRAINTINDEX   |SYSTEM     |SYSTEM          |       0|     2|    0
COLDEFAULT        |SYSTEM     |SYSTEM          |       0|     2|    0
TEMPSPACE         |SYSTEM     |SYSTEM          |       0|     2|    0
PARTINFO          |PURCHDB    |SYSTEM          |       0|     2|    0
VENDORSTATISTICS  |PURCHDB    |SYSTEM          |       0|     2|    0
PASEX2            |THOMAS     |SYSTEM          |       1|     0|    1
PASEX7            |THOMAS     |SYSTEM          |       1|     1|    1
PASEX7            |THOMAS     |SYSTEM          |       2|     0|    1
```

Figure 2-14. Information in SYSTEM.SECTION on Stored Sections

The first eighteen rows in this query result describe the sections stored for the system views. The next two rows describe the two views in the sample database: PurchDB.PartInfo and PurchDB.VendorStatistics. Views are always stored as invalid sections, because the run tree is always generated at run time.

The remaining rows describe sections associated with two preprocessed programs. Program pasex2 contains only one section, for executing the SELECT command in the program illustrated in Figure 2-6. Program pasex7 contains two sections, one for executing the SELECT command associated with a DECLARE CURSOR command and one for executing a FETCH command.

Stored sections remain in the system catalog until they are deleted with the DROP MODULE command or by invoking the preprocessor with the -d option:

```
isql=> DROP MODULE PASEX2;
```

         or

```
$ psqlpas ../sampledb/PartsDBE -i pasex2 -d
```

You must have proper table access authority to drop a module.

Stored sections are marked invalid when the following occur:

- The UPDATE STATISTICS command is executed.

- Tables accessed in the program are dropped, altered, or assigned new owners.

- Indexes or DBEFileSets related to tables accessed in the program are changed.

- Module owner authorization changes occur that affect the execution of embedded commands.

When an invalid section is validated at run time, the validated section is committed when the program issues a COMMIT WORK command. If a COMMIT WORK command is not executed, ALLBASE/SQL must validate the section again the next time the program is executed. For this reason, you should embed COMMIT WORK commands even following SELECT commands, since the COMMIT WORK command may be needed even when data is not changed by a program.

Invoking the Pascal Preprocessor

The Pascal preprocessor can be invoked to perform either of the following steps:

- Only check the syntax of embedded SQL commands.

- Check the syntax of embedded SQL commands, create compilable output, store a module in a DBEnvironment, and create an installable module.

This section describes how to invoke the preprocessor both interactively and in the background.

Syntax Checking Mode

You use the following command to only check the syntax of the SQL commands embedded in the source file.

Syntax

$$\text{psqlpas -s} \begin{bmatrix} \text{-i } SourceFileName \\ \text{-p } ModifiedSourceFileName \end{bmatrix} | \ldots |$$

Parameters

-s   causes the preprocessor to only check SQL syntax.

-i *SourceFileName*   identifies the source file.

-p *ModifiedSourceFile*   identifies the modified source code file.

Description

1. The preprocessor does not access a DBEnvironment when it is run in this mode.

2. When performing only syntax checking, the preprocessor does not convert the SQL commands into Pascal constructs. Therefore the modified source code file does not contain any preprocessor-generated calls to ALLBASE/SQL external procedures.

3. The include and installable module files are created, but incomplete.

Authorization

You do not need ALLBASE/SQL authorization when you use the preprocessor to only check SQL syntax. In other words, the system tables that store who has DBA, RESOURCE, and OWNER privileges on tables are not checked.

Example

```
$ psqlpas -s -i mysource


                                        MON JUL 10, 1991  4:48 PM
HP36217-02A.E1.00      PASCAL Preprocessor/9000        ALLBASE/SQL
(C)COPYRIGHT HEWLETT-PACKARD CO. 1982,1983,1984,1985,1986,1987,1988,
1989,1990,1991. ALL RIGHTS RESERVED.

Syntax checked.
1 ERRORS    0 WARNINGS
END OF PREPROCESSING.

$ more sqlmsg
                                        MON JUL 10, 1991  4:48 PM
HP36217-02A.E1.00      PASCAL Preprocessor/9000        ALLBASE/SQL
(C)COPYRIGHT HEWLETT-PACKARD CO. 1982,1983,1984,1985,1986,1987,1988,
1989,1990,1991. ALL RIGHTS RESERVED.


        SELECT PartNumber, PartName, SalesPrice INTO :PartNumber, :PartName,
        :SalesPrice :SalesPriceInd FROM PurchDB.Parts WHERE PartNumber =
        :PartNumber;

****** HP SQL errors.  (DBERR 10977)
****** in SQL statement ending in line 128
***    Syntax error.  (DBERR 1001)

Syntax checked.

  1 ERRORS    0 WARNINGS
END OF PREPROCESSING
```

Full Preprocessing Mode

You use the following command to both check SQL syntax and create output files that can be processed by the Pascal compiler. This command also stores a module in the DBEnvironment named and creates a file containing an installable version of the module.

Syntax

$$\texttt{psqlpas } DBEnvironmentName \begin{bmatrix} \texttt{-i } SourceFileName \\ \texttt{-p } ModifiedSourceFileName \\ \texttt{-o } OwnerName \\ \texttt{-m } ModuleName \\ \texttt{-d } \begin{bmatrix} \texttt{-r} \end{bmatrix} \end{bmatrix} \mid \dots \mid$$

Parameters

DBEnvironmentName	identifies the DBEnvironment in which a module is to be stored.
`-i` *SourceFileName*	identifies the name of the input file containing the source code to be preprocessed.
`-p` *ModifiedSourceFile Name*	identifies the name of the modified source code file. If a name is not specified, the preprocessor generated code is written to a file with the same name as SourceFileName but with the file extension .p.
`-o` *OwnerName*	associates the stored module with a user's userid, a ClassName, or a GroupName. You can specify an owner name for the module if you have DBA authority in the DBEnvironment where the module is to be stored. You can also specify a group as owner if you are a member of the group. If not specified, the owner name is your userid. Any object names in the SourceFileName not qualified with an owner name are qualified with this OwnerName.
`-m` *ModuleName*	assigns a name to the stored module. Module names must follow the rules governing ALLBASE/SQL basic names as described in the *ALLBASE/SQL Reference Manual* . If a module name is not specified with the -p option, the preprocessor uses the *ModifiedSourceFileName* as the module name. If a module name is not specified *without* the -p option, the preprocessor uses the *SourceFileName* as the module name.
`-d`	deletes any module currently stored in the DBEnvironment by the ModuleName and OwnerName specified in the command string. If not specified, any module having these names is not dropped, and existing RUN authorities for that module are preserved.
`-r`	is specified when the program being preprocessed already has a stored module and you want to revoke existing RUN authorities for that module. The -r option cannot be specified unless -d is also specified. If the -r option is not specified, it is assumed that all existing RUN authorities for that module are to be preserved.

Description

1. When the program being preprocessed already has a stored module, be sure to use the -d option, or else an error will result. Also, be sure that no one is currently executing the module when you invoke the preprocessor. To avoid conflicts, do your preprocessing in single user mode, during off hours.

2. The preprocessor starts a DBE session in the DBEnvironment named in the preprocessor command by issuing a CONNECT TO 'DBEnvironmentName' command. If the autostart flag is OFF, the DBE session can be initiated only after a START DBE command has been processed.

3. If the DBEnvironment to be accessed is operating in single user mode, preprocessing can occur only when another DBE session for the DBEnvironment does not exist.

4. When the preprocessor's DBE session begins, ALLBASE/SQL processes a BEGIN WORK command. When preprocessing is completed, the preprocessor submits a COMMIT WORK command, and any sections created are committed to the system catalog. If the preprocessor detects an error in the source file, it processes a ROLLBACK WORK command before terminating, and no sections are stored in the DBEnvironment. Preprocessor warnings do not prevent sections from being stored.

5. Since all preprocessor DBE sessions initiate only one transaction, any log file space used by the session is not available for re-use until after the session terminates. If rollforward logging is not in effect, you can issue the CHECKPOINT command in ISQL before preprocessing to increase the amount of available log space. Refer to the *ALLBASE/SQL Database Administration Guide* for additional information on log file space management, such as using the START DBE NEWLOG command to increase the size of the log and recovering log space when rollforward logging is in effect.

6. During preprocessing, system catalog pages accessed for embedded commands are locked. In multi-user mode, other DBE sessions accessing the same objects must wait, and the potential for a deadlock exists. Therefore minimize competing transactions when preprocessing an application program. Refer to the "Locks Held on the System Catalog by SQL Commands" appendix of the *ALLBASE/SQL Database Administration Guide* for information on operations that lock system catalog pages.

7. For improved runtime performance, use ISQL to issue the UPDATE STATISTICS command before preprocessing for each table accessed in a data manipulation command when an index on that table has been added or dropped and when data in the table is often changed.

8. If you specify an OwnerName or ModuleName in a language other than ASCII, be sure that the language you are using is also the language of the DBEnvironment in which the module will be stored.

Authorization

To preprocess a program for the first time in this mode, you need CONNECT or DBA authority in the DBEnvironment the program accesses. After a stored module exists, you need module OWNER or DBA authority in the DBEnvironment.

Example

```
$ psqlpas ../sampledb/PartsDBE -o Karen -m KarensPgm -i prog1.sql -d


                                        MON JUL 10, 1991  4:48 PM
HP36217-02A.E1.00      PASCAL Preprocessor/9000          ALLBASE/SQL
(C) COPYRIGHT HEWLETT-PACKARD CO. 1982,1983,1984,1985,1986,1987,1988,
1989,1990,1991.  ALL RIGHTS RESERVED.

  0 ERRORS   1 WARNINGS
END OF PREPROCESSING.


$ more sqlmsg


                                        MON JUL 10, 1991  4:48 PM
HP36217-02A.E1.00      PASCAL Preprocessor/9000          ALLBASE/SQL
(C) COPYRIGHT HEWLETT-PACKARD CO. 1982,1983,1984,1985,1986,1987,1988,
1989,1990,1991.  ALL RIGHTS RESERVED.


DBEnvironment        = ../sampledb/PartsDBE
Module Name          = KarensPgm

******  SELECT PartNumber, PartName, SalesPrice INTO :PartNumber, :PartName,
 :SalesPrice WHERE PartNumber = :PartNumber
******  HP SQL warnings.  (DBWARN 10602)           |
******  User Karen does not have SELECT authority on PurchDB.Parts (DBERR 2301)

  1 Sections stored in DBEnvironment.

  0 ERRORS   1 WARNINGS
END OF PROCESSING
$
```

Running the Preprocessor in the Background

You can run the Pascal preprocessor in the background, a practice useful for jobs that take a long time to complete. Running a job in the background frees the terminal for performing other foreground processes. In the following example, any messages usually sent to the terminal are piped to the system printer:

```
$ psqlpas -s -i myprogram | lp &
```

Preprocessing Errors

Several types of errors can occur while you are using the Pascal preprocessor, for example:

- Unexpected preprocessor or DBEnvironment termination.
- Preprocessor invocation errors.
- Source file errors.
- DBEnvironment errors.

Preprocessor or DBEnvironment Termination

Whenever the Pascal preprocessor stops running unexpectedly while you are using it in full preprocessing mode, sections stored during the preprocessor's DBE session are automatically dropped when the DBEnvironment is next started up. Unexpected preprocessor session termination occurs, for example, when a DBA issues a STOP DBE command during a preprocessor DBE session.

Preprocessor Invocation Errors

If the ALLBASE/SQL message catalog is not available or if the source file named cannot be found, preprocessing terminates with the following messages, respectively:

```
Message catalog not available - see System Administrator  (DBERR 100008)

File open Error.  (DBERR 10907)
```

In addition, the invocation line may name a DBEnvironment that does not exist or contain erroneous syntax:

```
******  Cannot connect to DBEnvironment.  (DBERR 10953)

ERRORS  Processing terminated prematurely.  (DBERR 10923)
```

Source File Errors

When the Pascal preprocessor encounters errors when parsing the source file, messages are placed in sqlmsg. Refer to the discussion earlier in this chapter under "ALLBASE/SQL Message File" for additional information on this category of errors.

DBEnvironment Errors

Some errors can be caused because of any of the following:

- A DBEnvironment is not started yet.
- Resources are insufficient.
- A deadlock has occurred.

Refer to the *ALLBASE/SQL Database Administration Guide* for information on handling DBEnvironment errors.

3

Embedding SQL Commands

In every ALLBASE/SQL Pascal program, you embed SQL commands in the declaration section and the procedure section of your program to:

①	Declare the SQL Communications Area (SQLCA).
②	Declare host variables.
③	Start a DBE session by connecting to the DBEnvironment.
④ ⑤	Define transactions.
⑥	Implicitly check the status of SQL command execution.
⑦	Terminate the DBE session.
⑧	Define or manipulate data in the DBEnvironment.
⑨	Explicitly check the status of SQL command execution.
⑩	Obtain error and warning messages from the ALLBASE/SQL message catalog.

The program listing shown in Figure 3-1 illustrates where in a main program you can embed SQL commands to accomplish the activities listed above. In a subprogram, host variable declarations cannot be in the global declaration part.

This chapter is a high-level road map to the logical and physical aspects of embedding SQL commands in a program. It addresses the reasons for embedding commands to perform the above activities. It also gives general rules for how and where to embed SQL commands for these activities. First however, it shows a program containing commands for the basic SQL functions listed above. Then it describes the general rules that apply when you embed *any* SQL command, referring to the numbered statements in the program.

```
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* This program illustrates the use of SQL's SELECT command to      *)
(* retrieve one row or tuple at a time.                             *)
(* BEGIN WORK is executed before the SELECT and a COMMIT WORK       *)
(* after the SELECT.  An indicator variable is also used for        *)
(* SalesPrice.                                                      *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

Program pasex2(input, output);

label
    1000,
    9999;

const
    OK            =        0;
    NotFound      =      100;
    MultipleRows  = -10002;
    DeadLock      = -14024;

var
    EXEC SQL INCLUDE SQLCA;                                    1

        (* Begin Host Variable Declarations *)
    EXEC SQL  Begin Declare Section;                           2
    PartNumber       : packed array[1..16] of char;
    PartName         : packed array[1..30] of char;
    SalesPrice       : longreal;
    SalesPriceInd    : SQLIND;
    SQLMessage       : packed array[1..132] of char;
    EXEC SQL  End Declare Section;
        (* End Host Variable Declarations *)

    Abort              : boolean;

procedure SQLStatusCheck;  (* Procedure to Display Error Messages *)
   Forward;

$PAGE $




(* Directive to set SQL Whenever error checking *)
EXEC SQL Whenever SqlError goto 1000;
```

Figure 3-1. Sample Program pasex2

```
Procedure ConnectDBE;  (* Procedure to Connect to PartsDBE *)
begin

writeln('Connect to PartsDBE');
EXEC SQL CONNECT TO '../sampledb/PartsDBE';                    3

end;  (* End of ConnectDBE Procedure *)


Procedure BeginTransaction;  (* Procedure to Begin Work *)
begin

writeln;
writeln('Begin Work');
EXEC SQL BEGIN WORK;                                          4

end;  (* End BeginTransaction Procedure *)


procedure EndTransaction;  (* Procedure to Commit Work *)
begin

writeln;
writeln('Commit Work');
EXEC SQL COMMIT WORK;                                         5

end;  (* End EndTransaction Procedure *)

(* Directive to reset SQL Whenever error checking *)
EXEC SQL WHENEVER SQLERROR CONTINUE;                          6


procedure TerminateProgram;   (* Procedure to Release PartsDBE *)
begin

writeln('Release PartsDBE');
EXEC SQL COMMIT WORK RELEASE;                                 7

writeln;
writeln('Terminating Program');
Goto 9999;  (* Goto exit point of main program *)

end;  (* End TerminateProgram Procedure *)
$PAGE $
```

Figure 3-1. Sample Program pasex2 (page 2 of 5)

```
procedure DisplayRow;    (* Procedure to Display Parts Table Rows *)
begin
writeln;
writeln('Part Number: ', PartNumber);
writeln('Part Name:   ', PartName);
if SalesPriceInd < 0 then
   writeln('Sales Price is NULL')
else
   writeln('Sales Price: ', SalesPrice:10:2);

end;  (* End of DisplayRow *)
$PAGE $

procedure SelectData; (* Procedure to Query Parts Table *)
begin

repeat

writeln;
prompt('Enter Part Number within Parts Table or "/" to STOP> ');
readln(PartNumber);
writeln;

if PartNumber[1] <> '/' then
  begin

  BeginTransaction;

  writeln('SELECT PartNumber, PartName, SalesPrice');
  EXEC SQL SELECT PartNumber, PartName, SalesPrice              8
            INTO :PartNumber,
                 :PartName,
                 :SalesPrice   :SalesPriceInd
             FROM PurchDB.Parts
            WHERE PartNumber = :PartNumber;

  if SQLCA.SQLWARN[0] in ['W','w'] then
     begin
     write('SQL WARNING has occurred. The following row');
     writeln('of data may not be valid.');
     end;
```

Figure 3-1. Sample Program pasex2 (page 3 of 5)

```
   case SQLCA.SQLCODE of                                             ⑨
   OK            : DisplayRow;
   NotFound      : begin
                        writeln;
                        writeln('Row not found!');
                     end;
   MultipleRows : begin
                        writeln;
                        writeln('WARNING: More than one row qualifies.');
                     end;
   otherwise        begin
                        SQLStatusCheck;
                        end;

   end;  (* case *)

   EndTransaction;

end;  (* End if *)
until PartNumber[1] = '/';

end;        (* End of SelectData Procedure *)

procedure SQLStatusCheck;  (* Procedure to Display Error Messages *)
begin

Abort := FALSE;
if SQLCA.SQLCODE < DeadLock then Abort := TRUE;

repeat
EXEC SQL SQLEXPLAIN :SQLMessage;                                    ⑩
writeln(SQLMessage);
until SQLCA.SQLCODE = 0;

if Abort then
  begin

  TerminateProgram;

  end;

end;  (* End SQLStatusCheck Procedure *)
$PAGE $
```

Figure 3-1. Sample Program pasex2 (page 4 of 5)

```
begin  (* Beginning of Program *)

write('Program to SELECT specified rows from ');
writeln('the Parts Table - PASEX2');
writeln;
writeln('Event List:');
writeln('  Connect to PartsDBE');
writeln('  Begin Work');
writeln('  SELECT specified row from Parts Table');
writeln('   until user enters "/" ');
writeln('  Commit Work');
writeln('  Disconnect from PartsDBE');
writeln;

ConnectDBE;
SelectData;
TerminateProgram;

(* Whenever Routine - Serious DBE Error *)
(* SQL Whenever SQLError Entry Point *)
1000:

  (* Begin *)
  SQLStatusCheck;
  TerminateProgram;
  (* End *)

(* Exit Point for the main program *)
9999:

end.   (* End of Program *)
```

Figure 3-1. Sample Program pasex2 (page 5 of 5)

General Rules for Embedding SQL

Embedded SQL commands must appear in certain locations within the Pascal program. Each embedded SQL command must be accompanied by a prefix and followed by a semicolon. Comments may be placed within an embedded command, and non-numeric literals in embedded commands may be continued from one line to another.

An embedded SQL command has no maximum length. A dynamic SQL command can be no longer than 2048 bytes as a literal string, but the maximum size of the host variable is 32,762.

Location of SQL Commands

SQL commands must be put into the following specific areas:

- BEGIN DECLARE SECTION and END DECLARE SECTION may appear in any declaration section in a main program. In a subprogram, these commands cannot appear in the global declaration section.
- INCLUDE SQLCA should be in the global declaration section of the main program. INCLUDE SQLDA should be in the global declaration of any main or in a declaration section of a procedure.

300/400   Series 300 and Series 400 systems do not allow subprograms.

- All other SQL commands may appear in any statement part.

Prefix and Suffix

Precede each SQL command with the prefix EXEC SQL and terminate each SQL command with a semicolon (;). The entire prefix, EXEC SQL, must be specified on one line. For example, the following examples are legal:

```
EXEC SQL SELECT PartName INTO :PartName
        FROM Purchdb.Parts WHERE PartNumber = :PartNumber;


EXEC SQL SELECT PartName
         INTO :PartName
         FROM Purchdb.Parts
         WHERE PartNumber = :PartNumber;
```

However, the following is not legal:

```
EXEC
    SQL SELECT PartName INTO :PartName
        FROM Purchdb.Parts WHERE PartNumber = :PartNumber;
```

Punctuation

SQL commands are always terminated with a semicolon (;).

Pascal Comments

You may insert Pascal comment lines within or between embedded SQL commands. A comment begins with { and closes with } or begins with (* and closes with *), as follows:

```
EXEC SQL SELECT PartNumber, PartName
      {  put the data into the following host variables  }
INTO :PartNumber, :PartName
      (* find the data in the following table *)
FROM Purchdb.Parts
      {  retrieve only data that satisfies this search condition *)
WHERE PartNumber = :PartNumber;
      (* end of command }
```

ALLBASE/SQL Comments

SQL comments can be inserted in any line of an SQL statement, except the last line, by prefixing the comment character with at least one space followed by two hyphens followed by one space:

```
EXEC SQL SELECT * FROM PurchDB.Parts     -- This code selects Parts Table values
          WHERE SalesPrice > 500.0;
```

The comment terminates at the end of the current line. (The decimal point in the 500 improves performance when being compared to SalesPrice, which also has a decimal; no data type conversion is necessary.)

Declaring the SQLCA

The SQL Communication Area (SQLCA) is an ALLBASE/SQL data structure that contains current information about a program's DBE session.

Every ALLBASE/SQL Pascal main program must contain an SQLCA declaration in the global declaration part. The SQLCA should be a VAR parameter in a subprogram to which the main program passes the SQLCA.

300/400   Series 300 and Series 400 systems do not allow subprograms.

As shown in Figure 3-1 at ①, you can declare the SQLCA by using the INCLUDE SQLCA command:

```
EXEC SQL INCLUDE SQLCA;
```

When the preprocessor parses this command, it inserts the following type definition into the modified source code file:

```
SQLCA: SQLCA_TYPE;
```

You can also declare the SQLCA explicitly as follows:

```
SQLCA: SQLCA_TYPE;
```

The Pascal preprocessor generates a complete Pascal declaration for the SQLCA in SQLTYPE. The following seven fields in the SQLCA record are available for programmers to use:

```
SQLCODE

SQLERRD[3]              SQLWARN[2]
SQLWARN[0]              SQLWARN[3]
SQLWARN[1]              SQLWARN[6]
```

Some values ALLBASE/SQL places into these fields indicate warning and error conditions that resulted when the immediately preceding SQL command was executed. Other values simply provide information attendant to normal command execution but are programmatically useful. For example, when you submit an UPDATE command, the number of rows updated is placed in SQLERRD [3]. If this value is greater than one, the program may want to advise the user of that condition and process a ROLLBACK WORK or COMMIT WORK command based on the user's response.

Examples discussed later in this chapter under "Implicit Status Checking" and "Explicit Status Checking" illustrate how the program in Figure 3-1 uses some of the SQLCA fields to determine the success or failure of SQL command execution.

Declaring Host Variables

Variables used in SQL commands in an executable section are known as **host variables**. All host variables used in a program must be declared in a declaration part. In a subprogram, the host variables must be declared in the declaration section of any level of a procedure—but not in the global declaration part. The host variable declarations must appear between the two following SQL commands:

```
EXEC SQL BEGIN DECLARE SECTION;
  .
  .    Host variables are declared here
  .    in Pascal data declarations.

EXEC SQL END DECLARE SECTION;
```

In Figure 3-1, host variable declarations start at ②.

You can put only one such declaration section in a program, and all host variables must be declared between the BEGIN/END DECLARE SECTION commands in a declaration part.

The SELECT command shown at ⑧ in Figure 3-1 uses three host variables for data, one for each of the columns in the PurchDB.Parts table. When used in an embedded SQL command, host variables are preceded with a colon, as follows:

```
:PartNumber
:PartName
:SalesPrice
```

For detailed information regarding host variables, see the chapter, "Host Variables."

Starting a DBE Session

As at ③ in Figure 3-1, in most application programs you embed the CONNECT command to start a DBE session in a program:

```
EXEC SQL CONNECT to '../sampledb/PartsDBE';
```

If autostart mode is ON at run time, this command starts a DBE session. If autostart mode is OFF, a DBA must issue a START DBE command before the program can be executed. Regardless of the autostart mode in effect, the program user must have CONNECT and RUN authority for this command to execute.

You can embed the START DBE command in a program to start a DBE session if the owner of the program has DBA authority. However, only one copy of the program can be executed at a time, by a user with DBA authority. For single-user DBEnvironments, this constraint poses no problem. In a multiuser environment, however, once a DBEnvironment is started, only the CONNECT command can be used to initiate additional DBE sessions.

Place the DBE session initiation command in an executable section of your program such that it executes at run time before all other SQL commands *except* the WHENEVER command.

Defining Transactions

You define transactions in an executable section to control what changes are committed to a DBEnvironment and when they are committed.

A transaction consists of all the SQL commands that are executed between a BEGIN WORK command and either a COMMIT WORK command or a ROLLBACK WORK command. When a COMMIT WORK command is successfully executed, all operations performed within the transaction are permanent in the DBEnvironment. When a ROLLBACK WORK command is executed, none of the changes remain in the DBEnvironment.

The number and duration of transactions in an application program depend on the following factors:

■ Concurrency: Concurrent DBE sessions may compete for data and index locks and buffers.

■ Update activities: Applications that are update intensive should issue COMMIT WORK commands more frequently to avoid data re-entry in the event of a failure.

■ Data consistency: Program changes to a table that are meaningful only if changes are made to another table should be committed or undone at the same time to ensure the data remains consistent.

For detailed information regarding transaction management, see the document, "Performance Guidelines".

The commands at ④ and ⑤ in Figure 3-1 start and end a transaction that consists of a single execution of the SELECT command in procedure SelectData.

The BEGIN WORK command in procedure BeginTransaction is optional, but recommended. If you omit a BEGIN WORK command, ALLBASE/SQL automatically issues a BEGIN WORK on your behalf before executing the first SQL command that requires that a transaction be in progress. However, such an implicit BEGIN WORK always has the default isolation level of Repeatable Read (RR). If you need to increase concurrency with a different isolation level, use an explicit BEGIN WORK command.

The COMMIT WORK command in procedure EndTransaction terminates the transaction after each execution of the SELECT command. Because the program does no DBEnvironment updates, this command is used to terminate the transaction even if an error is encountered. In programs that update data in a DBEnvironment, a ROLLBACK WORK command could be used to undo the effects of any database changes that occurred during a transaction before the error occurred.

Implicit Status Checking

You can use the WHENEVER command, as at ⑥ in Figure 3-1, to have ALLBASE/SQL examine SQLCA values and cause a specific action to be taken. The WHENEVER command is a preprocessor directive that specifies the action to be taken if an error or warning condition occurs during each execution of the SQL command:

```
EXEC SQL WHENEVER SQLERROR CONTINUE;
                      |          |
                      |          |
                      |          |
                      |        the action
                      |
                   the condition
```

Each WHENEVER command affects all ALLBASE/SQL commands that follow it in the source listing until another WHENEVER command is encountered.

If execution of the COMMIT WORK RELEASE command at ⑦ or the SELECT command at ⑧ causes an error condition, ALLBASE/SQL takes no special action. This occurs because the WHENEVER command shown above *precedes* the COMMIT WORK RELEASE and SELECT commands in the source listing.

The WHENEVER SQLERROR command at ⑥ turns off the implicit status checking of the WHENEVER SQLERROR command that appears earlier in the source listing:

```
EXEC SQL WHENEVER SQLERROR goto 1000;
```

This WHENEVER command specifies where to pass control when an error occurs during execution of the CONNECT, BEGIN WORK, or COMMIT WORK commands.

Although you can use a WHENEVER command to have ALLBASE/SQL examine the values in certain fields of the SQLCA, you can also examine the values yourself, as discussed under "Explicit Status Checking" later in this chapter.

Terminating a DBE Session

As illustrated at ⑦ in Figure 3-1, you can terminate a DBE session with the RELEASE option of the COMMIT WORK command. The program in Figure 3-1 terminates its DBE session whenever the following conditions occurs:

■ The user enters a slash (/) in response to the prompt in procedure SelectData.

■ The program encounters an error serious enough to set Abort to TRUE in procedure SqlStatusCheck.

■ The program encounters an error when processing the CONNECT, BEGIN WORK, or COMMIT WORK commands.

Defining and Manipulating Data

You embed data definition and data manipulation commands in statement parts.

Data Definition

You can embed the following SQL commands to create objects or change existing objects:

```
ALTER DBEFILE          CREATE INDEX          DROP GROUP
ALTER TABLE            CREATE TABLE          DROP INDEX
CREATE DBEFILE         CREATE VIEW           DROP MODULE
CREATE DBEFILESET      DROP DBEFILE          DROP TABLE
CREATE GROUP           DROP DBEFILESET       DROP VIEW
```

Data definition commands are useful for such activities as creating temporary tables or views to simplify data manipulation or creating an index that improves the program's performance:

```
EXEC SQL CREATE INDEX PartNameINDEX
                 ON Purchdb.Parts (PartName);
```

The index created with this command expedites data access operations based on partial key values:

```
EXEC SQL SELECT PartName
         INTO :PartName
          FROM Purchdb.Parts
         WHERE PartName LIKE :PartialKey;
```

Data Manipulation

SQL has the following four basic data manipulation commands:

■ SELECT - Retrieves data.

■ INSERT - Adds rows.

■ DELETE - Deletes rows.

■ UPDATE - Changes column values.

These four commands can be used for various types of data manipulation operations:

■ Simple data manipulation: operations that retrieve a single row, insert a single row, or delete or update a limited number of rows.

■ Sequential table processing: operations that use a cursor to operate on a row at a time within a set of rows. A **cursor** is a pointer the program advances through the set of rows.

■ Bulk operations: operations that manipulate multiple rows with a single execution of a data manipulation command.

■ Dynamic operations: operations specified by the user at run time.

In all non-dynamic data manipulation operations, you use host variables to pass data back and forth between your program and the DBEnvironment. Host variables can be used in the data manipulation commands wherever the syntax in the *ALLBASE/SQL Reference Manual* allows them.

The SELECT command shown at ⑧ in Figure 3-1 retrieves the row from PurchDB.Parts that contains a part number matching the value in the host variable named in the WHERE clause (PartNumber). The three values in the row retrieved are stored in three host variables named in the INTO clause (PartNumber, PartName, and SalesPrice). An indicator variable (SalesPriceInd) is also used in the INTO clause, to flag the existence of a null value in column SalesPrice:

```
EXEC SQL SELECT  PartNumber, PartName, SalesPrice
           INTO :PartNumber,
                :PartName,
                :SalesPrice :SalesPriceIND
           FROM  Purchdb.Parts
          WHERE  PartNumber = :PartNumber;
```

You can also use host variables in non-SQL statements; in this case, omit the colon:

```
SalesPrice  := Response;
EXEC SQL SELECT  COUNT(PartNumber)
           INTO :PartCount
           FROM  Purchdb.Parts
          WHERE  SalesPrice > :SalesPrice;
```

All host variables used in procedures must be declared in a declaration part, as discussed earlier in this chapter under "Declaring Host Variables".

Explicit Status Checking

In explicit status checking, shown at ⑨ in Figure 3-1, you explicitly examine an SQLCA field for a particular value, then perform an operation if the value exists. In this, example, the SQLCA field named SQLCODE is examined to determine whether it contains one of the following values:

- 0, indicating no error occurred.

- 100, indicating no rows qualified for the SELECT operation.

- -10002, indicating more than one row qualified for the SELECT operation.

If SQLCODE contains any other value, procedure SQLStatusCheck is executed. Values less than -14024 indicate errors serious enough to warrant terminating the program.

Obtaining ALLBASE/SQL Messages

As shown at ⑩ in Figure 3-1, you use the SQLEXPLAIN command to obtain a message from the ALLBASE/SQL message catalog that describes the condition related to certain SQLCA values:

```
EXEC SQL SQLEXPLAIN :SQLMessage;
```

ALLBASE/SQL puts a message from the ALLBASE/SQL message catalog into the host variable named SQLMessage, and the program displays the message.

Sometimes more than one message may be needed to completely describe how an SQL command executed. To retrieve all messages, the program in Figure 3-1 executes SQLEXPLAIN until SQLCODE is equal to zero. ALLBASE/SQL sets SQLCODE to zero when no more messages are available.

You can use SQLEXPLAIN in conjunction with either implicit or explicit status checking. Procedure SQLStatusCheck is also executed from the routine labelled 1000, which is used in conjunction with the first WHENEVER command in the program.

The default message catalog is /usr/lib/nls/n-computer/hpsqlcat. An alternate name for the same catalog is /usr/lib/hpsqlcat. For native language users, the catalog is /usr/lib/nls/$LANG/hpsqlcat, where $LANG is the current language. If this catalog is not available, ALLBASE/SQL issues a warning and uses the default catalog instead.

4

Host Variables

Host variables are variables used in SQL commands in the executable section. They are used to pass the following information between an application program and ALLBASE/SQL:

- Data values.

- Null value indicators.

- String truncation indicators.

- Bulk processing rows to process.

- Dynamic commands.

- Savepoint numbers.

- Messages from the ALLBASE/SQL message catalog.

- DBEnvironment names.

All host variables used in the executable section of a program or subprogram are declared in the declaration section. The declarations must be compatible with ALLBASE/SQL data types. The type declaration entries must also satisfy certain preprocessor criteria.

300/400   Series 300 and Series 800 systems do not allow subprograms.

This chapter identifies where in the procedure section you can use host variables and then discusses how to write type declarations that complement the way host variables are used.

Using Host Variables

Host variables are used in SQL commands as follows:

- To pass data values with the following data manipulation commands:

    ```
    SELECT
    INSERT
    DELETE
    UPDATE
    DECLARE
    FETCH
    REFETCH
    UPDATE WHERE CURRENT
    ```

■ To hold null value indicators in these data manipulation commands:

```
SELECT
INSERT
FETCH
REFETCH
UPDATE
UPDATE WHERE CURRENT
```

■ In queries to indicate string truncation and the string length before truncation.

■ To identify the starting row and the number of rows to process in the INTO clause of the following commands:

```
BULK SELECT
BULK INSERT
```

■ To pass dynamic commands at run time with the following commands:

```
PREPARE
EXECUTE IMMEDIATE
```

■ To hold savepoint numbers, which are used in the following commands:

```
SAVEPOINT
ROLLBACK WORK TO :savepoint
```

■ To hold messages from the ALLBASE/SQL message catalog, obtained by using the SQLEXPLAIN command.

■ To hold a DBEnvironment name in the CONNECT command.

Later in this section are examples illustrating where, in the commands itemized above, the SQL syntax supports host variables.

Host Variable Names

ALLBASE/SQL host variable names in Pascal programs must:

■ Contain from 1 to 30 bytes.

■ Conform to the rules for ALLBASE/SQL basic names.

■ Contain characters chosen from the following set: the 26 letters of the ASCII alphabet, the 10 decimal digits, a hyphen (-), or valid characters for any native language you are using.

■ Begin the name with an alphabetic character. However, avoid any user-defined names beginning with the prefix, SQL, as this can create a conflict with ALLBASE/SQL-generated names.

■ Not begin or end with a hyphen.

■ Not be the same as any ALLBASE/SQL or Pascal reserved word.

In all SQL commands containing host variable syntax, the host variable name must be preceded by a colon:

:*HostVariableName*

Input and Output Host Variables

Host variables can be used for input or for output:

■ Input host variables provide data for ALLBASE/SQL.

■ Output host variables contain data from ALLBASE/SQL.

Be sure to initialize an input host variable before using it. When using cursor operations with the SELECT command, initialize the input host variables contained in the select list and WHERE clause before you execute the OPEN command.

In the following SELECT command, the INTO clause contains two output host variables: PartNumber and PartName. ALLBASE/SQL puts data from the PurchDB.Parts table into these host variables. The WHERE clause contains one input host variable, PartNumber. ALLBASE/SQL reads data from this host variable to determine which row to retrieve.

```
EXEC SQL SELECT    PartNumber, PartName
          INTO  :PartNumber,
                :PartName
          FROM  PurchDB.Parts
          WHERE  PartNumber = :PartNumber;
```

In this example, the host variable, PartNumber, is used for both input and output.

Indicator Variables

A special type of host variable called an **indicator variable**, is used in SELECT, FETCH, UPDATE, UPDATE WHERE CURRENT, and INSERT commands to identify null values and in SELECT and FETCH commands to identify truncated output strings.

An indicator variable must appear in an SQL command *immediately after* the host variable whose data it describes. The host variable and its associated indicator variable are *not* separated by a comma. In SELECT and FETCH commands, an indicator variable is an output host variable containing one of the following indicators, which describe data ALLBASE/SQL returns:

```
  0     value is not null
 -1     value is null
 >0     string value is truncated; number indicates data length
           before truncation.
```

In the INSERT, UPDATE, and UPDATE WHERE CURRENT commands, an indicator variable is an input host variable. The value you put in the indicator variable tells ALLBASE/SQL when to insert a null value in a column:

```
>=0     value is not null
 <0     value is null
```

The following SELECT command uses an indicator variable, PartNameInd, for data from the PartName column. When this column contains a null value, ALLBASE/SQL puts a -1 into PartNameInd:

```
EXEC SQL SELECT  PartNumber, PartName
           INTO :PartNumber,
                :PartName :PartNameInd
            FROM PurchDB.Parts
           WHERE PartNumber = :PartNumber;
```

Any column *not* defined with the NOT NULL attribute may contain null values. In the PurchDB.Parts table, ALLBASE/SQL prevents the PartNumber column from containing null values, because it was defined as NOT NULL. In the other two columns, however, null values are allowed:

```
CREATE PUBLIC TABLE PurchDB.Parts
  (PartNumber      CHAR(16)      NOT NULL,
   PartName        CHAR(30),
   SalesPrice      DECIMAL(10,2) );
```

Null values have certain properties that you need to remember when manipulating data that may be null. For example, ALLBASE/SQL ignores columns or rows containing null values when evaluating an aggregate function (except that COUNT (*) includes all null values). Refer to the *ALLBASE/SQL Reference Manual* for a complete account of the properties of null values.

Be sure to use an indicator variable in the SELECT and FETCH commands whenever columns accessed may contain null values. A *runtime error* results if ALLBASE/SQL retrieves a null value and the program contains no indicator variable.

An indicator variable will also detect truncated strings in the SELECT and FETCH commands. In the SELECT command illustrated above, PartNameInd contains a value >0 when a part name is too long for the host variable declared to hold it. The value in PartNameInd indicates the actual length of the string *before* truncation.

Bulk Processing Variables

Bulk processing variables can be used with the BULK option of the SELECT or the INSERT command.

When used with the BULK SELECT command, two input host variables may be named following the array name in the INTO clause to specify how ALLBASE/SQL should store the query result in the array:

```
INTO :ArrayName [,:StartIndex [,:NumberOfRows]]
```

The StartIndex value denotes at which array element the query result should start. The NumberOfRows value is the maximum, total number of rows ALLBASE/SQL should put into the array:

```
EXEC SQL BULK SELECT  PurchasePrice * :Discount,
                      OrderQty,
                      OrderNumber
                INTO :OrdersArray,
                     :FirstRow,
                     :TotalRows
                FROM  PurchDB.OrderItems
               WHERE  OrderNumber
                      BETWEEN :LowValue AND :HighValue
            GROUP BY  OrderQty, OrderNumber;
```

ALLBASE/SQL puts the entire query result, or the number of rows specified in TotalRows, whichever is less, into the array named OrdersArray, starting at the array subscript stored in FirstRow. If neither of these input host variables is specified, ALLBASE/SQL stores as many rows as the array can hold, starting at OrdersArray[1]. If FirstRow plus TotalRows is greater than the size of the array, a runtime error occurs and the program aborts.

Bulk processing variables may be used with the BULK INSERT command to direct ALLBASE/SQL to insert only certain rows from the input array:

```
EXEC SQL BULK INSERT INTO   PurchDB.Orders
                   VALUES (:OrdersArray,
                           :FirstRow,
                           :TotalRows);
```

If a starting index or total number of rows is not specified, ALLBASE/SQL inserts, starting at the beginning of the array, as many rows as there are elements in the array.

Declaring Host Variables

In the declaration section, you declare all host variables you use in any executable section of your program.

Creating Declaration Sections

Host variables may be declared in either a global or local declaration section of a main program, but only in a local declaration section of a subprogram and only in a Level 1 procedure. You can reference the host variables in any level procedure. These declarations can appear in the main source code file and/or in any files included by this file.

300/400   Series 300 and Series 400 systems do not allow subprograms.

At run time, the scope of a host variable is the same as that of any other Pascal variables declared in the same declaration section. At preprocessing time, however, all host variable declarations are treated as global declarations.

In any declaration section, you declare host variables in what is known as a **declaration section**. A declaration section consists of the SQL command BEGIN DECLARE SECTION, one or more variable declarations, and the SQL command END DECLARE SECTION, as shown in Figure 4-1. More than one declaration section may appear in any declaration section. The same host variable name cannot appear in more than one declaration section.

Each host variable is declared by using a Pascal type description. The declaration contains the same components as any Pascal variable declaration:

```
EXEC SQL BEGIN DECLARE SECTION;
    OrderNumber  : integer;
    |              |
    |              |
    |            a data type
    |
    a data name



    EXEC SQL END DECLARE SECTION;
```

The data name must be the same as the corresponding host variable name in the statement section. The data type must satisfy ALLBASE/SQL data type and Pascal preprocessor requirements.

```
Program EXAMPLE (input, output);
var
.
.
.
EXEC SQL BEGIN DECLARE SECTION;
.
.  Declarations for global host variables.
.
EXEC SQL END DECLARE SECTION;
.
.
.
PROCEDURE QUERY;
var
.
.
.
EXEC SQL BEGIN DECLARE SECTION;
.
.  Declarations for local host variables.
.
EXEC SQL END DECLARE SECTION;
.
.
.
begin
.
.
.
end;
.
.
.
begin
.
.
.
end.
```

Figure 4-1. Host Variable Declarations

Declaring Variables for Data Types

Table 4-1 summarizes the syntax of data type descriptions for host variables holding each type of ALLBASE/SQL data. It also illustrates how to declare indicator variables, arrays for holding multiple rows, and host variables that hold dynamic commands, savepoint numbers, message catalog messages, and DBEnvironment names. Only the type descriptions shown in Table 4-1 are supported by the Pascal preprocessor. The preprocessor does not, for example, support user defined types.

You can also declare program variables that are not host variables within a declaration section. All variables that appear in a declaration section, however, must have Pascal data types illustrated in Table 4-1.

CHAR Data

You can insert strings ranging from 1 to 3996 characters into a CHAR column.

When ALLBASE/SQL assigns data to a char host variable, it adds blanks if necessary on the right of the string to fill up the accepting variable.

VARCHAR Data

VARCHAR strings can range from 1 to 3996 characters. ALLBASE/SQL stores only the actual value of the string, not any trailing blanks.

The string data type in Pascal is equivalent to the VARCHAR data type in ALLBASE/SQL. The string data type in Pascal stores the actual length of the string in a four-byte field preceding the string itself. The VendorRemarks column in the PurchDB.Vendors table is defined as VARCHAR(60). It is therefore declared as follows:

```
VendorRemarks    : string[60];
```

On output, you can use the Pascal strlen function to determine the actual length of data ALLBASE/SQL assigns to an output host variable declared as a string. On input, ALLBASE/SQL automatically stores only the actual value of the string.

SMALLINT Data

Values can range from -32768 to +32767 in a column defined as SMALLINT. When the Pascal preprocessor detects a host variable declared as SmallInt, it defines the host variable as follows in SQLTYPE:

```
type
   SmallInt = shortint;
```

INTEGER Data

Values can range from -2,147,483,648 to +2,147,483,647 in a column defined as INTEGER.

Table 4-1. Data Type Declarations

SQL DATA TYPES	PASCAL TYPE DESCRIPTION
CHAR(1)	*DataName* : char;
CHAR(n)	*DataName* : array [1..*n*] of char; or *DataName* : packed array [1..*n*] of char;
VARCHAR(*n*)	*DataName* : string[*n*];
BINARY(1)	*DataName* : char;
BINARY(n)	*DataName* : array [1..*n*] of char; or *DataName* : packed array [1..*n*] of char;
VARBINARY(*n*)	*DataName* : string[*n*];
SMALLINT	*DataName* : smallint;
INTEGER	*DataName* : integer;
REAL	*DataName* : real;
FLOAT	*DataName* : longreal;
DECIMAL	*DataName* : longreal;
DATE	*DataName* : packed array[1..10] of char; [1]
TIME	*DataName* : packed array[1..8] of char; [1]
DATETIME	*DataName* : packed array[1..23] of char; [1]
INTERVAL	*DataName* : packed array[1..20] of char; [1]

[1] Applies to default format specification only.

FLOAT Data

ALLBASE/SQL offers the option of specifying the precision of floating point data. You have the choice of a 4-byte or an 8-byte floating point number. The keywords REAL and FLOAT(1) through FLOAT(24) specifications map to a 4-byte float. The FLOAT(25) through FLOAT(53) and DOUBLE PRECISION specifications map to an 8-byte float.

The REAL data type could be useful when the number you are dealing with is very small, and you do not require a great deal of precision. However, it is subject to overflow and underflow errors if the value goes outside its range. It is also subject to greater rounding errors than double precision. With the DOUBLE PRECISION (8-byte float) data type, you can achieve significantly higher precision and have available a larger range of values.

By using the CREATE TABLE or ALTER TABLE command, you can define a floating point column by using a keyword from the following table. See the *ALLBASE/SQL Reference Manual* for complete syntax specifications.

Table 4-2. ALLBASE/SQL Floating Point Column Specifications

Possible Keywords	Range of Possible Values	Stored In and Boundary Aligned On
REAL or FLOAT(n) where $n = 1$ through 24	-3.402823 E$+38$ through -1.175495 E-38 and 1.175495 E-38 through 3.402823 E$+38$ and 0	4 bytes
DOUBLE PRECISION or FLOAT or FLOAT(n) where $n = 25$ through 53	-1.79769313486231 E$+308$ through -2.22507385850721 E-308 and $+2.22507385850721$ E-308 through $+1.79769313486231$ E$+308$ and 0	8 bytes

Floating Point Data Compatibility. Floating point data types are compatible with each other and with other ALLBASE/SQL numeric data types (DECIMAL, INTEGER, and SMALLINT). All arithmetic operations and comparisons and aggregate functions are supported.

BINARY Data

As with other data types, use the CREATE TABLE or ALTER TABLE command to define a BINARY or VARBINARY column. Up to 3996 bytes can be stored in such a column.

BINARY data is stored as a fixed length of left-justified bytes. It is zero padded up to the fixed length you have specified. VARBINARY data is stored as a variable length of left-justified bytes. You specify the maximum possible length. (Note that CHAR and VARCHAR data is stored in a similar manner except that CHAR data is blank padded.)

Binary Data Compatibility. BINARY and VARBINARY data types are compatible with each other and with CHAR and VARCHAR data types. They can be used with all comparison operators and the aggregate functions MIN and MAX, but arithmetic operations are not allowed.

Using the LONG Phrase with Binary Data Types. If the amount of data in a given column of a row can exceed 3996 bytes, it must be defined as a LONG column. Use the CREATE TABLE or ALTER TABLE command to specify the column as either LONG BINARY or LONG VARBINARY.

LONG BINARY and LONG VARBINARY data is stored in the database just as BINARY and VARBINARY data, except that its maximum possible length is practically unlimited.

Use LONG VARBINARY when saving space is your main consideration. However, LONG BINARY offers faster data access.

LONG BINARY and LONG VARBINARY data types are compatible with each other, but not with other data types. Also, the concept of inputting and accessing LONG column data differs from that of other data types. Refer to the *ALLBASE/SQL Reference Manual* for detailed syntax and to the chapter "Programming with Long Columns" for information about using LONG column data.

DECIMAL Data

The DECIMAL data type is not supported in Pascal, but it is compatible with a Pascal longreal data type. If you use the DECIMAL data type in a dynamically preprocessed PREPARE statement with an output data buffer or in BULK operations, you must code Pascal statements yourself to convert Binary Coded Decimal (BCD) representation to character representation. If you use an input buffer with dynamic preprocessing, you must also write code that converts the character representation to BCD format before the data is placed in the input buffer. An example of Pascal code to do this conversion is shown in Procedure BCDToString in the sample program pasex10a, Figure 10-9.

When you use DECIMAL values in arithmetic operations and certain aggregate functions, the precision and scale of the result are functions of the precisions and scales of the values in the operation. Refer to the *ALLBASE/SQL Reference Manual* for a complete account of how to calculate the precision and scale of DECIMAL results.

300/400   DECIMAL fields used in arithmetic operations on Series 300 and Series 400 systems are converted to FLOAT to perform the operation, then converted back to DECIMAL. This may result in a less precise rather than a more precise result, in contrast to what would normally be anticipated.

DATE, TIME, DATETIME, and INTERVAL Data

```
EXEC SQL BEGIN DECLARE SECTION;
     BatchStamp    : packed array[1..23] of char; (* DATETIME DATA TYPE *)
     TestDate      : packed array[1..10] of char; (* DATE DATA TYPE     *)
     TestDateInd   : SqlInd;
     TestStart     : packed array[1..8]  of char; (* TIME DATA TYPE     *)
     TestStartInd  : SqlInd;
     LabTime       : packed array[1..20] of char; (* INTERVAL DATA TYPE *)
     LabTimeInd    : SqlInd;
EXEC SQL END DECLARE SECTION;

(* DECLARE and OPEN CURSOR C1 here.  Nulls not allowed for BatchStamp. *)

EXEC SQL FETCH C1
         INTO  :BatchStamp,
               :TestDate  :TestDateInd,
               :TestStart :TestStartInd,
               :LabTime   :LabTimeInd;
```

Using Default Data Values

You can choose a default value other than NULL when you create or alter a table by using the DEFAULT specification. Then when data is inserted, and a given column is not in the insert list, the specified default value is inserted. Or when you alter a table, adding a column to existing rows, every occurrence of the column is initialized to the default value.

When a table or column is defined with the DEFAULT specification, you will not get an error if a column defined as NOT NULL is not specified in the insert list of an INSERT command. Without the DEFAULT specification, if a column is defined as NOT NULL, it must have some value inserted into it. However, if the column is defined with the DEFAULT specification, it satisfies both the requirement that it be NOT NULL and have some value, in this case, the default value (unless the DEFAULT value is NULL). If a column not in an insert list does allow a NULL, then a NULL is inserted instead of the default value.

Your default specification options are as follows:

- NULL.
- USER (this indicates the current DBEUserID).
- A constant.
- The result of the CURRENT_DATE function.
- The result of the CURRENT_TIME function.
- The result of the CURRENT_DATETIME function.

Complete syntax for the CREATE TABLE and ALTER TABLE commands as well as definitions of the above options are found in the *ALLBASE/SQL Reference Manual* .

In effect, by choosing any option other than NULL, you assure the column's value to be NOT NULL and of a particular format, unless and until you use the UPDATE command to enter another value.

In the following example, the OrderNumber column defaults to the constant 5, and it *is* possible to insert a NULL value into the column:

```
CREATE PUBLIC TABLE PurchDB.Orders (
               OrderNumber INTEGER DEFAULT 5,
               VendorNumber INTEGER,
               OrderDate   CHAR(8))
          IN OrderFS
```

However, suppose you want to define a column default and specify that the column cannot be null. In the next example, the OrderNumber column defaults to the constant 5, and it is *not* possible to insert a NULL value into this column:

```
CREATE PUBLIC TABLE PurchDB.Orders (
               OrderNumber INTEGER DEFAULT 5 NOT NULL,
               VendorNumber INTEGER,
               OrderDate   CHAR(8))
          IN OrderFS
```

Coding Considerations

Any default value must be compatible with the data type of its corresponding column. For example, when the default is an integer constant, the column for which it is the default must be created with an ALLBASE/SQL data type of INTEGER, REAL, or FLOAT.

In your application, you input or access data for which column defaults have been defined just as you would data for which defaults are not defined. In this chapter, refer to the section "Declaring Variables for Data Types" for information on using the data types in your program. Also refer to the section "Declaring Variables for Compatibility" for information relating to compatibility.

When the DEFAULT Clause Cannot be Used

■ You can specify a default value for any ALLBASE/SQL column except those defined as LONG BINARY or LONG VARBINARY. For information on these data types, see the section in this chapter titled "Using the LONG Phrase with Binary Data Types."

■ With the CREATE TABLE command, you can use either a DEFAULT NULL specification or the NOT NULL specification. An *error* results if both are specified for a column as in the next example:

```
CREATE PUBLIC TABLE PurchDB.Orders (
                OrderNumber INTEGER DEFAULT NULL NOT NULL,
                VendorNumber INTEGER,
                OrderDate    CHAR(8))
            IN OrderFS
```

Declaring Variables for Compatibility

Under the following conditions, ALLBASE/SQL performs data type conversion when executing SQL commands containing host variables:

■ When the data types of values transferred between your program and a DBEnvironment do not match.

■ When data of one type is moved to a host variable of a different type.

■ When values of different types appear in the same expression.

Data types for which type conversion can be performed are called compatible data types. Table 4-3 summarizes data type-host variable compatibility. It also points out which data type combinations are incompatible and which data type combinations are equivalent, i.e., require no type conversion. E describes an equivalent situation, C a compatible situation, and I an incompatible situation.

Table 4-3. Pascal Data Type Equivalency and Compatibility

ALLBASE/ SQL DATA TYPES	CHAR	STRING	SMALLINT	INTEGER	LONGREAL	PACKED ARRAY OF CHAR
CHAR	E	C	I	I	I	E
VARCHAR	C	E	I	I	I	C
BINARY	C	C	I	I	I	C
VARBINARY	C	C	I	I	I	C
DATE	E	C	I	I	I	I
TIME	E	C	I	I	I	I
DATETIME	E	C	I	I	I	I
INTERVAL	E	C	I	I	I	I
SMALLINT	I	I	E	C	C	I
INTEGER	I	I	C	E	C	I
DECIMAL	I	I	C	C	C	I
REAL	I	I	C	C	E	I
FLOAT	I	I	C	C	E	I

As the following example illustrates, the INFO command available in ISQL provides the information you need to declare host variables compatible with or equivalent to ALLBASE/SQL data types. It also provides the information you need to determine whether an indicator variable is needed to handle null values.

```
isql=> INFO PurchDB.OrderItems;

Column Name         Data Type (length)                    Nulls Allowed
--------------------------------------------------------------------
ORDERNUMBER         Integer                               NO
ITEMNUMBER          Integer                               NO
VENDPARTNUMBER      Char (16)                             YES
PURCHASEPRICE       Decimal (10,2)                        NO
ORDERQTY            Smallint                              YES
ITEMDUEDATE         Char (8)                              YES
RECEIVEDQTY         Smallint                              YES
```

The example identified in Figure 4-2 produces a single-row query result. The declare section contains data types equivalent to or compatible with the data types in the PurchDB.OrderItems table:

- OrderNumber is an integer variable because the column whose data it holds is INTEGER.

- PurchasePrice is declared as a LONGREAL variable because it holds a DECIMAL column.

- Discount is declared as a LONGREAL variable because it is used in an arithmetic expression with a DECIMAL column.

- OrderQty is declared as a SmallInt variable because it holds the SMALLINT result of a SMALLINT column, OrderQty.

- OrderQtyInd is an indicator variable, necessary because the resulting OrderQty can contain null values. Note in the INFO example above that this column allows null values.

```
   var
   .

   .

   .


   EXEC SQL BEGIN DECLARE SECTION;
   .

   .

   .


        Discount              : longreal;
        PurchasePrice         : longreal;
        OrderQty              : SmallInt;
        OrderQtyInd           : SqlInd;
        OrderNumber           : integer;

   .

   .

   .

   EXEC SQL END DECLARE SECTION;
   .

   .

   .

   begin
   .

   .

   .

        EXEC SQL SELECT  PurchasePrice * :Discount,
                         OrderQty,
                   INTO :PurchasePrice,
                        :OrderQty :OrderQtyInd
                   FROM  PurchDB.OrderItems
                   WHERE  OrderNumber = :OrderNumber
```

Figure 4-2. Declaring Host Variables for Single-Row Query Result

The example identified in Figure 4-3 is similar to that in Figure 4-2. This query, however, is a
BULK query, which may return a multiple-row query result. It also incorporates a HAVING
clause. Here are some points to notice:

■ OrdersArray is the name of the array for storing the query result. It can hold up to 26
rows. Each row in the array has the same format as that in the single-row query result just
discussed.

■ FirstRow and TotalRows are declared as SmallInt variables, since their maximum value is
the size of the array, in this case, 26.

■ GroupCriterion is an integer variable because its value is compared in the HAVING clause
with the result of a COUNT function, which is always an INTEGER value.

```
Program EXAMPLE (input, output);
.
.
.
var

EXEC SQL BEGIN DECLARE SECTION;
.
.
.

    Discount                    : longreal;
    OrdersArray                 : packed array[1..26] of packed record
    PurchasePrice               : longreal;
    OrderQty                    : SmallInt;
    OrderQtyInd                 : SqlInd;
    OrderNumber                 : integer;
       end;
    FirstRow                    : SmallInt;
    TotalRows                   : SmallInt;
    LowValue                    : integer;
    HighValue                   : integer;
    GroupCriterion              : integer;
EXEC SQL END DECLARE SECTION;
.
.
.
begin
.
.
.
    EXEC SQL BULK SELECT  PurchasePrice * :Discount,
                          OrderQty,
                          OrderNumber
                   INTO  :OrdersArray,
                         :FirstRow,
                         :TotalRows
                   FROM  PurchDB.OrderItems
                  WHERE  OrderNumber
                         BETWEEN :LowValue AND :HighValue
               GROUP BY  OrderQty, OrderNumber
                 HAVING  COUNT(ItemNumber) > :GroupCriterion;
```

Figure 4-3. Declaring Host Variables for Multiple-Row Query Result

String Data Conversion

When ALLBASE/SQL moves VARCHAR data to a packed array of CHAR variable of a larger size, it pads the string on the right with spaces to fill up the host variable. When ALLBASE/SQL stores the value in a string host variable into a CHAR column, it pads the value on the right with spaces to fill up the column.

String Data Truncation

If the target host variable used in a SELECT or FETCH operation is too small to hold an entire string, the string is truncated. You can use an indicator variable to determine the actual length of the string before truncation:

```
EXEC SQL BEGIN DECLARE SECTION.
    LittleString            : packed array[1..n] of char;
    LittleStringInd         : SqlInd;
.
.
.
EXEC SQL END DECLARE SECTION.
.
.
.
begin
.
.
.
    EXEC SQL SELECT  BigString
                INTO :LittleString :LittleStringInd;
                .
                .
                .
```

When the value in column BigString is too long to fit in host variable LittleString, ALLBASE/SQL puts the actual byte length of the string into indicator variable LittleStringInd.

If a column is too small to hold a string in an INSERT or an UPDATE operation, the string is truncated and stored, but ALLBASE/SQL gives no error or warning indication.

It is possible to store native language data in a character column defined as ASCII. It is the programmer's responsibility to verify the language definition of the column that is to receive the data. If the character column is defined for a native language, truncation always occurs on a proper character boundary for that language.

Numeric Data Conversion

When you use numeric data of different types in an expression or comparison operation, data types with less precision are converted into data types of greater precision. The result has the greater precision. ALLBASE/SQL numeric types available in Pascal have the following precedence, from highest to lowest:

1. FLOAT
2. DECIMAL
3. INTEGER
4. SMALLINT

The following example illustrates numeric type conversion:

```
EXEC SQL BEGIN DECLARE SECTION;
    Discount         : integer;
    MaxPurchasePrice : integer;
 .
 .
 .
EXEC SQL END DECLARE SECTION;
 .
 .
 .
begin
 .
 .
 .
    EXEC SQL SELECT  MAX(PurchasePrice) * :Discount
              INTO :MaxPurchasePrice
              FROM  PurchDB.OrderItems;
```

The select list of the query illustrated contains an aggregate function, MAX. The argument of the function is the PurchasePrice column, defined in the PartsDBE DBEnvironment as DECIMAL(10,2). Therefore the result of the function is DECIMAL. Since the host variable named Discount is declared as an integer, a data type compatible with DECIMAL, ALLBASE/SQL converts the value in Discount to a DECIMAL quantity having a precision of 10 and a scale of 0.

After multiplication, data conversion occurs again before the DECIMAL result is stored in the integer host variable MaxPurchasePrice . In this case, the fractional part of the DECIMAL value is truncated.

Refer to the *ALLBASE/SQL Reference Manual* for additional information on how type conversion can cause truncation and overflow of numeric values.

Declaring Variables for Program Elements

The following section discusses how to declare elements specific to ALLBASE/SQL programs. Table 4-4 provides the syntax of these special elements.

Table 4-4. Program Element Declarations

PROGRAM ELEMENT	PASCAL TYPE DESCRIPTION
Indicator variable	*IndVarName* : `SqlInd`;
Array of *n* rows	*ArrayName* : `packed or unpacked array` `[1..`*n*`] of packed or unpacked record`
Data values	*Column1Name* : *Valid data type;* *Column2Name* : *Valid data type;*
Indicator variable	*IndVarName* : `SqlInd`;
StartIndex	*StartIndexName* : `smallint or` *StartIndexName* : `integer`;
NumberOfRows	*NumRowsName* : `smallint or` *NumRowsName* : `integer`;
Dynamic commands	*CommandName* : `packed array[1..`*n*`] of char; or` *CommandName* : `string[`*n*`]`;
Savepoint numbers	*SavepointName* : `smallint; or` *SavepointName* ; `integer`;
Message catalog messages	*MessageName* : `packed array[1..`*n*`] of char; or` *MessageName* : `string[`*n*`]`;
DBEnvironment name	*DBEName* : `packed array[1..`*n*`] of char; or` *DBEName* : `string[`*n*`]`;

SQLCA Array

Every ALLBASE/SQL Pascal main program must have the SQL Communications Area (SQLCA) declared in the global declaration part. You can use the INCLUDE command to declare the SQLCA:

```
EXEC SQL INCLUDE SQLCA;
```

When the preprocessor parses this command, it inserts the following type definition into the modified source file:

```
sqlca:      Sqlca_Type:
```

Optionally, you can use this type definition in the global declaration part of your source file instead of using the INCLUDE command to declare the sqlca.

Refer to the chapter, "Runtime Status Checking," for further information regarding the sqlca.

Dynamic Processing Arrays

For programs which accept dynamic queries, you include three special declarations in a declaration part:

```
EXEC SQL INCLUDE SQLDA;
```

   This command causes the preprocessor to declare the SQLDA as
   type SQLDA_TYPE, defined in the preprocessor-generated type
   declaration include file.

```
SQLFmts            : array [1..MaxFmtArray]  of SQLFormat_type;
```

   This declaration identifies the format array and its size.
   MaxFmtArray is a constant representing the maximum number
   of columns you expect in the query result. SQLFORMAT_TYPE
   is defined in the type declaration include file.

```
DataBuffer         : packed array [1..MaxDataBuff] or char;
```

   This declaration identifies a data buffer and its size.
   MaxDataBuff is a constant representing the maximum number
   of bytes you will need to hold the number of rows you
   request in the SqlNRow field of the SQLDA.

See the chapter on "Using Dynamic Operations" for more information.

Bulk Processing Arrays

When you declare a structure array for holding the results of a BULK SELECT or BULK FETCH operation, ensure that you declare the fields in the same order as in the select list. (For single-row query results, however, the order of declaration does not have to match the select list order.) In addition, each indicator variable field must be declared in the declaration of the structure array immediately after the host variable field it describes. And if used, the bulk processing indicator variables (starting index and number of rows) are referenced in order, immediately following the reference to your array name. Refer back again to Figure 4-3 for an example.

Indicator Variables

Each indicator variable field used in a BULK SELECT must be declared immediately following the host variable field it describes, as shown in Figure 4-3. Figure 4-2 shows the indicator variable optionally declared immediately following the host variable field. If a column allows nulls, a null indicator *must* be declared for it.

Dynamic Commands

The maximum size for the host variables used to hold dynamic commands is 32,762 bytes. However, in Figure 4-4, the host variable is declared to hold the maximum size of dynamic command: 2048 bytes.

```
var
.
.
.

EXEC SQL BEGIN DECLARE SECTION;
.
.
.

   Dynamic Command          : string[2048];
.
.
.
EXEC SQL END DECLARE SECTION;
.
.
.
begin
.
.
.

    EXEC SQL PREPARE  CommandOnTheFly
               FROM :DynamicCommand  ;
```

Figure 4-4. Declaring Host Variables for Dynamic Commands

Savepoint Numbers

Savepoint numbers are positive numbers ranging from 1 to 2,147,483,647. A host variable for holding a savepoint number should be declared as an integer.

```
     ⋮
EXEC SQL BEGIN DECLARE SECTION;
      Savepoint1       : integer;
     ⋮
EXEC SQL END DECLARE SECTION;
     ⋮
    EXEC SQL SAVEPOINT :Savepoint1;
     ⋮
    EXEC SQL ROLLBACK WORK TO :Savepoint1;
```

Figure 4-5. Declaring Host Variables for Savepoint Numbers

Messages from the Message Catalog

The maximum size of a message catalog message is 256 bytes. Figure 4-6 illustrates how a host variable for holding a message might be declared.

```
Program EXAMPLE (input, output);
     ⋮
var

EXEC SQL BEGIN DECLARE SECTION;
      SQLMessage      :string[256]  ;
     ⋮
EXEC SQL END DECLARE SECTION;
     ⋮
begin
     ⋮
EXEC SQL SQLEXPLAIN :SQLMessage  ;
writeln (SQLMessage);
```

Figure 4-6. Declaring Host Variables for Message Catalog Messages

DBEnvironment Name

The maximum pathname (either relative or absolute) of a DBECon file is 128 bytes. The DBECon file name is the same as the DBEnvironment name. The name you store in this host variable does not have to be delimited by single quotation marks.

```
var
EXEC SQL BEGIN DECLARE SECTION;
.

.

.
SomDBE    : string[128];
.

.

.
EXEC SQL END DECLARE SECTION;
.

.

.
begin
prompt ('Enter DBEnvironment name> ');
readln (SomeDBE);
EXEC SQL CONNECT to :SomeDBE;
```

Figure 4-7. Declaring Host Variables for DBEnvironment Names

This host variable can be declared as a string or as a character array. In the example, it is declared as a character array large enough to hold the absolute file name of any DBECon file.

5

Runtime Status Checking and the SQLCA

This chapter examines the need for runtime status checking. It describes the SQLCA and the conditions under which its data items are set by ALLBASE/SQL. It also gives several examples of implicit and explicit status checking, some of which use SQLEXPLAIN to display a status message. Examples of handling specific status checking tasks are included under "Approaches to Status Checking."

When an SQL command is executed, ALLBASE/SQL returns information describing how the command executed. This information signals one or more of the following status conditions:

■ The command was successfully executed.

■ The command could not be executed because an error condition occurred, but the current transaction will continue.

■ No rows qualified for a data manipulation operation.

■ A specific number of rows were placed into output host variables.

■ A specific number of rows qualified for an INSERT, UPDATE, or DELETE operation.

■ The command was executed, but a warning condition resulted.

■ The command was executed, but a character string was truncated.

■ The command was executed, but a null value was eliminated from an aggregate function.

■ The command could not be executed because the number of variables in a SELECT or FETCH statement is unequal to the number of columns in the table being operated on. This applies to dynamic processing only.

■ The command could not be executed because an error condition necessitated rolling back the current transaction.

Based on this runtime status information, a program can COMMIT WORK, ROLLBACK WORK, continue, terminate, display a message, or perform some other appropriate activity.

■ You can use the WHENEVER command to perform **implicit status checking**. This means that ALLBASE/SQL checks the SQLCODE and SQLWARN0 values for you, then takes an action based on information you provide in the WHENEVER command.

■ You can write Pascal code that explicitly examines one or more of the seven SQLCA elements, then proceeds on the basis of their values. This kind of status checking is called **explicit status checking**.

■ You can use a *combination* of both implicit and explicit status checking.

In conjunction with status checking of any kind, you can use the SQLEXPLAIN command. This command retrieves a message from the ALLBASE/SQL message catalog that describes an error or warning condition.

When several errors or warnings occur, you can use SQLEXPLAIN to retrieve messages for all of them. Messages are available to your program with the most sever error appearing first. When ALLBASE/SQL rolls back the current transaction, the message indicating the roll back will be the first message, since it is the most severe. An example of this scenario is presented later in this chapter under "SQLCODE." Refer to the *ALLBASE/SQL Message Manual* for an explanation of all error and warning messages.

Purposes of Status Checking

Status checking is performed primarily for the following reasons:

- To gracefully handle runtime error and warning conditions.
- To maintain data consistency.
- To return information about the most recently executed command.

Handling Runtime Errors and Warnings

A program is said to be **robust** if it anticipates common runtime errors and handles them gracefully. In online applications, robust programs may allow the user to decide what to do when an error occurs rather than just terminating. This approach is useful, for example, when a deadlock occurs.

If a deadlock occurs, SQLCODE is set to -14024 and an SQLEXPLAIN call retrieves the following message:

```
Deadlock detected.  (DBERR 14024)
```

ALLBASE/SQL rolls back the transaction containing the SQL command that caused the deadlock. You may want to either give the user the option of restarting the transaction, automatically re-execute the transaction a finite number of times before notifying the user of the deadlock, or re-execute the transaction until the deadlock is resolved.

Maintaining Data Consistency

Two or more data values, rows, or tables are said to be **consistent** if they agree in some way. Changes to such interdependent values are either committed or rolled back at the *same* time in order to retain data consistency. In other words, the set of operations that form a transaction are considered as an **atomic** operation; either all or none of the operations are performed on the database. Status checking in this case determines whether to commit or roll back work.

For example, in the sample database, PartsDBE, each order is defined by rows in two tables: one row in the PurchDB.Orders table and one or more rows in the PurchDB.OrderItems table. A transaction that deletes orders from the database has to delete all the rows for a specific order from *both* tables to maintain data consistency. A program containing such a transaction should commit work to the database only if it is able to delete the row from the PurchDB.Orders table and delete all the rows for the same order from the PurchDB.OrderItems table:

```
EXEC SQL BEGIN WORK;
EXEC SQL DELETE FROM PurchDB.OrderItems
              WHERE OrderNumber = :OrderNumber;
```

If this command succeeds, the program submits the following command.

```
EXEC SQL DELETE FROM PurchDB.Orders
              WHERE OrderNumber = :OrderNumber;
```

If this command succeeds, the program submits a COMMIT WORK command. If this command does not succeed, the program submits a ROLLBACK WORK command to ensure that all rows related to the order are deleted at the same time.

Checking the Most Recently Executed Command

Depending on which ALLBASE/SQL command was most recently executed, you can make checks to insure that the command executed in a manner appropriate to the program's context. The following section, "Using the SQLCA," gives explanations based on each SQLCA element. Later in this chapter, the section "Explicit Status Checking Techniques" provides examples based on specific programming tasks.

Using the SQLCA

The SQLCA is used for communicating information between the application program and ALLBASE/SQL. SQL places information in the SQLCA each time it is called. Since there is no guarantee that information from one call to SQL will be present after the next call to SQL, any information needed from the SQLCA must be obtained after each call to ALLBASE/SQL.

Every ALLBASE/SQL Pascal main program must have the SQLCA declared in the global declaration section. You can use the INCLUDE command to declare the SQLCA:

```
EXEC SQL INCLUDE SQLCA;
```

When the Pascal preprocessor parses this command, it inserts the following type definition into the modified source file:

```
sqlca: Sqlca_Type
```

You can also use this type definition in the global declaration section of your source file instead of using the INCLUDE command to declare the SQLCA.

The Pascal preprocessor generates the following record declaration for sqlca_type in the type include file. This portion of the type include file contains some conditional statements. The entire type include file can be found in the chapter, "Using the ALLBASE/SQL Pascal Preprocessor." (It is recommended that you initialize the SqlcaId element to blanks, one time, before the first SQL statement in your program.)

```
    SQLCA_TYPE = record
      SQLCAID : packed array [1..8] of char;
      SQLCABC : integer;
      SQLCODE : integer;
      SQLERRM : string[255];
      SQLERRP : packed array [1..8] of char;
      SQLERRD : array [1..6] of integer;
  $if 'XOPEN_SQLCA'$
      SQLWARN0, SQLWARN1, SQLWARN2,
      SQLWARN3, SQLWARN4, SQLWARN5,
      SQLWARN6, SQLWARN7 : char;
  $else$
      SQLWARN : packed array [0..7] of char;
  $endif$
      SQLEXT  : packed array [1..8] of char;
      end;
```

The following elements in this record are available for you to use in status checking. The other elements are reserved for use by ALLBASE/SQL only.

```
    SQLCA.SQLCODE or SQLCODE
    SQLCA.SQLERRD[3]
    SQLCA.SQLWARN[0] or SQLCA.SQLWARN0
    SQLCA.SQLWARN[1] or SQLCA.SQLWARN1
    SQLCA.SQLWARN[2] or SQLCA.SQLWARN2
    SQLCA.SQLWARN[3] or SQLCA.SQLWARN3
    SQLCA.SQLWARN[6] or SQLCA.SQLWARN6
```

In conformance with XOPEN standards, SQLCODE can be used to address this particular element, and each SQLWARN element can be addressed without the use of square brackets. If you choose to use XOPEN standards addressing, you must include the following compiler directive in your source code:

```
    $ SET 'XOPEN_SQLCA=TRUE'$
```

(Note, use this compiler directive *only* if you are using XOPEN standards addressing.)

The SQLCA must be passed whenever you call a subprogram that executes SQL commands. The recommended method of doing so is to declare the SQLCA globally in the main program. This is true even though your main program contains no other SQL statement. The SQLCA must be a VAR parameter to these subprograms in order to save space and attain the best performance.

300/400 Series 300 and Series 400 systems do not allow subprograms.

Table 5-1. SQLCA Status Checking Fields

FIELD NAME	SET TO	CONDITION
SQLCA.SQLCODE or SQLCODE	0	No error occurred during command execution
	Less than 0	Error, command not executed
	100	No rows qualify for DML operation (does not apply to dynamic commands)
SQLCA.SQLERRD[3]	Number of rows put into output host variables	Data retrieval operation
	Number of rows processed	Data change operation
	0	Error in single row data change operation
	0	SQLCODE equals 100
SQLCA.SQLWARN[0] or SQLCA.SQLWARN0	W	Warning, command not properly executed
SQLCA.SQLWARN[1] or SQLCA.SQLWARN1	W	At least one character string value was truncated when being stored in a host variable
SQLCA.SQLWARN[2] or SQLCA.SQLWARN2	W	At least one null value was eliminated from the argument set of an aggregrate function
SQLCA.SQLWARN[3] or SQLCA.SQLWARN3	W	For dynamic commands only, when the number of host variables in a SELECT or FETCH is unequal to the number of columns in the table being operated on
SQLCA.SQLWARN[6] or SQLCA.SQLWARN6	W	The current transaction was rolled back

SQLCODE

SQLCODE can contain one of the following values:

- 0, when an SQL command executes without generating a warning or error condition.

- A negative number, when an SQL command cannot be executed because an error condition exists.

- 100, when no row qualifies for one of the following commands, but no error condition exists:

```
SELECT                             FETCH
INSERT                             BULK FETCH
UPDATE (non-dynamic execution only)   UPDATE WHERE CURRENT
DELETE (non-dynamic execution only)   DELETE WHERE CURRENT
BULK SELECT
```

Note that when you prepare and execute UPDATE or DELETE commands and no rows qualify for the operation, SQLCODE is not set to 100. You can use SQLCA.SQLERRD[3] to detect this condition, as discussed later in this chapter.

Negative SQLCODE values are the same as the numbers associated with their corresponding messages in the ALLBASE/SQL message catalog. For example, the error message associated with an SQLCODE of -2613 is:

```
Precision digits lost in decimal operation MULTIPLY.  (DBERR 2613)
```

SQLCODE is set by all SQL commands except the following directives:

```
BEGIN DECLARE SECTION
DECLARE
END DECLARE SECTION
INCLUDE
WHENEVER
```

When SQLCODE is -4008, -14024, or a greater negative value than -14024, ALLBASE/SQL automatically rolls back the current transaction. When this condition occurs, ALLBASE/SQL also sets SQLWARN[6] to W. Refer to the discussion later in this chapter on SQLWARN[6] for more on this topic.

More than one SQLCODE is returned when more than one error occurs. For example, if you attempt to execute the following SQL command, two negative SQLCODE values result:

```
EXEC SQL ADD PUBLIC, GROUP1 TO GROUP GROUP1;
```

The following SQLCODES associated with the two errors are:

-2308, which indicates the reserved name PUBLIC is invalid.

-2318, which indicates you cannot add a group to itself.

To obtain all SQLCODEs associated with the execution of an SQL command, you execute the
SQLEXPLAIN command until SQLCODE is 0:

```
      if SQLCA.SQLCODE = 100 then
        writeln('No rows qualified for this operation.');
      else
        if SQLCA.SQLCODE < 0 then SQLStatusCheck;
   .
   .
   .

  procedure SQLStatusCheck;
  begin
    repeat
    EXEC SQL SQLEXPLAIN :SQLMessage;
    writeln(SQLMessage);
    until SQLCA.SQLCODE = 0;
  end;
```

The procedure named SQLStatusCheck is executed when SQLCODE is a negative number.
Before executing SQLEXPLAIN for the first time, the program has access to the first
SQLCODE returned. Each time SQLEXPLAIN is executed subsequently, the next SQLCODE
becomes available to the program, and so on until SQLCODE equals 0.

This example explicitly tests the value of SQLCODE twice: first to determine whether it is
equal to 100, then to determine whether it is <0. If the value 100 exists, no error will have
occurred and the program will display the message, "No rows qualify for this operation."

It is necessary for the program to display its own message in this case because SQLEXPLAIN
messages are available to your program only when SQLCODE contains a negative number or
when SQLWARN[0] contains a W.

The SQLCODE is also used in implicit status checking in the following situations:

- ALLBASE/SQL tests for the condition SQLCODE less than 0 when you use the
  SQLERROR option of the WHENEVER command.

- ALLBASE/SQL tests for the condition SQLCODE equal to 100 when you use the NOT
  FOUND option of the WHENEVER command.

In the following situation, when ALLBASE/SQL detects a negative SQLCODE, the code
routine at label 2000 is executed. When ALLBASE/SQL detects an SQLCODE of 100, the
code routine at label 4000 is executed instead, as follows:

```
  EXEC SQL WHENEVER SQLERROR GOTO 2000;
  EXEC SQL WHENEVER NOT FOUND GOTO 4000;
```

WHENEVER commands remain in effect for *all* SQL commands that appear physically after
them in the source program until another WHENEVER command for the same condition
occurs.

The scope of WHENEVER commands is fully explained later in this chapter under "Implicit
Status Checking Techniques."

SQLERRD[3]

SQLERRD[3] can contain one of the following values:

- 0, when SQLCODE is 100 or when one of the following commands causes an error condition:

```
INSERT
UPDATE
DELETE
UPDATE WHERE CURRENT
DELETE WHERE CURRENT
```

    If an error occurs during execution of INSERT, UPDATE, or DELETE, one or more rows may have been processed prior to the error. In these cases, you may want to either COMMIT WORK or ROLLBACK WORK, depending on the application. For example, if all or no rows should be updated for logical data consistency, use ROLLBACK WORK. However, if logical data consistency is not an issue, COMMIT WORK may minimize re-preprocessing time.

- A positive number, when SQLCODE is 0. In this case, the positive number provides information about the number of rows processed in the following data manipulation commands:

    The number of rows inserted, updated, or deleted in one of the following operations:

```
BULK INSERT
INSERT
UPDATE
DELETE

UPDATE WHERE CURRENT
DELETE WHERE CURRENT
```

    The number of rows put into output host variables when one of the following commands is executed:

```
SELECT
BULK SELECT
FETCH
BULK FETCH
```

- A positive number when SQLCODE is less than 0. In this case, SQLERRD[3] indicates the number of rows that were successfully retrieved or inserted prior to the error condition:

```
BULK SELECT
BULK FETCH
BULK INSERT
```

    As in the case of INSERT, UPDATE, and DELETE, mentioned above, you can use either a COMMIT WORK or ROLLBACK WORK command, as appropriate.

SQLCA.SQLWARN[0]

A W in SQLWARN[0] in conjunction with a 0 in SQLCODE indicates that the SQL command just executed caused a warning condition.

Warning conditions flag unusual but not necessarily important conditions. For example, if a program attempts to submit an SQL command that grants an already existing authority, a message such as the following would be retrieved when SQLEXPLAIN is executed:

    User peg already has DBA authorization. (DBWARN 2006)

In the case of the following warning, the situation may or may not indicate a problem:

    A transaction in progress was aborted. (DBWARN 2010)

This warning occurs when a program submits a RELEASE command without first terminating a transaction with a COMMIT WORK or ROLLBACK WORK. If the transaction did not perform any UPDATE, INSERT, or DELETE operations, this situation will not cause work to be lost. If the transaction *did* perform UPDATE, INSERT, or DELETE operations, the database changes are rolled back when the RELEASE command is processed.

You retrieve the appropriate warning message by using SQLEXPLAIN. Note that you *cannot* explicitly test sqlwarn[0] the way you can test SQLCODE, since sqlwarn[0] always contains W when a warning occurs.

An error and a warning condition may exist at the same time. In this event, SQLCODE is set to a negative number, and sqlwarn[0] is set to W. Messages describing all the warnings and errors can be displayed as follows:

```
        if SQLCA.SQLCODE <> 0 then
          repeat
          DisplayMessage;
          until SQLCA.SQLCODE = 0;
        .
        .
        .
  procedure DisplayMessage;
  begin
    EXEC SQL SQLEXPLAIN  :StatusMessage;
    writeln(StatusMessage);
  end;
```

If multiple warnings but no errors result when ALLBASE/SQL processes a command, SQLWARN[0] is set to W and remains set until the last warning message has been retrieved by SQLEXPLAIN or another SQL command is executed. In the following example, DisplayWarning is executed when this condition exists.

```
if ((SQLCA.SQLCODE = 0) and (SQLCA.SQLWARN[O] = 'W')) then
  repeat
  DisplayWarning;
  until SQLCA.SQLWARN[O] <> 'W';
  .
  .
  .
procedure DisplayWarning;
begin
  EXEC SQL SQLEXPLAIN :StatusMessage;
  writeln(StatusMessage);
end;
```

When you use the SQLWARNING option of the WHENEVER command, ALLBASE/SQL checks for a W in SQLWARN[0]. You can use the WHENEVER command to do *implicit* status checking (equivalent to that done *explicitly* above) as follows:

```
EXEC SQL WHENEVER SQLWARNING GOTO 3000;
EXEC SQL WHENEVER SQLERROR GOTO 2000;
```

SQLCA.SQLWARN[1]

A W in sqlwarn[1] indicates truncation of at least one character string value when the string was stored in a host variable. Any associated indicator variable is set to the value of the string length before truncation, for example:

For example:

```
EXEC SQL SELECT  PartNumber,
                 PartName
           INTO :PartNumber
                :PartName :PartNameInd
           FROM  PurchDB.Parts
          WHERE  PartNumber = :PartNumber;
```

If PartName was declared as a character array of 20 bytes, and the PartName column in the PurchDB.Parts table has a length of 30 bytes, then SQL performs the following tasks:

- SQLWARN[1] is set to W.

- PartNameInd is set to 30 (the length of PartName in the table).

- SQLCODE is set to 0.

- SQLEXPLAIN retrieves the message:

```
        Character string truncation during storage in host variable.
        (DBWARN 2040)
```

SQLCA.SQLWARN[2]

A W in sqlwarn[2] indicates that at least one null value was eliminated from the argument set of an aggregrate function.

For example:

```
EXEC SQL SELECT  MAX(OrderQty)
           INTO :MaxOrderQty
           FROM  PurchDB.OrderItems;
```

If any OrderQty values are null:

- SQLWARN[2] is set to W.

- SQLCODE is set to 0.

- SQLEXPLAIN retrieves the message:

```
        NULL values eliminated from the argument of an aggregate
        function.   (DBWARN 2041)
```


SQLCA.SQLWARN[3]

A W in sqlwarn[3] indicates that the number of columns specified in a *dynamic* SELECT or FETCH statement is unequal to the number of columns indicated in the sqld field of the SQLDA. Under normal circumstances, this error does not occur, because the DESCRIBE command sets the sqld field correctly. Look at this example:

```
EXEC SQL PREPARE DynamicCommand from 'SELECT PartNumber, PartName
                                      FROM PurchDB.Parts;';

EXEC SQL DESCRIBE DynamicCommand INTO SQLDA; /*SQLDA.SQLD is always set
                                               at DESCRIBE by ALLBASE/SQL.*/
EXEC SQL DECLARE DynamicCursor FOR DynamicCommand;
EXEC SQL OPEN DynamicCursor;
/* Set up the SQLDA for a fetch. */
begin
  with SQLDA do
    begin
      SqlBufLen := sizeof(DataBuffer);
      SqlNRow := SqlBufLen DIV SqlRowLen;
      SqlRowBuf := waddress(DataBuffer);
      Sqld := 1;   /* Oops!  sqlda.sqld is incorrectly reset by the program. */
                   /* You should NEVER do this. */
                   /* Only ALLBASE/SQL should set this field. */
    end;
end;
.
/* Do the fetch. */
EXEC SQL FETCH DynamicCursor USING DESCRIPTOR SQLDA;
```

The FETCH will fail and ALLBASE/SQL performs the following tasks:

- SQLWARN[3] is set to W.

- SQLCODE is set to -2762.

- SQLEXPLAIN retrieves the message:

```
Select list has ! items and host variable buffer has !.
(DBERR 2762)
```

SQLCA.SQLWARN[6]

When an error occurs that causes ALLBASE/SQL to roll back the current transaction, SQLWARN[6] is set to W. ALLBASE/SQL automatically rolls back transactions when SQLCODE is equal to -4008, or equal to or less than -14024.

When such errors occur, ALLBASE/SQL does the following:

- Sets SQLWARN[6] to W.

- Sets SQLWARN[0] to W.

- Sets SQLCODE to a negative number.

If you want to terminate your program any time ALLBASE/SQL has to roll back the current transaction, you can just test sqlwarn[6].

```
if SQLCA.SQLCODE < 0 then
  if SQLCA.SQLWARN[6] = 'W' then
    begin
    SQLStatusCheck;
    TerminateProgram;
    end
  else
    SQLStatusCheck;
```

In this example, the program executes procedure SQLStatusCheck when an error occurs. The program terminates whenever ALLBASE/SQL has rolled back a transaction, but continues if an error has occurred but was not serious enough to cause transaction roll back.

Approaches to Status Checking

This section presents examples of how to use implicit and explicit status checking and to notify program users of the results of status checking.

Implicit status checking is useful when control to handle warnings and errors can be passed to *one* predefined point in the program. Explicit status checking is useful when you want to test for specific SQLCA values before passing control to *one of several* locations in your program.

Error and warning conditions detected by either type of status checking can be conveyed to the program user in various ways:

- SQLEXPLAIN can be used one or more times after an SQL command is processed to retrieve warning and error messages from the ALLBASE/SQL message catalog. (The ALLBASE/SQL message catalog contains messages for every negative SQLCODE and for every condition that sets SQLWARN0.)

- Your own messages can be displayed when a certain condition occurs.

- You can choose not to display a message; for example, if a condition exists that is irrelevant to the program user or when an error is handled internally by the program.

Implicit Status Checking Techniques

The WHENEVER command has two components: a **condition** and an **action**. The command syntax format is:

```
EXEC SQL WHENEVER Condition Action;
```

There are three possible WHENEVER conditions:

- SQLERROR

  If WHENEVER SQLERROR is in effect, ALLBASE/SQL checks for a negative SQLCODE after processing any SQL command *except*:

  ```
  BEGIN DECLARE SECTION
  DECLARE
  END DECLARE SECTION
  INCLUDE
  SQLEXPLAIN
  WHENEVER
  ```

- SQLWARNING

  If WHENEVER SQLWARNING is in effect, ALLBASE/SQL checks for a W in SQLWARN0 after processing any SQL command *except*:

  ```
  BEGIN DECLARE SECTION
  DECLARE
  END DECLARE SECTION
  INCLUDE
  SQLEXPLAIN
  WHENEVER
  ```

■ NOT FOUND

If WHENEVER NOT FOUND is in effect, ALLBASE/SQL checks for the value 100 in SQLCODE after processing a SELECT or FETCH command.

A WHENEVER command for each of these conditions can be in effect at the same time.

There are three possible WHENEVER actions:

■ STOP

If WHENEVER Condition STOP is in effect, ALLBASE/SQL rolls back the current transaction and terminates the DBE session and the program when the condition exists.

■ CONTINUE

If WHENEVER Condition CONTINUE is in effect, program execution continues when the condition exists. Any earlier WHENEVER command for the same condition is cancelled.

■ GOTO LineLabel.

If WHENEVER Condition GOTO LineLabel is in effect, the code routine located at that alpha-numeric line label is executed when the condition exists. The line label must appear in the code block where the GOTO is executed. GOTO and GO TO forms of this action have exactly the same effect.

Any action may be specified for any condition.

The WHENEVER command causes the preprocessor to generate status-checking and status-handling code for each SQL command that comes after it *physically* in the program until another WHENEVER command for the same condition is found. In the following program sequence, for example, the WHENEVER command in Procedure1 is in effect for SQLCommand1, but not for SQLCommand2, even though SQLCommand1 is executed first at run time:

```
  .
  .
  .
procedure Procedure2;
begin
  EXEC SQL SQLCommand2;
end;
procedure Procedure1;
begin
  EXEC SQL WHENEVER SQLERROR GOTO 2000;
  EXEC SQL SQLCommand1;
end;
  .
  .
  .
  Procedure1;
  Procedure2;
```

The code that the preprocessor generates depends on the condition and action in a WHENEVER command. In the previous example, the preprocessor inserts a test for a negative SQLCODE and a sentence that invokes the code at Line Label 2000, as follows:

```
$Skip_Text ON$
EXEC SQL WHENEVER SQLERROR GOTO 2000;
$Skip_Text OFF$                                        .

$Skip_Text ON$
EXEC SQL SQLCommand1;
$Skip_Text OFF$
Statements for executing SQLCommand1 appear here
if SQLCA.SQLCODE < 0 then
  goto 2000;
```

As the previous example illustrates, you can pass control to an exception-handling paragraph with a WHENEVER command, but you use a GOTO statement with a numeric line label, rather than a procedure name. Therefore after the exception-handling paragraph is executed, control cannot *automatically* return to the paragraph which invoked it. You must use another GOTO to explicitly pass control to a specific point in your program:

```
(* WHENEVER Routine -- SQL Error *)
2000:
   if SQLCA.SQLCODE < -14024 then
     TerminateProgram;
   else
     repeat
     EXEC SQL SQLEXPLAIN :SQLMessage;
     writeln(SQLMessage);
     until SQLCA.SQLCODE = 0;
     GOTO 500;      (* Goto Restart/Reentry point of main program *)
```

This exception-handling routine *explicitly* checks the first SQLCODE returned. The program terminates, or it continues from the Restart/Reentry point after all warning and error messages are displayed. Note that a GOTO statement was required in this paragraph in order to allow the program to continue. Using a GOTO statement may be impractical when you want execution to continue from different places in the program, depending on the part of the program that provoked the error. This situation is discussed under "Explicit Status Checking Techniques" later in the chapter.

Program Illustrating Implicit and Explicit Status Checking

The program in Figure 5-1 contains five WHENEVER commands to demonstrate implicit status checking. It also uses two explicit status checking routines.

- The WHENEVER command numbered 1 handles errors associated with the following commands:

      CONNECT
      BEGIN WORK
      COMMIT WORK

- The WHENEVER command numbered 2 turns off the first WHENEVER command.

- The WHENEVER commands numbered 3 through 5 handle warnings and errors associated with the SELECT command.

The code routine located at Label 1000 is executed when an error occurs during the processing of session related and transaction related commands. The program terminates after displaying all available error messages. If a warning condition occurs during the execution of these commands, the warning condition is ignored, because the WHENEVER SQLWARNING CONTINUE command is in effect by default.

The code routine located at Label 2000 is executed when an error occurs during the processing of the SELECT command. Procedure SQLStatusCheck is executed.

SQLStatusCheck explicitly examines SQLCODE to determine whether a deadlock or shared memory problem occurred (SQLCODE = -14024 or -4008) or whether the error was serious enough to warrant terminating the program (SQLCODE < -14024), for example:

- If a deadlock or shared memory problem occurred, the program attempts to execute the SelectData procedure as many as three times before notifying the user of the situation.

- If SQLCODE contains a value less than -14024, the program terminates after all available warnings and error messages from the ALLBASE/SQL message catalog have been displayed.

In the case of any other errors, the program displays all available messages, then prompts for another part number.

The code routine located at Label 3000 is executed when only a warning condition results during execution of the SELECT command. This code routine displays a message and the row of data retrieved.

The NOT FOUND condition that may be associated with the SELECT command is handled by the code routine located at Label 4000. This code routine displays the message "Row not found!", then passes control to EndTransaction. SQLEXPLAIN does not provide a message for the NOT FOUND condition, so the program must provide one.

```
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* This program illustrates the use of SQL's SELECT command to      *)
(* retrieve one row or tuple at a time.                             *)
(* Same as pasex2 with added status checking and deadlock routines *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

Program pasex5(input, output);

label
      500,
     1000,
     2000,
     3000,
     4000,
     9999;
const
  OK             =        0;
  NotFound       =      100;
  MultipleRows   = -10002;
  DeadLock       = -14024;
  NoMemory       =  -4008;
  TryLimit       =        3;
var

    EXEC SQL INCLUDE SQLCA;


         (* Begin Host Variable Declarations *)
    EXEC SQL  Begin Declare Section;
    PartNumber       : packed array[1..16] of char;
    PartName         : packed array[1..30] of char;
    SalesPrice       : longreal;
    SalesPriceInd    : SQLIND;
    SQLMessage       : packed array[1..132] of char;
    EXEC SQL  End Declare Section;
         (* End Host Variable Declarations *)

    Abort              : boolean;
    SQLCommandDone     : boolean;
    TryCounter         : integer;

procedure SQLStatusCheck;  (* Procedure to Display Error Messages *)
   Forward;
$PAGE $
```

Figure 5-1. Implicitly Invoking Status-Checking Routines

```
(* Directive to set SQL Whenever error checking *)
EXEC SQL Whenever SqlError GOTO 1000;                                    1

Procedure ConnectDBE;  (* Procedure to Connect to PartsDBE *)
begin

writeln('Connect to PartsDBE');
EXEC SQL CONNECT TO '../sampledb/PartsDBE';

end;  (* End of ConnectDBE Procedure *)

Procedure BeginTransaction;  (* Procedure to Begin Work *)
begin

writeln;
writeln('Begin Work');
EXEC SQL BEGIN WORK;

end;  (* End BeginTransaction Procedure *)

procedure EndTransaction;  (* Procedure to Commit Work *)
begin

writeln;
writeln('Commit Work');
EXEC SQL COMMIT WORK;

end;  (* End EndTransaction Procedure *)

(* Directive to reset SQL Whenever error checking *)
EXEC SQL Whenever SqlError CONTINUE;                                     2

procedure TerminateProgram;   (* Procedure to Release PartsDBE *)
begin

writeln('Release PartsDBE');
EXEC SQL COMMIT WORK RELEASE;

writeln;
writeln('Terminating Program');
Goto 9999;  (* Goto exit point for main program *)

end;  (* End TerminateProgram Procedure *)
$PAGE $
```

Figure 5-1. Implicitly Invoking Status-Checking Routines (page 2 of 6)

```
procedure DisplayRow;   (* Procedure to Display Parts Table Rows *)
begin

writeln;
writeln('Part Number: ', PartNumber);
writeln('Part Name:   ', PartName);
if SalesPriceInd < 0 then
   writeln('Sales Price is NULL')
else
   writeln('Sales Price: ', SalesPrice:10:2);

end;  (* End of DisplayRow *)
$PAGE $
(* Directives to set SQL Whenever error checking *)
EXEC SQL Whenever SqlError   GOTO 2000;
EXEC SQL Whenever SqlWarning GOTO 3000;
EXEC SQL Whenever Not Found  GOTO 4000;
$PAGE $
procedure SelectData; (* Procedure to Query Parts Table *)
begin
repeat
if SQLCommandDone then
  begin

  writeln;
  prompt('Enter Part Number within Parts Table or "/" to STOP> ');
  readln(PartNumber);
  writeln;

  TryCounter := 0;
  end;

if PartNumber[1] <> '/' then
  begin

    BeginTransaction;

    TryCounter := TryCounter + 1;
    writeln('SELECT PartNumber, PartName, SalesPrice');
    EXEC SQL SELECT PartNumber, PartName, SalesPrice
             INTO :PartNumber,
                  :PartName,
                  :SalesPrice  :SalesPriceInd
```

3

4

5

Figure 5-1. Implicitly Invoking Status-Checking Routines (page 3 of 6)

```
                 FROM PurchDB.Parts
                WHERE PartNumber = :PartNumber;
      (* If no errors occur set command done flag and display row *)
      SQLCommandDone := TRUE;
      DisplayRow;

      EndTransaction;

end;  (* End if *)
until PartNumber[1] = '/';

end;      (* End of SelectData Procedure *)
$PAGE $

procedure SQLStatusCheck;  (* Procedure to Display Error Messages *)
begin

if ((SQLCA.SQLCODE = DeadLock) or (SQLCA.SQLCODE = NoMemory) then
  begin
   if TryCounter = TryLimit then
      begin
      SQLCommandDone := TRUE;
      writeln(Could not complete transaction. You may want to try again.');
      end
    else
      SQLCommandDone := FALSE;
  end

else
   begin
   Abort := FALSE;
   if SQLCA.SQLWARN[6] = 'W' then Abort := TRUE;

   repeat
     EXEC SQL SQLEXPLAIN :SQLMessage;
     writeln(SQLMessage);
   until SQLCA.SQLCODE = 0;

   if Abort then TerminateProgram;
   end;

end;  (* End SQLStatusCheck Procedure *)
$PAGE $

begin  (* Beginning of Program *)
```

Figure 5-1. Implicitly Invoking Status-Checking Routines (page 4 of 6)

```
write('Program to SELECT specified rows from ');
writeln('the Parts Table - PASEX5');
writeln;
writeln('Event List:');
writeln('  Connect to PartsDBE');
writeln('  Begin Work');
writeln('  SELECT specified row from Parts Table');
writeln('    until user enters "/" ');
writeln('  Commit Work');
writeln('  Disconnect from PartsDBE');
writeln;

ConnectDBE;

(* Initialize command done flag to true *)

SQLCommandDone := True;
(* Restart/Reentry point for Main Program *)
500:

SelectData;
TerminateProgram;

(* Whenever Routine - Serious DBE Error *)
(* SQL Whenever SQLError Entry Point 1 *)
1000:

  (* Begin *)
  SQLStatusCheck;
  TerminateProgram;
  (* End *)
$PAGE $

(* Whenever Routine - SQL Error *)
(* SQL Whenever SQLError Entry Point 2 *)
2000:

  (* Begin *)
  if SQLCA.SQLCODE = MultipleRows then
     Begin
     writeln;
     writeln('WARNING: More than one row qualifies.');
     EndTransaction;
     SQLCommandDone := True;
     end;
  SQLStatusCheck;
```

Figure 5-1. Implicitly Invoking Status-Checking Routines (page 5 of 6)

```
  Goto 500;   (* Goto Restart/Reentry point of main program *)
  (* End *)

(* Whenever Routine - SQL Warning *)
(* SQL Whenever SQL Warning Entry Point *)
3000:

  (* Begin *)
  writeln('SQL WARNING has occurred. The following row');
  writeln('of data may not be valid.');
  DisplayRow;
  EndTransaction;
  SQLCommandDone := True;
  Goto 500;   (* Goto Restart/Reentry point of main program *)
  (* End *)

(* Whenever Routine - Not Found Error *)
(* SQL Whenever Not Found Entry Point *)
4000:

  (* Begin *)
  writeln;
  writeln('Row not found!');
  EndTransaction;
  SQLCommandDone := True;
  Goto 500;   (* Goto Restart/Reentry point of main program *)
  (* End *)

(* Exit Point for main program *)
9999:

end.   (* End of Program *)
```

Figure 5-1. Implicitly Invoking Status-Checking Routines (page 6 of 6)

Explicit Status Checking Techniques

With explcit error handling, you invoke a function after *explicitly* checking sqlca values rather than using the WHENEVER command. The program in Figure 5-1 has already illustrated several uses of explicit error handling to do the following:

■ Isolate errors so critical that they caused ALLBASE/SQL to roll back the current transaction.

■ Control the number of times SQLEXPLAIN is executed.

■ Detect when more than one row qualifies for a simple SELECT operation.

The example in Figure 5-1 illustrates how implicit routines can sometimes reduce the amount of status checking code. As the number of SQL operations in a program increases, however, the likelihood of needing to return to *different* locations in the program after execution of such a routine increases.

The example shown in Figure 5-2 contains four data manipulation operations: INSERT, UPDATE, DELETE, and SELECT. Each of these operations is executed from its own procedure.

As in the program in Figure 5-1, one procedure is used for explicit status checking: SQLStatusCheck. Unlike the program in Figure 5-1, however, this procedure is invoked after an explicit test of SQLCODE is made immediately following each data manipulation operation.

Because the status checking is included in a procedure rather than a routine following the embedded SQL command, control returns to the point in the program where SQLStatusCheck is invoked.

```
const
  OK            =        0;
  NotFound      =      100;
  MultipleRows  = -10002;
  DeadLock      = -14024;
  NoMemory      =   -4008;
  TryLimit      =        3;
      .
      .
      .
    procedure SelectActivity;
    begin
        This procedure prompts for a number that indicates whether the
        user wants to SELECT, UPDATE, DELETE, or INSERT rows, then invokes
        a procedure that accomplishes the selected activity.  The DONE
        flag is set when the user enters a slash.
    end;
    .
    .
    .
    procedure InsertData;
    begin
      Statements that accept data from the user appear here.

      EXEC SQL INSERT
                INTO PurchDB.Parts (PartNumber,
                                     PartName,
                                     SalesPrice)
                    VALUES (:PartNumber,
                            :PartName,
                            :SalesPrice);

      if SQLCA.SQLCODE <> OK then SQLStatusCheck;
    .
    .
    .
    end;
```

Figure 5-2. Explicitly Invoking Status-Checking Procedure

```
    procedure UpdateData;
    begin
        This procedure verifies that the row(s) to be changed exist, then
        invokes procedure DisplayUpdate to accept new data from the user.

        EXEC SQL SELECT  PartNumber, PartName, SalesPrice
                   INTO :PartNumber,
                        :PartName,
                        :SalesPrice
                   FROM  PurchDB.Parts
                  WHERE  PartNumber = :PartNumber;
        case SQLCA.SQLCODE of
          OK            : begin
                             DisplayUpdate;
                          end;
          NotFound      : begin
                             writeln;
                             writeln('Row not found!');
                          end;
          MultipleRows  : begin
                             writeln;
                             writeln('Row not found!');
                          end;
          otherwise     : begin
                             SQLStatusCheck;
                          end;
        end;
  ⋮
  end;
  ⋮
  procedure DisplayUpdate;
  begin
        Code that prompts the user for new data appears here.
        EXEC SQL UPDATE PurchDB.Parts
                    SET PartName = :PartName,
                        SalesPrice = :SalesPrice,
                  WHERE PartNumber = :PartNumber;
        if SQLCA.SQLCODE <> 0 then SQLStatusCheck;
  .
  .
  .
  end;
```

Figure 5-2. Explicitly Invoking Status-Checking Procedure (page 2 of 5)

```
    procedure DeleteData;
    begin
        This procedure verifies that the row(s) to be deleted exist, then
        invokes procedure DisplayDelete to delete the tow(s)

        EXEC SQL SELECT PartNumber, PartName, SalesPrice
                 INTO :PartNumber,
                      :PartName,
                      :SalesPrice
                  FROM PurchDB.Parts
                 WHERE PartNumber = :PartNumber;


        case SQLCA.SQLCODE of
          OK            : begin
                              DisplayDelete;
                          end;
          NotFound      : begin
                              writeln;
                              writeln('Row not found!');
                          end;
          MultipleRows  : begin
                          writeln;
                          writeln('Row not found!');
                          end;
          OtherWise     : begin
                              SQLStatusCheck;
                          end;
        end;
    ⋮
end;
⋮
procedure DisplayDelete;
begin
    Statements that verify that the deletion should actually occur
    appear here.
        EXEC SQL DELETE FROM PurchDB.Parts
                        WHERE PartNumber = :PartNumber;
        if SQLCA.SQLCODE <> 0 then SQLStatusCheck;
    ⋮



end;
```

Figure 5-2. Explicitly Invoking Status-Checking Procedure (page 3 of 5)

```
      procedure SelectData;
      begin
       Statements that prompt for a partnumber appear here.

        EXEC SQL SELECT PartNumber, PartName, SalesPrice
                  INTO :PartNumber,
                       :PartName,
                       :SalesPrice
                FROM PurchDB.Parts
                WHERE PartNumber = :PartNumber;


        case SQLCA.SQLCODE of
        OK            : begin
                          DisplayRow;
                        end;
        NotFound      : begin
                          writeln;
                          writeln('Row not found!');
                        end;
        MultipleRows  : begin
                          writeln;
                          writeln('Row not found!');
                        end;
        otherwise     : begin
                          SQLStatusCheck;
                        end;
        end;
  .
  .
  .
end;
  .
  .
  .
```

Figure 5-2. Explicitly Invoking Status-Checking Procedure (page 4 of 5)

```
      procedure SQLStatusCheck;
      begin

      if ((SQLCA.SQLCODE = DeadLock) or (SQLCA.SQLCODE = NoMemory) then
         begin
         if TryCounter = TryLimit then
            begin
            SQLCommandDone := TRUE;
            writeln('Transaction incomplete. You may want to try again.');
            end
         else
            SQLCommandDone := FALSE;
         end
      else
         begin
         Abort := FALSE;
         if SQLCA.SQLCODE < DeadLock then Abort := TRUE;

         repeat

            EXEC SQL SQLEXPLAIN :SQLMessage;
            writeln(SQLMessage);

         until SQLCA.SQLCODE = 0;

         if Abort then TerminateProgram;
         end;

      end;    (* End SQLStatusCheck Procedure *)
```

Figure 5-2. Explicitly Invoking Status-Checking Procedure (page 5 of 5)

Handling Deadlock and Shared Memory Problems

A deadlock exists when two transactions need data that the other transaction already has locked. When a deadlock occurs, ALLBASE/SQL rolls back the transaction with the larger priority number. If two deadlocked transactions have the same priority, ALLBASE/SQL rolls back the newer transaction.

An SQLCODE of -14024 indicates that a deadlock has occurred:

```
Deadlock detected.  (DBERR 14024)
```

An SQLCODE of -4008 indicates that ALLBASE/SQL does not have access to the amount of shared memory required to execute a command:

```
ALLBASE/SQL shared memory allocation failed in DBCORE. (DBERR 4008)
```

One way of handling deadlocks and shared memory problems is shown in the previous example, Figure 5-2. A SELECT command is executed, and, if an error occurs, function SQLStatusCheck is executed. If the first error detected was a deadlock or a shared memory problem, the SELECT command is automatically re-executed as many as three times before the user is notified of the situation. If other errors occurred before the deadlock or shared memory problem, the transaction is not automatically re-applied. If an error with an SQLCODE less than -14024 occurred, the program is terminated after the error messages are displayed.

Determining Number of Rows Processed

SQLERRD[3] is useful in the following ways:

- To determine how many rows were processed in one of the following operations when the operation could be executed without error:

  ```
  SELECT
  INSERT
  UPDATE
  DELETE
  ```

  and Cursor operations:

  ```
  FETCH
  UPDATE WHERE CURRENT
  DELETE WHERE CURRENT
  ```

  The SQLERRD[3] value can be used in these cases only when SQLCODE does not contain a negative number. When SQLCODE is 0, SQLERRD[3] is always equal to 1 for SELECT, FETCH, UPDATE WHERE CURRENT, and DELETE WHERE CURRENT operations. SQLERRD[3] may be greater than 1 if more than one row qualifies for an INSERT, UPDATE, or DELETE operation. When SQLCODE is 100, SQLERRD[3] is 0.

- To determine how many rows were processed in one of the BULK operations:

  ```
  BULK SELECT
  BULK FETCH
  BULK INSERT
  ```

In this case, you also need to test SQLCODE to determine whether the operation executed without error. If SQLCODE is negative, SQLERRD[3] contains the number of rows that could be successfully retrieved or inserted before an error occurred. If SQLCODE is 0, SQLERRD[3] contains the total number of rows that ALLBASE/SQL put into or took from the host variable array. If, in a BULK SELECT operation, more rows qualify than the array can accommodate, SQLCODE will be 0.

Examples appear on the following pages.

INSERT, UPDATE, and DELETE Operations. The example in Figure 5-2 could be modified to display the number of rows inserted, updated, or deleted by using SQLERRD[3]. In the case of the update operation, for example, the actual number of rows updated could be displayed after the UPDATE command is executed:

```
   .
   .
   .
procedure DisplayUpdate;
begin

     Code that prompts the user for new data appears here.
     EXEC SQL UPDATE PurchDB.Parts
                 SET PartName = :PartName,
                     SalesPrice = :SalesPrice,
               WHERE PartNumber = :PartNumber;

     case SQLCA.SQLCODE of
       OK        : begin
                     NumberOfRows := SQLERRD[3];
                     writeln('The number of rows updated was: ' NumberOfRows);
                     end;
       OtherWise : begin
                     writeln('No rows could be updated!');
                     SQLStatusCheck;
                     end;
       end;
     until Done;
  end;
```

If the UPDATE command is successfully executed, SQLCODE is 0 and SQLERRD[3] contains the number of rows updated. If the UPDATE command cannot be successfully executed, SQLCODE contains a negative number and SQLERRD[3] contains a 0.

BULK Operations. When using the BULK SELECT, BULK FETCH, or BULK INSERT commands, you can use the SQLERRD[3] value several ways:

- If the command executes without error, to determine the number of rows retrieved into an output host variable array or inserted from an input host variable array.

- If the command causes an error condition, to determine the number of rows that could be successfully put into or taken out of the host variable array before the error occurred.

In the code identified as ① in Figure 5-3, the value in SQLERRD[3] is displayed when only some of the qualifying rows could be retrieved before an error occurred.

In the code identified as ②, the value in SQLERRD[3] is compared with the maximum array size to determine whether more rows might have qualified than the program could display. You could also use a cursor and execute the FETCH command until SQLCODE=100.

In the code identified as ③, the value in SQLERRD[3] is used to control the number of times procedure DisplayRow is executed.

```
const
   OK            =       0;
   NotFound      =     100;
   MaximumRows   =     200;

var
         (*Begin Host Variable Declarations *)
   EXEC SQL Begin Declare Section;
   PartsTable           : packed array[1..200] of
                            packed record
                            PartNumber      : packed array[1..16] of char;
                            PartName        : packed array[1..30] of char;
                            SalesPrice      : longreal;
                          end;
   SQLMessage           : packed array[1..132] of char;
   EXEC SQL End Declare Section;
         (* End Host Variable Declarations *)

   SQLCA : SQLCA_type;    (* SQL Communication Area *)

   I                : integer;
   NumberOfRows : integer;

   procedure BulkSelect;
   begin
     EXEC SQL BULK SELECT PartNumber,
                          PartName,
                          SalesPrice
                   INTO :PartsTable
                   FROM  PurchDB.Parts;

     case SQLCA.SQLCODE of
        OK            : DisplayTable;
        NotFound      : begin
                          writeln;
                          writeln('No rows qualify for this operation!');
                          end;
        OtherWise     : begin
                          NumberOfRows := SQLERRD[3];                    (1)
                          writeln('Only ' NumberOfRows 'rows were retrieved ');
                          writeln('   before an error occurred!');
                          DisplayTable;
                          SQLStatusCheck;
                   end;
        end;
      end;
```

Figure 5-3. Using SQLERRD[3] After a BULK SELECT Operation

```
          .
          .
          .
    procedure DisplayTable;
    begin
      if SQLERRD[3] = MaximumRows then                              ②
      begin
          writeln;
          writeln('WARNING:  There may be additional rows that qualify!');
      end;
      The column headings are displayed here.
      for I := 1 to SQLERRD[3] do
        DisplayRow;                                                 ③
      writeln;
    end;

    procedure DisplayRow;
    begin
      writeln(PartNumber(I), '|');
      writeln(PartName(I),  '|');
      writeln(SalesPrice(I), '|');
    end;
```

Figure 5-3. Using SQLERRD[3] After a BULK SELECT Operation (page 2 of 2)

Detecting End of Scan

Previous examples in this chapter have illustrated how an SQLCODE of 100 can be detected and handled for data manipulation commands that do not use a cursor. When a cursor is being used, this SQLCODE value is used to determine when all rows in an active set have been fetched:

```
procedure FetchRow;
begin
  EXEC SQL FETCH  CURSOR1
            INTO :PartNumber,
                 :PartName,
                 :SalesPrice;

  case SQLCA.SQLCODE of
    OK           : DisplayRow;
    NotFound     : begin
                     DoneFetch := TRUE;
                     writeln;
                     writeln('Row not found or no more rows.');
                   end;
    OtherWise    : begin
                     SQLStatusCheck;
                   end;

  end;
end;
.
.
.
    EXEC SQL OPEN CURSOR1;
.
.
.
    repeat
    FetchRow
    until DoneFetch := TRUE;
```

In this example, the active set is defined when the OPEN command is executed. The cursor is then positioned before the first row of the active set. When the FETCH command is executed, the first row in the active set is placed into the program's host variables, then displayed. The FETCH command retrieves one row at a time into the host variables until the last row in the active set has been retrieved; the next attempt to FETCH after the last row from the active set has been fetched sets SQLCODE to NotFound (defined as 100 in the declaration part). If no rows qualify for the active set, SQLCODE is NotFound the first time procedure FetchRow is executed.

Determining When More Than One Row Qualfies

If more than one row qualifies for a non-BULK SELECT or FETCH operation, ALLBASE/SQL sets SQLCODE to -10002. In the following example, when SQLCODE is MultipleRows (defined as -10002 in the declaration part) a status checking procedure is not invoked, but a warning message is displayed:

```
procedure UpdateData;
begin
    This procedure verifies that the row(s) to be changed exist, then invokes
    procedure DisplayUpdate to accept new data from the user.
    EXEC SQL SELECT  OrderNumber, ItemNumber, OrderQty
             INTO :OrderNumber,
                  :ItemNumber,
                  :OrderQty
             FROM  PurchDB.OrderItems
             WHERE  OrderNumber = :OrderNumber;

    case SQLCA.SQLCODE of
      OK             : DisplayUpdate;
      NotFound       : begin
                         writeln;
                         writeln('Row not found.');
                         end;
      MultipleRows   : begin
                         writeln;
                         writeln('WARNING: More than one row qualifies');
                         DisplayUpdate;
                         end;
      OtherWise      : begin
                         SQLStatusCheck;
                         end;
    end;
end;
```

Detecting Log Full Condition

When the log file is full, log space must be reclaimed before ALLBASE/SQL can process any additional transactions. Your program can detect the situation, and it can be corrected by the DBA.

SQLEXPLAIN retrieves the following message:

```
Log full.  (DBERR 14046)
```

In the following example, SQLCODE is checked for a log full condition. If the condition is true, ALLBASE/SQL has rolled back the current transaction. The program issues a COMMIT WORK command, the SQLStatusCheck function routine is executed to display any error or warning messages, and the program is terminated.

```
if SQLCA.SQLCODE = -14046 then
  begin
  CommitWork;
  SQLStatusCheck;
  TerminateProgram;
  end;
```

Handling Out of Space Conditions

It is possible that data or index space may be exhausted in a DBEFileSet. This could happen as rows are being added or an index is being created or when executing queries which require that data be sorted. Your program can detect the problem, and the DBA must add index or data space to the appropriate DBEFileSet.

SQLEXPLAIN retrieves the following message:

```
Data or Index space exhausted in DBEFileSet.  (DBERR 2502)
```

In the following example, SQLCODE is checked for an out of space condition. If the condition is true, the program rolls back the transaction to an appropriate savepoint. The program issues a COMMIT WORK command, the SQLStatusCheck routine is executed to display any messages, and the program is terminated.

```
if SQLCA.SQLCODE = -2502 then
  begin
  RollbackWork;
  CommitWork;
  TerminateProgram;
  end;
```

Checking for Authorizations

When the DBEUserID related to an ALLBASE/SQL command does not have the authority to execute the command, the following message is retreived by SQLEXPLAIN:

```
User ! does not have ! authorization.   (DBERR 2300)
```

In the following example, SQLCODE is checked to determine if the user has proper connect authority. If the condition is true, the SQLStatusCheck routine is executed to display any messages, and the program is terminated.

```
EXEC SQL CONNECT TO '../sampledb/PartsDBE';

if SQLCA.SQLCODE = -2300 then
begin
SQLStatusCheck;
TerminateProgram;
end;  (* End if *)
```

6

Overview Of Data Manipulation

To manipulate data in an ALLBASE/SQL DBEnvironment, you use one of the following SQL commands:

- SELECT: To retrieve one or more rows from one or more tables.

- INSERT: To insert one or more rows into a single table.

- DELETE: To delete one or more rows from a single table.

- UPDATE: To change the value of one or more columns in one or more rows in a single table.

Four techniques exist for using these commands in a program:

- In simple data manipulation, you retrieve or insert a single row or you delete or update one or more rows based on a specific criterion.

- In sequential table processing, you operate on a set of rows, one row at a time, using a cursor. A cursor is a pointer that identifies one row in the set of rows, called the active set. You move through the active set, retrieving a row at a time and optionally updating or deleting it.

- In BULK table processing, you manipulate multiple rows at a time using a host variable declared as an array. You can retrieve rows from a table into the host variable or insert data from the host variable into rows of a table. A cursor can, but need not, be used for some BULK operations.

- In dynamic operations, you preprocess SQL commands at run time. For example, a program might accept data manipulation commands from a user. A cursor is used to handle dynamic SELECT operations.

Table 6-1 summarizes which data manipulation commands can be used in each technique. Note that the FETCH command is included in this table, since it must be used when you manipulate data using a cursor.

Table 6-1. How Data Manipulation Commands May Be Used

TYPE OF OPERATION	SELECT	FETCH	INSERT	DELETE	UPDATE	DELETE WHERE CURRENT	UPDATE WHERE CURRENT
Simple	X		X	X	X		
Sequential	X	X				X	X
BULK	X	X	X				
Dynamic	X	X	X	X	X		

The remainder of this chapter briefly examines each of the four data manipulation techniques (each technique is discussed in detail in Chapters 7 through 10) and introduces the use of a cursor for data manipulation. First, however, this chapter addresses the query, a description of data you want to retrieve. Queries are fundamental to ALLBASE/SQL data manipulation because some of the elements of a query are also used to describe and limit data when you update or delete it. In addition, it is common programming practice to retrieve and display rows prior to changing or deleting them.

The Query

A query is a SELECT command that describes to ALLBASE/SQL the data you want retrieved. You can retrieve all or only certain data from a table. You can have ALLBASE/SQL group or order the rows you retrieve or perform certain calculations or comparisons before presenting data to your program. You can retrieve data from multiple tables. You can also retrieve data using views or combinations of tables and views.

The SELECT Command

The SELECT command identifies the columns and rows you want in your query result as well as the tables and views to use for data access. The columns are identified in the select list. The rows are identified in several clauses (GROUP BY, HAVING, and ORDER BY). The tables and views to access are identified in the FROM clause. Data thus specified is returned into host variables named in the INTO clause, as the following syntax shows:

```
EXEC SQL     SELECT  SelectList
               INTO  HostVariables
               FROM  TableNames
              WHERE  SearchCondition1
           GROUP BY  ColumnName
             HAVING  SearchCondition2
           ORDER BY  ColumnID;
```

To retrieve all data from a table, the SELECT command need specify only the following:

```
EXEC SQL BULK SELECT  *
                INTO :MyArray
                FROM  PurchDB.Parts;
```

Although the shorthand notation * can be used in the select list to indicate you want all columns from one or more tables or views, it is better programming practice to explicitly name columns. Then, if the tables or views referenced are altered, your program will still retrieve only the data its host variables are designed to accommodate:

```
EXEC SQL BULK SELECT  PartNumber,
                PartName,
                SalesPrice
                INTO :MyArray
                FROM  PurchDB.Parts;
```

The SELECT command has several clauses you can use to format the data retrieved from any table:

- The WHERE clause specifies a search condition. A search condition consists of one or more predicates. A predicate is a test each row must pass before it is returned to your program.

- The GROUP BY clause and the HAVING clause tell how to group rows retrieved before applying any aggregate function in the select list to each group of rows.

- The ORDER BY clause causes ALLBASE/SQL to return rows in ascending or descending order, based on the value in one or more columns.

The following SELECT command contains a WHERE clause that limits rows returned to those not containing a salesprice; the predicate used in the WHERE clause is known as the null predicate:

```
EXEC SQL BULK SELECT  PartName,
                SalesPrice
             INTO :MyArray
             FROM PurchDB.Parts
            WHERE SalesPrice IS NULL;
```

In the UPDATE and DELETE commands, you may need a WHERE clause to limit the rows ALLBASE/SQL changes or deletes. In the following case, the sales price of parts priced lower than $1000 is increased 10 percent; the WHERE clause in this case illustrates the comparison predicate:

```
EXEC SQL UPDATE PurchDB.Parts
           SET SalesPrice = SalesPrice * 1.1
         WHERE SalesPrice < 1000.00;
```

The *ALLBASE/SQL Reference Manual* details the syntax and semantics for these and other predicates.

When you use an aggregate function in the select list, you can use the GROUP BY clause to indicate how ALLBASE/SQL should group rows before applying the function. You can also use the HAVING clause to limit the groups to only those satisfying certain criteria. The following SELECT command will produce a query result containing two columns: a sales price and a number indicating how many parts have that price:

```
EXEC SQL BULK SELECT  SalesPrice,
                      COUNT(PartNumber)
              INTO :MyArray
              FROM  PurchDB.Parts
          GROUP BY  SalesPrice
            HAVING   AVG(SalesPrice) > 1500.00;
```

The GROUP BY clause in this example causes ALLBASE/SQL to group all parts with the same sales price together. The HAVING clause causes ALLBASE/SQL to ignore any group having an average sales price less than or equal to $1500.00. Once the groups have been defined, ALLBASE/SQL applies the aggregate function COUNT to each group.

Null values in a GROUP BY column constitute their own group. Therefore, a query result having a null value in the column used to group rows would contain a separate row for each null value.

An aggregate function is one example of an ALLBASE/SQL expression. An expression specifies a value. An expression can be used in several places in the SELECT command as well as in the other data manipulation commands. Refer to the *ALLBASE/SQL Reference Manual* for the syntax and semantics of expressions, as well as the effect of null values on them.

The rows in the query result obtained with the preceding query could be returned in a specific order by using the ORDER BY clause. In the following case, the rows are returned in descending sales price order:

```
EXEC SQL BULK SELECT  SalesPrice,
                      COUNT(PartNumber)
              INTO :MyArray
              FROM  PurchDB.Parts
          GROUP BY  SalesPrice
            HAVING  AVG(SalesPrice) > 1500.00
          ORDER BY   SalesPrice DESC;
```

The examples shown so far have all included the BULK option and a host variable array, because the query results would most likely contain more than one row. Besides the BULK table processing technique, the sequential table processing technique could also be used to handle multiple-row query results. Later in this chapter you'll find examples of both these techniques, as well as examples illustrating simple data manipulation, in which only one row query results are expected.

Selecting from Multiple Tables

To retrieve data from more than one table or view, the query describes to ALLBASE/SQL how to join the tables before deriving the query result in the following places:

- In the FROM clause, you identify the tables and views to be joined.

- In the WHERE clause, you specify a join condition. A join condition defines the condition(s) under which rows should be joined.
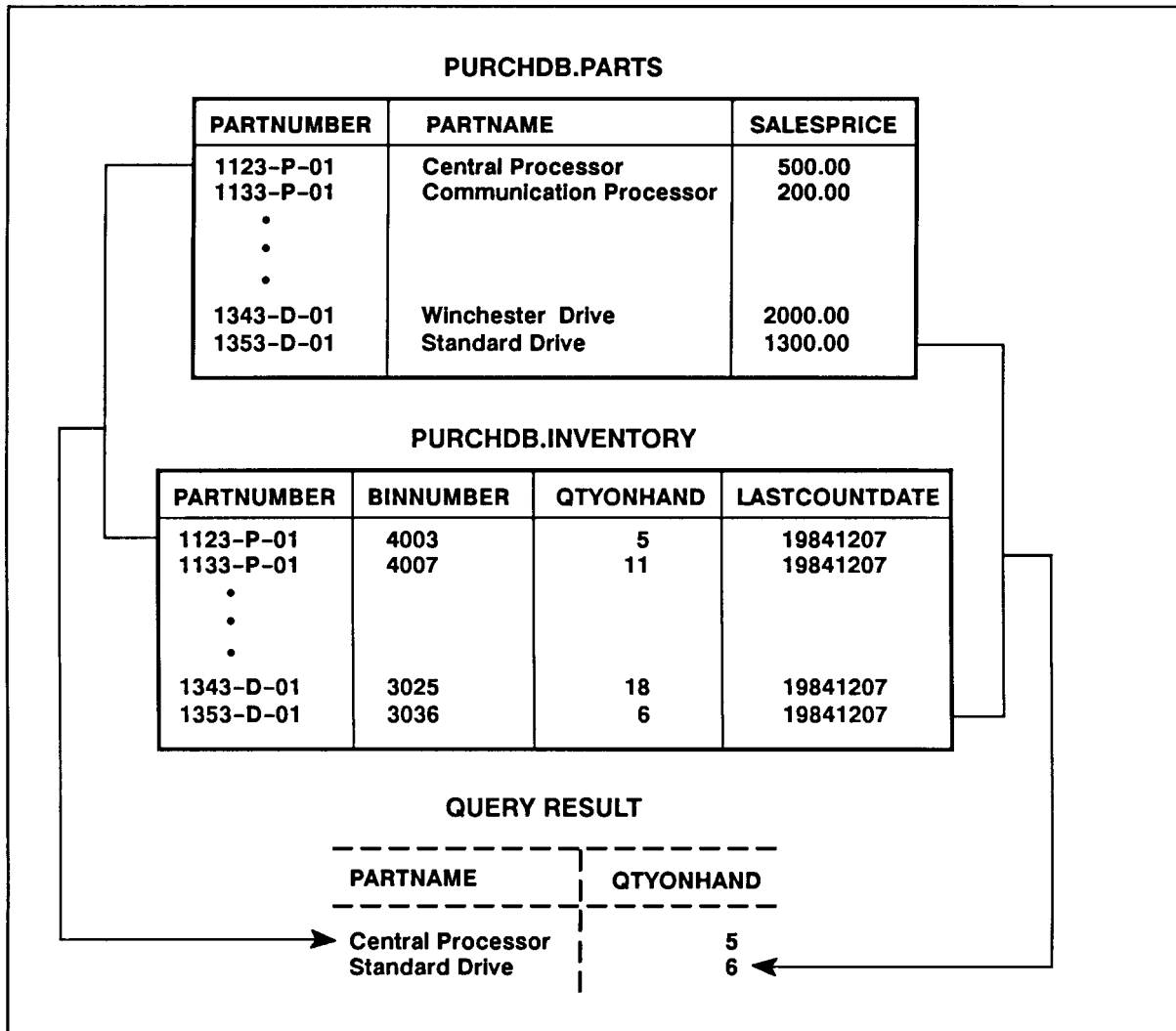
To obtain a query result consisting of the name of each part and its quantity-on-hand, you need data from two tables in the sample database: PurchDB.Parts and PurchDB.Inventory. The join condition in this case is that you want ALLBASE/SQL to join rows in these tables that have the same part number:

```
EXEC SQL BULK SELECT  PartName,
                      QtyOnHand
               INTO  :MyArray
               FROM  PurchDB.Parts,
                     PurchDB.Inventory
              WHERE  PurchDB.Parts.PartNumber =
                     PurchDB.Inventory.PartNumber;
```

Whenever two or more columns in a query have the same name but belong to different tables, you avoid ambiguity by qualifying the column names with table names. Because the columns specified in the join condition shown above have the same name (PartNumber) in both tables, they are fully qualified with table names (PurchDB.Parts and PurchDB.Inventory). If one of the columns named PartNumber were named PartNum, the WHERE clause could be written as follows:

```
WHERE PartNumber = PartNum
```

ALLBASE/SQL creates a row for the query result whenever the PartNumber value in one table matches that in the second table. As illustrated in Figure 6-1, any row containing a null PartNumber is excluded from the join, as are rows that have a PartNumber value in one table, but not the other.

PURCHDB.PARTS

PARTNUMBER	PARTNAME	SALESPRICE
1123-P-01	Central Processor	500.00
1133-P-01	Communication Processor	200.00
•		
•		
•		
1343-D-01	Winchester Drive	2000.00
1353-D-01	Standard Drive	1300.00

PURCHDB.INVENTORY

PARTNUMBER	BINNUMBER	QTYONHAND	LASTCOUNTDATE
1123-P-01	4003	5	19841207
1133-P-01	4007	11	19841207
•			
•			
•			
1343-D-01	3025	18	19841207
1353-D-01	3036	6	19841207

QUERY RESULT

PARTNAME	QTYONHAND
Central Processor	5
Standard Drive	6

LG200125_009a

Figure 6-1. Sample Query Joining Multiple Tables

You can also join a table to itself. This type of join is useful when you want to identify values within one table that have certain relationships.

The PurchDB.SupplyPrice table contains the unit price, delivery time, and other data for every vendor that supplies any part. Most parts are supplied by more than one vendor, and prices vary with vendor. You can join the PurchDB.SupplyPrice table to itself in order to identify for which parts the difference among vendor prices is greater than $50. The query and its result would appear as follows:

```
EXEC SQL BULK SELECT  X.PartNumber,
                      X.VendorNumber,
                      X.UnitPrice,
                      Y.VendorNumber,
                      Y.UnitPrice
               INTO :MyArray
               FROM  PurchDB.SupplyPrice X,
                     PurchDB.SupplyPrice Y
              WHERE  X.PartNumber = Y.PartNumber AND
                     X.UnitPrice  > (Y.UnitPrice + 50.00);
```

PARTNUMBER	VENDORNUMBER	UNITPRICE	VENDORNUMBER	UNITPRICE
1123-P-01	9007	550.00	9002	450.00
1123-P-01	9012	525.00	9002	450.00
1123-P-01	9007	550.00	9008	475.00
1123-P-01	9007	550.00	9003	475.00
1433-M-01	9007	700.00	9003	645.00
1623-TD-01	9011	1800.00	9015	1650.00

These vendors charge at least $50 more for a part than the vendors identified in the next two columns.

To obtain such a query result, ALLBASE/SQL joins one copy of the table with another copy of the table, using the join condition specified in the WHERE clause:

■ You name each copy of the table in the FROM clause by using a correlation name. In this example, the correlations names are X and Y. Then you use the correlation name to qualify column names in the select list and other clauses in the query.

■ The join condition in this example specifies that for each part number, the query result should contain a row only when the price of the part from vendor to vendor differs by more than $50.

Join variables can be used in any query as a shorthand way of referring to a table, but they must be used in queries that join a table to itself so that ALLBASE/SQL can distinguish between the two copies of the table.

Selecting Using Views

Views are used to restrict data visibility as well as to simplify data access:

■ Data visibility can be limited using views by defining them such that only certain columns and/or rows are accessible through them.

■ Data access can be simplified using views by creating views based on joins or containing columns that are derived from expressions or aggregate functions.

The sample database has a view called PurchDB.VendorStatistics, defined as follows:

```
CREATE VIEW   PurchDB.VendorStatistics
              (VendorNumber,
               VendorName,
               OrderDate,
               OrderQuantity,
               TotalPrice)
         AS
     SELECT   PurchDB.Vendors.VendorNumber,
              PurchDB.Vendors.VendorName,
              OrderDate,
              OrderQty,
              OrderQty * PurchasePrice
       FROM   PurchDB.Vendors,
              PurchDB.Orders,
              PurchDB.OrderItems
      WHERE   PurchDB.Vendors.VendorNumber    =
              PurchDB.Orders.VendorNumber
        AND   PurchDB.OrderItems.OrderNumber  =
              PurchDB.OrderItems.OrderNumber
```

This view combines information from three base tables to provide a summary of data on existing orders with each vendor. One of the columns in the view consists of a computed expression: the total cost of an item on order with the vendor.

Note that the select list of the SELECT command defining this view contains some qualified and some unqualified column names. Columns OrderDate, OrderQty, and PurchasePrice need not be qualified, because these names are unique among the column names in the three tables joined in this view. In the WHERE clause, however, both join conditions must contain fully qualified column names, since the columns are named the same in each of the joined tables.

You can use a view in a query without restriction. In the FROM clause, you identify the view as you would identify a table. When you reference columns belonging to the view, you use the column names used in the view definition. In the view above, for example, the column containing quantity-on-order is called OrderQuantity, not OrderQty as it is in the base table (PurchDB.OrderItems).

The VendorStatistics view can be used to quickly determine the total dollar amount of orders existing for each vendor. Because the view definition contains all the details for deriving this information, the query based on this view is quite simple:

```
EXEC SQL BULK SELECT  VendorNumber,
                      SUM(TotalPrice)
                INTO :MyArray
                FROM  PurchDB.VendorStatistics
            GROUP BY  VendorNumber;
```

The query result appears as follows:

```
------------+----------------------
VENDORNUMBER|(EXPR)
------------+----------------------
        9001|            31300.00
        9002|             6555.00
        9003|             6325.00
        9004|             2850.00
        9006|             2010.00
        9008|            12460.00
        9009|             7750.00
        9010|             9180.00
        9012|            12280.00
        9013|             8270.00
        9014|             2000.00
        9015|            17550.00
```

Although you can use views in queries without restriction, you can use only some views to insert, update, or delete rows:

■ You cannot INSERT, UPDATE, or DELETE using a view if the view definition contains one of the following:

   □ Join operation.
   □ Aggregate function.
   □ DISTINCT option.
   □ GROUP BY clause.
   □ ORDER BY clause.
   □ UNION.

■ You cannot INSERT using a view if any column of the view is computed in an arithmetic expression.

The PurchDB.VendorStatistics view cannot be used for any INSERT, UPDATE, or DELETE operation because it is based on a three table join and contains a column (TotalPrice) derived from a multiplication operation.

Simple Data Manipulation

In simple data manipulation, you retrieve or insert single rows or update one or more rows based on a specific criterion. In most cases, the simple data manipulation technique is used to support the random retrieval and/or change of specific rows.

In the following example, if the user wants to perform a DELETE operation, the program performs the operation only if a single row qualifies. If no rows qualify or if more than one row qualifies, the program displays a message. Note that the host variables in this case are designed to accommodate only a single row. In addition, two of the columns may contain null values, so an indicator variable is used for these columns:

```
var
    EXEC SQL BEGIN DECLARE SECTION;
    PartNumber        : packed array[1..16] of char;
    PartName          : packed array[1..30] of char;
    PartNameInd       : sqlInd;
    SalesPrice        : longreal;
    SalesPriceInd     : sqlInd;
    EXEC SQL END DECLARE SECTION;
    .
    .
    .
procedure DoQuery;
begin

    This procedure accepts a part number from the user,
    then executes a query to determine whether one or
    more rows containing that value actually exists.

    EXEC SQL SELECT  PartNumber, PartName, SalesPrice
               INTO :PartNumber,
                    :PartName :PartNameInd,
                    :SalesPrice :SalesPriceInd
               FROM  PurchDB.Parts
              WHERE  PartNumber = :PartNumber;

    case SQLCA.SQLCODE of
    0          : DisplayDelete;
    100        : writeln('Row not found!');
    -10002     : writeln('WARNING:  More than one row qualifies!');
    otherwise  : SqlStatusCheck;
    end;
    .
    .
    .
```

```
procedure DisplayDelete;
```

*The qualifying row is displayed for the user to
verify that it should be deleted before the following
command is executed:*

```
EXEC SQL DELETE FROM PurchDB.Parts
              WHERE PartNumber = :PartNumber;
```

The chapter, "Simple Data Manipulation," provides more details about simple data
manipulation.

Introducing the Cursor

You use a cursor to manage a query result that may contain more than one row when you
want to make all the qualifying rows available to the program user. Cursors are used in
sequential table processing, BULK table processing, and dynamic SELECT operations, as
shown later in this chapter.

Like the cursor on a terminal screen, an ALLBASE/SQL cursor is a position indicator. It does
not, however, point to a column. Rather, it points to one row in an active set. An active set
is a query result obtained when a SELECT command associated with a cursor (defined in a
DECLARE CURSOR command) is executed (using the OPEN CURSOR command).

Each cursor used in a program must be declared before it is used. You use the DECLARE
CURSOR command to declare a cursor. The DECLARE CURSOR command names the
cursor and associates it with a particular SELECT command:

```
EXEC SQL DECLARE  Cursor1
         CURSOR FOR
         SELECT  PartName,
                 SalesPrice
          FROM PurchDB.Parts
          WHERE PartNumber BETWEEN :LowValue AND :HighValue
       ORDER BY PartName;
```

All cursor names within one program must be unique. You use a cursor name when you
perform data manipulation operations using the cursor.

Any reference to a cursor must be within a preprocessed unit, that is, a preprocessed file.
For example, you cannot have a DECLARE CURSOR statement in a main program and
open the cursor in a subprogram or the reverse of this rule, unless the main program and the
subprogram are in the same file.

300/400   Series 300 and Series 400 systems do not allow subprograms.

The SELECT command in the cursor declaration does not specify any output host variables.
The SELECT command can, however, contain input host variables, as in the WHERE clause
of the cursor declaration above.

Rows in the active set are returned to output host variables when the FETCH command is executed:

```
EXEC SQL OPEN Cursor1;
.
.           The OPEN command allocates internal
.           buffer space for the active set
.
.
EXEC SQL [BULK] FETCH Cursor1 INTO OutputHostVariables;

            The FETCH command delivers one row or
            (if the BULK option is used)
            multiple rows of the active set into output
            host variables
```

If a serial scan will be used to retrieve the active set, ALLBASE/SQL locks the table(s) when the OPEN command is executed. If an index scan will be used, locks are placed when rows are fetched.

Both the OPEN and the FETCH commands position the cursor, as follows:

- The OPEN command positions the cursor before the first row of the active set.

- The effect of the FETCH command on the cursor depends on whether the BULK option is used.

  If the BULK option is not used, the FETCH command advances the cursor to the next row of the active set and delivers that row to the output host variables.

  If the BULK option is used, the FETCH command delivers as many rows as the output host variables (declared as an array) can accommodate and advances the cursor to the last row delivered.

The row at which the cursor points at any one time is called the current row. When a row is a current row, you can delete it as follows:

```
EXEC SQL DELETE FROM PurchDB.Parts
                WHERE CURRENT OF Cursor1;
```

When you delete the current row, the cursor remains between the row deleted and the next row in the active set until you execute the FETCH command again:

```
EXEC SQL FETCH  Cursor1
         INTO :PartName :PartNameInd,
              :SalesPrice :SalesPriceInd;
```

When a row is a current row you can update it if the cursor declaration contains a FOR UPDATE OF clause naming the column(s) you want to change. The following cursor, for example, can be used to update the SalesPrice column of the current row by using the WHERE CURRENT OF option in the UPDATE command:

```
EXEC SQL DECLARE Cursor2
          CURSOR FOR
          SELECT PartName, SalesPrice
            FROM PurchDB.Parts
            WHERE PartNumber BETWEEN :LowValue AND :HighValue
        FOR UPDATE OF SalesPrice;
.
.        Because the DECLARE CURSOR command is not
.        executed at run time, no status checking
.        code needs to appear here.
.
EXEC SQL OPEN Cursor2;
.
.        The program fetches and displays one row at a time.
.        The OPEN command allocates internal
.        buffer space for the active set,
.        but no rows are fetched until the FETCH
.        command is executed.
.
EXEC SQL FETCH  Cursor2
          INTO :PartName :PartNameInd,
               :SalesPrice :SalesPriceInd;
.
.        If the program user wants to change the SalesPrice
.        of the row displayed (the current row), the UPDATE
.        command is executed.  The new SalesPrice entered by
.        the user is stored in an input host variable named
.        NewSalesPrice.
.
EXEC SQL UPDATE PurchDB.Parts
            SET SalesPrice = :NewSalesPrice
          WHERE CURRENT OF Cursor2;

        After the UPDATE command is executed, the updated
        row remains the current row until the FETCH command
        is executed again.
```

The restrictions that govern deletions and updates using a view also govern deletions and updates using a cursor. You cannot delete or update a row using a cursor if the cursor declaration contains any of the following:

- Join operation.

- Aggregate function.

- DISTINCT option.

- GROUP BY clause.

- UNION statement.

- ORDER BY statement.

After the last row in the active set has been fetched, the cursor is positioned after the last row fetched and the value in SQLCODE is equal to 100. Therefore, to retrieve all rows in the active set, you execute the FETCH command until SQLCA.SQLCODE is not 0:

```
while SQLCA.SQLCODE = 0 do
  begin
  EXEC SQL FETCH  Cursor3
           INTO :PartNumber,
                :PartName :PartNameInd,
                :SalesPrice :SalesPriceInd;

  case SQLCA.SQLCODE of
  0         : DisplayRow;
  100       : writeln('Row not found or no more rows!');
  otherwise : SqlStatusCheck;
  end;
end;
```

When you are finished operating on an active set, you use the CLOSE command:

```
EXEC SQL CLOSE Cursor3;
```

When you close a cursor, the active set becomes undefined and you cannot use the cursor again unless you issue an OPEN command to reopen it. The COMMIT WORK and ROLLBACK WORK commands also close any open cursors, automatically.

Figure 6-2 summarizes the effect of the cursor related commands on the position of the cursor and on the active set. All the commands shown, plus the DECLARE CURSOR command, must be included within one preprocessed unit (main program or subprogram).

300/400   Series 300 and Series 400 systems do not allow subprograms.
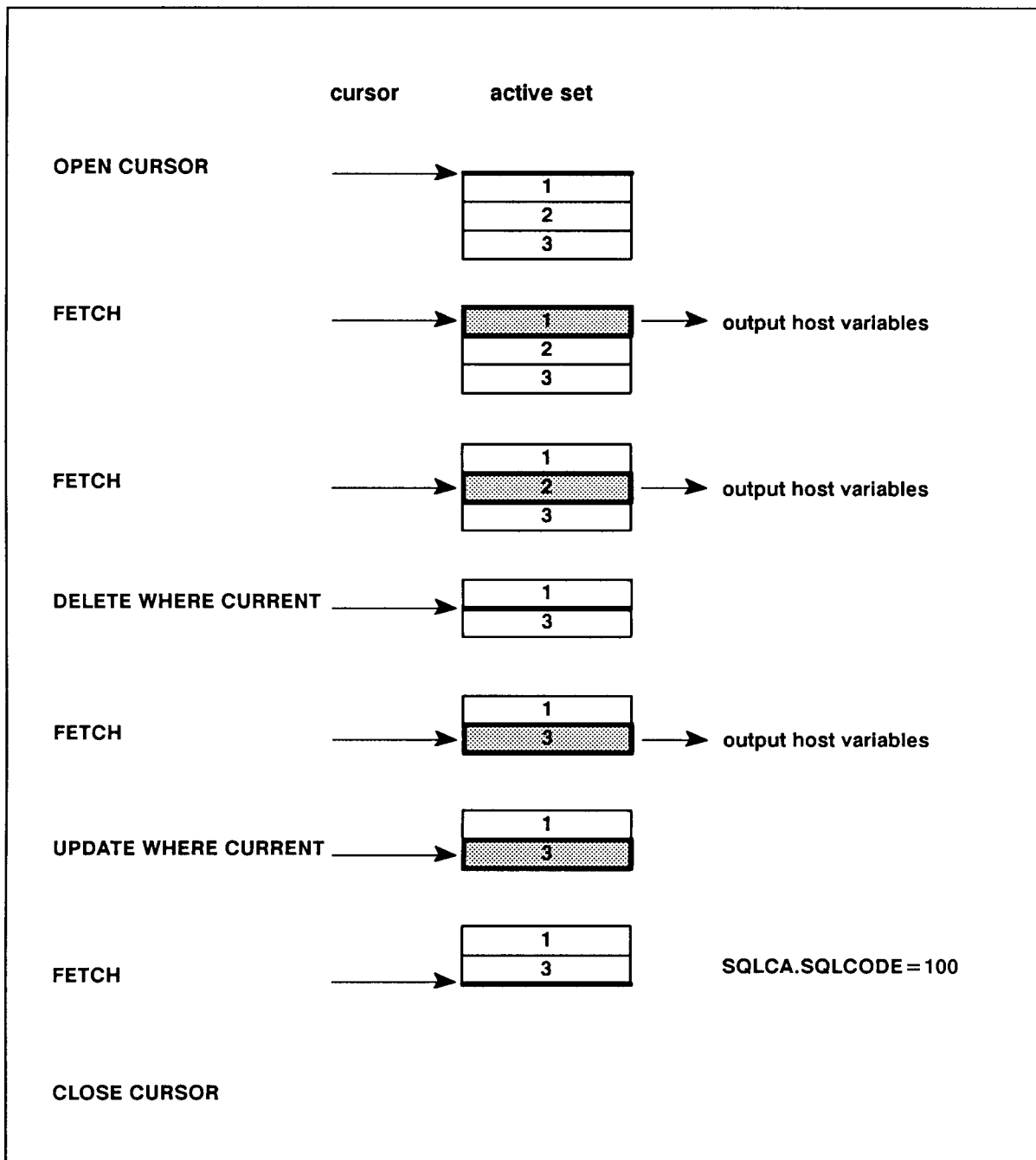
Figure 6-2. Effect of SQL Commands on Cursor and Active Sets

LG200125_010a

Sequential Table Processing

In sequential table processing, you process an active set by fetching a row at a time and optionally deleting or updating it. Sequential table processing is useful when the likelihood of row changes throughout a set of rows is high and when a program user does not need to review multiple rows to decide whether to change a specific row.

In the following example, rows for parts having the same SalesPrice are displayed one at a time. The program user can delete a displayed row or change its SalesPrice. Note that the host variable declarations are identical to those for the simple data manipulation example, since only one row at a time is fetched. Rows are fetched as long as SQLCODE is 0, as shown in the following example:

```
const
     OK         =    0;
     NotFound   = 100;
var
     EXEC SQL BEGIN DECLARE SECTION;
     PartNumber        : packed array[1..16] of char;
     PartName          : packed array[1..30] of char;
     PartNameInd       : sqlInd;
     SalesPrice        : longreal;
     SalesPriceInd     : sqlInd;
     EXEC SQL END DECLARE SECTION;
  .
  .
procedure GetActiveSet;
begin
   The cursor declared allows the user to change the SalesPrice of
   the current row.  It can also be used to delete the current row.

   EXEC SQL DECLARE PriceCursor
            CURSOR FOR
            SELECT PartNumber, PartName, SalesPrice
              FROM PurchDB.Parts
             WHERE SalesPrice = :SalesPrice
      FOR UPDATE OF SalesPrice;
  .
  .        The program accepts a salesprice value from the user.
  .
   EXEC SQL OPEN PriceCursor;

   if SQLCA.SQLCODE <> OK then
     begin
     SqlStatusCheck;
     ReleaseDBE;
     end
   else
     GetRow;
end;
```

```
procedure GetRow;
begin

   while SQLCA.SQLCODE = OK do
   begin

     EXEC SQL FETCH  PriceCursor
              INTO :PartNumber,
                   :PartName :PartNameInd,
                   :SalesPrice :SalesPriceInd;

     case SQLCA.SQLCODE of
     OK          : DisplayRow
     NotFound    : writeln('No more rows!');
     otherwise   : SqlStatusCheck;
     end;
end;
procedure DisplayRow;
begin
```

Each row fetched is displayed. Depending on the user's response to a program prompt, the row may be deleted or its SalesPrice value changed.

```
   if response[1] = 'D' then
     begin
     EXEC SQL DELETE FROM PurchDB.Parts
              WHERE CURRENT OF PriceCursor;
     .
     .   Status checking code appears here.
     .
     end;
   if response[1] = 'U' then
     begin
     .
     .   A new SalesPrice is accepted.
     .
     EXEC SQL UPDATE PurchDB.Parts
                SET SalesPrice = :SalesPrice
              WHERE CURRENT OF PriceCursor;
     .
     .   Status checking code appears here.
     .
     end;
```

Sequential table processing is discussed in more detail in the chapter, "Processing with Cursors."

Bulk Table Processing

BULK table processing offers a way to retrieve or insert multiple rows with the execution of a single SQL command. Three commands can be used in this fashion:

■ You can use the BULK SELECT command when you know in advance the maximum number of rows in a multiple-row query result, as when the query result will contain a row for each month of the year or day of the week. This command minimizes the time a table is locked for the retrieval operation, because the program can execute the BULK SELECT command, then immediately terminate the transaction, even before displaying any rows.

■ You can use the BULK FETCH command to handle multiple-row query results of unpredictable maximum length. This use of a cursor is most suitable for display only applications, such as programs that let a user browse through a query result, so many rows at a time.

■ You can use the BULK INSERT command to insert multiple rows into a table. Like the BULK SELECT command, this command is efficient for concurrency, because any exclusive lock acquired to insert rows need be held only until the BULK INSERT command is executed.

In each of these three commands, the host variables that hold rows are in an array, as illustrated in the following example. The example shows how you can use a cursor to retrieve and display ten rows at a time from the active set. The host variable named StartIndex is set to 1 so that the first row in each group of rows fetched is stored in the first element of the PartsTable array. The host variable named NumberOfRows controls the maximum number of rows returned with each execution of the BULK FETCH command. StartIndex and NumberOfRows are set before the first BULK FETCH is executed.

```
    const
        OK            =   0;
        NotFound      = 100;
        MaximumRows   =  10;
    var
        EXEC SQL BEGIN DECLARE SECTION;
        PartsTable : packed array[1..10] of
                        packed record
                            PartNumber  : packed array[1..16] of char;
                            PartName    : packed array[1..30] of char;
                            PartNameInd : sqlInd;
        StartIndex   : integer;
        NumberOfRows : integer;
        EXEC SQL END DECLARE SECTION;
      .
      .
      .
    procedure DeclareCursor;
    begin
```

```
      EXEC SQL DECLARE PartsCursor
               CURSOR FOR
               SELECT PartNumber, PartName
                 FROM PurchDB.Parts;
    ⋮
   function OpenCursor:  boolean;
   begin
     EXEC SQL OPEN PartsCursor;
     if SQLCA.SQLCODE <> OK then
       begin
       OpenCursor := FALSE;
       SqlStatusCheck;
       ReleaseDBE;
       end
     else
       OpenCursor := TRUE;
     end;
    ⋮
   procedure GetRows;
   begin
     if OpenCursor then
     begin
         StartIndex   := 1;
         NumberOfRows := MaximumRows;
         while SQLCA.SQLCODE = OK do
           begin
             EXEC SQL BULK FETCH  PartsCursor
                            INTO :PartsTable,
                                 :StartIndex,
                                 :NumberOfRows;
```

As many as ten rows are put into the PartsTable
array. If the FETCH command executes without
error the value in SQLERRD(3) indicates the
number of rows returned to PartsTable.

```
             case SQLCA.SQLCODE of
             OK         : for i := 1 to sqlca.sqlerrd(3) do
                          writeln(Partnumber,'|',PartName);
             NotFound   : writeln ('No more rows qualify!');
             otherwise  : SqlStatusCheck;
             end; (*case*)
           end; (*do*)
     end; (*if*)
```

BULK table processing is discussed in additional detail in the chapter, "Bulk Table
Processing."

Dynamic Operations

Dynamic operations offer a way to execute SQL commands that cannot be completely defined until run time. You accept part or all of an SQL command that can be dynamically preprocessed from the user, then use one of the following techniques to preprocess and execute the command:

■ If the dynamic command is not a query, you can use the PREPARE command to preprocess it, then execute it later during the same transaction using the EXECUTE command. Alternatively, you can use the EXECUTE IMMEDIATE command to preprocess and execute the dynamic command in one step.

■ If the dynamic command is a query, you use special data structures plus the following SQL commands to handle the dynamic command: PREPARE, DESCRIBE, DECLARE CURSOR, OPEN, FETCH, and CLOSE.

To determine whether a dynamic command is a query, you use the DESCRIBE command after preparing the dynamic command:

> *The program accepts an SQL command from the user and stores*
> *it in a host variable named DynamicCommand.*

```
EXEC SQL PREPARE DynamCommand FROM :DynamicCommand;
```

> *After the command is prepared, the DESCRIBE command is used*
> *to determine whether the prepared command is a SELECT command.*

```
EXEC SQL DESCRIBE DynamCommand INTO SQLDA;
```

> *The SQLDA.SQLD field of the SQLDA is examined to determine*
> *whether the dynamic command is a query.*

```
if SQLDA.SQLD = 0 then        (*the command is not a query*)
  begin
  EXEC SQL EXECUTE DynamCommand;
  .
  .
  .
else if SQLDA.SQLD > 0 then  (*the command is a query*)
  .
  .
  .
```

> *A non-query can be executed with the EXECUTE*
> *command, but a query requires special handling.*

Dynamic queries require special handling because you may not know in advance what a query result will look like. The number and type of columns, the existence of null values, the column names ... this and other information may need to be obtained by the program at run time because it is not known at programming time.

To obtain the information needed to parse any query result at run time, you use three special data structures in your program: the SQLDA, a format array, and a data buffer:

■ The SQLDA (SQL Description Area) and the format array contain information about the query result. These data structures have the format described in the chapter, "Host Variables."

■ The data buffer holds one or more rows fetched from the query result. Its format is also described in the chapter, "Host Variables."

The following example summarizes how you declare and use these data structures:

```
const
    OK              =     0;
    Notfound        =   100;
    NbrFmtRecords   =   255; (*maximum number of columns*)
    MaxDataBuff     = 1600; (*maximum number of bytes the program*)
                            (*allows in each fetch; 1600 bytes *)
                            (*accommodates the display of 20*)
                            (*80-character lines*)


var
    EXEC SQL BEGIN DECLARE SECTION;
    DynamicCommand          : String[1024] (*maximum length of*)
                                           (*a dynamic command*)
    EXEC SQL END DECLARE SECTION;

    EXEC SQL INCLUDE SQLDA;
                    (*declaration of SQLDA*)

    SQLFmts      : array[1..NbrFmtRecords] of SqlFormat_Type;
                   (*declaration of format array*)

    DataBuffer : packed array[1..MaxDataBuff] of char;
                 (*declaration of data buffer*)
⋮
procedure ExecuteDynamCmd;
begin
  with SQLDA do  (*You must set two SQLDA fields before DESCRIBE*)
    begin
    Sqln       := NbrFmtRecords; (*number of records in format array*)
    SqlFmtArr := waddress(SQLFmts); (*address of format array*)
    end;
  .
  .           The program accepts a dynamic command.
  EXEC SQL PREPARE DYNAMCOMMAND FROM :DYNAMICCOMMAND;
  .
  .           Status checking is done.
  EXEC SQL DESCRIBE DYNAMCOMMAND INTO SQLDA;
```

```
    if SQLDA.SQLD = 0              (*dynamic command is not a query*)
    .
    .      The program executes the dynamic command.
    .
    else if SQLDA.SQLD > 0 then    (*dynamic command is a query*)
    begin
      EXEC SQL DECLARE DYNAMCURSOR CURSOR FOR DYNAMCOMMAND;
      EXEC SQL OPEN DYNAMCURSOR;

      .
      .      If SQLCODE is 0, rows in the query result can be
             fetched.  First, however, you must set three
             SQLDA fields.
      with SQLDA do
      begin
        SqlBufLen := sizeof(DataBuffer); (*bytes in data buffer*)
        SqlNRow   := SqlBufLen DIV SqlRowLen;
                    (*number of rows to fetch into data buffer*)
        SqlRowBuf := waddress(DataBuffer); (*address of data buffer*)
      end;

      while SQLCA.SQLCODE = OK do
      begin
        EXEC SQL FETCH DYNAMCURSOR USING DESCRIPTOR SQLDA;
        if SQLCA.SQLCODE <> OK then
        begin
          if SQLCA.SQLCODE = NotFound then
             writeln('No more rows qualify.')

          else
          SQLStatusCheck;
        end;
        else
        DisplaySelect;
      end;
  .
  .
procedure DisplaySelect;
```

> *This procedure parses the data buffer and displays*
> *rows fetched. For each column in each row of the*
> *query result, various fields in the format array*
> *are used to identify where data values are located*
> *in the data buffer as well as the data type of*
> *these values and null value information.*

Techniques for parsing the data buffer and more examples of dynamic preprocessing are in the chapter, "Using Dynamic Operations."

7

Simple Data Manipulation

Simple data manipulation is a programming technique used to SELECT or INSERT a *single* row. It can also be used to INSERT, DELETE, or UPDATE one or more rows based on a *specific* criterion. These types of data manipulation operations are considered *simple* because they can be done with SQL data manipulation commands that:

- Do not contain the BULK option; therefore the host variables used are not arrays, and data references are simplified.

- Are not executed in conjunction with a cursor; therefore additional SQL commands such as FETCH and OPEN are not required.

- Are not dynamically preprocessed; and therefore, additional arrays and SQL commands are not required to execute them.

This chapter reviews how to use the SELECT, INSERT, DELETE, and UPDATE commands for simple data manipulation. It then briefly examines transaction management considerations. For further discussion of transaction management, refer to the *ALLBASE/SQL Reference Manual* .

A program illustrating simple data manipulation is found at the end of the chapter.

SQL Commands

The SQL commands used for simple data manipulation are:

```
SELECT
INSERT
DELETE
UPDATE
```

Refer to the *ALLBASE/SQL Reference Manual* for the complete syntax and semantics of these commands.

The SELECT Command

In simple data manipulation, you use the SELECT command to retrieve a single row, i.e., a one-row query result. The syntax of the SELECT command that describes a one-row query result is:

```
SELECT   SelectList
  INTO   HostVariables
  FROM   TableNames
 WHERE   SearchCondition
```

Note that the GROUP BY, HAVING, and ORDER BY clauses are not necessary, since these clauses usually describe multiple-row query results.

You may omit the WHERE clause from certain queries when the select list contains *only* aggregate functions:

```
EXEC SQL SELECT  AVG(SalesPrice)
           INTO :AvgSalesPrice
           FROM  PurchDB.Parts;
```

A WHERE clause may be used, however, to qualify the rows over which the aggregate function is applied:

```
EXEC SQL SELECT  AVG(SalesPrice)
           INTO :AvgSalesPrice
           FROM  PurchDB.Parts
           WHERE  SalesPrice > :SalesPrice;
```

If the select list does *not* contain aggregate functions, a WHERE clause is used to restrict the query result to a single row:

```
EXEC SQL SELECT  PartName,  SalesPrice
           INTO :PartName, :SalesPrice
           FROM  PurchDB.Parts
           WHERE  PartNumber = :PartNumber;
```

Because the host variables that hold query results for a simple SELECT command are not arrays of records, they can hold only a single row. A runtime error occurs when multiple rows qualify for a simple SELECT command. You can test for an SQLCODE value of -10002 to detect this condition:

```
const
    MultipleRows = -10002;
 .
 .
 .
procedure GetRow;
 .
 .
 .
        The SELECT command is executed here.


        if SQLCA.SQLCODE = MultipleRows  then
           writeln('WARNING:  More than one row qualifies.');
```

When multiple rows qualify but the receiving host variables are not in an array of records and the BULK option is not specified, *none* of the rows are returned.

When a column named in the WHERE clause has a unique index on it, you can omit testing for multiple-row query results if the column was defined NOT NULL. A unique index prevents the key column(s) from having duplicate values. The following index, for example, ensures that only one row will exist for any part number in PurchDB.Parts:

```
CREATE UNIQUE INDEX PartNumIndex
       ON PurchDB.Parts (PartNumber)
```

If a key column of a unique index *can* contain a null value, the unique index ensures that no more than one null value can exist for that column.

Another method of qualifying the rows you want to select is to use the LIKE specification to search for a particular character string pattern.

For example, suppose you want to search for all VendorRemarks that contain a reference to 6%. Since the percent sign (%) happens to be one of the wildcard characters for the LIKE specification, you could use the following SELECT statement specifying the exclamation point (!) as your escape character.

```
SELECT * FROM PurchDB.Vendors
  WHERE VendorRemarks LIKE '%6!%%' ESCAPE '!'
```

The first and last percent sign characters are the wildcard characters. The next to the last percent sign, preceded by an exclamation point, is the percent sign that you want to escape, so that it is actually used in the search pattern for the LIKE clause.

The character following an escape character must be either a wildcard character or the escape character itself. Complete syntax is presented in the *ALLBASE/SQL Reference Manual* .

It is useful to execute the SELECT command *before* executing the INSERT, DELETE, or UPDATE commands in the following situations:

■ When an application updates or deletes rows, the SELECT command can retrieve the target data for user verification before the data is changed. This technique minimizes inadvertent data changes:

> *The program accepts a part number from the user into a host variable*
> *named PartNumber, then retrieves a row for that part.*

```
EXEC SQL SELECT  PartNumber,  BinNumber
          INTO :PartNumber, :BinNumber
          FROM  PurchDB.Inventory
          WHERE  PartNumber = :PartNumber;
```

> *The row is displayed, and the user is asked if the bin number is*
> *to be changed. If not, the user is prompted for another part number.*
> *If so, the user is prompted for the new bin number, which is accepted into*
> *the host variable named BINNUMBER. Then the UPDATE*
> *command is executed:*

```
EXEC SQL UPDATE PurchDB.Inventory
          SET BinNumber  = :BinNumber
          WHERE PartNumber = :PartNumber;
```

■ To prohibit the multiple-row changes possible if multiple rows qualify for an UPDATE or DELETE operation, an application can use the SELECT command. If multiple rows qualify for the SELECT operation, the UPDATE or DELETE would not be executed. Alternatively, the user could be advised that multiple rows would be affected and given a choice about whether to perform the change:

> *The program prompts the user for an order number and a vendor part*
> *number in preparation for allowing the user to change the vendor part number.*

The following SELECT command determines whether more than one line item
exists on the order for the specified vendor part number:

```
EXEC SQL SELECT  ItemNumber
           INTO :ItemNumber
           FROM  PurchDB.OrderItems
          WHERE  OrderNumber     = :OrderNumber
            AND  VendPartNumber = :VendPartNumber;
```

When more than one row qualifies for this query, the program lets the
user decide whether to proceed with the update operation.

■ When an application lets the user INSERT a row that must contain a value higher than an
existing value, the SELECT command can identify the highest existing value:

```
EXEC SQL SELECT  MAX(OrderNumber)
           INTO :MaxOrderNumber
           FROM  PurchDB.Orders;
```

The program can increment the maximum order number by one, then provide
the user with the new number and prompt for information describing the new order.

The INSERT Command

In simple data manipulation, you use INSERT command syntax to either insert a single row
or copy one or more rows into a table from another table.

Use the following syntax of the INSERT command to insert a single row:

```
INSERT INTO  TableName
            (ColumnNames)
     VALUES (DataValues)
```

You can omit *ColumnNames* when you provide values for all columns in the target table:

```
EXEC SQL INSERT INTO   PurchDB.Parts
              VALUES (:PartNumber,
                      :PartName   :PartNameInd,
                      :SalesPrice :SalesPriceInd);
```

Remember that when you *do* include ColumnNames but do not name all the columns in the
target table, ALLBASE/SQL attempts to insert a null value into each unnamed column. If an
unnamed column was defined as NOT NULL, the INSERT command fails.

To copy one or more rows from one or more tables to another table, use the following form of the INSERT command:

```
INSERT  INTO   TableName
               (ColumnNames)
       SELECT   SelectList
         FROM   TableNames
        WHERE   SearchCondition1
     GROUP BY   ColumnName
       HAVING   SearchCondition2
```

Note that the SELECT command embedded in this INSERT command *cannot* contain an INTO or ORDER BY clause. In addition, any host variables used must be within the WHERE or HAVING clauses:

The following example copies historical data for filled orders into PurchDB.OldOrders, then deletes rows for these orders from PurchDB.Orders, keeping that table minimal in size. The INSERT command copies rows from PurchDB.Orders to PurchDB.OldOrders.

```
EXEC SQL INSERT  INTO   PurchDB.OldOrders
                        (OldOrder, OldVendor, OldDate)
                SELECT  OrderNumber, VendorNumber, OrderDate
                  FROM  PurchDB.Orders
                 WHERE  OrderNumber = :OrderNumber;
```

Then the DELETE command deletes rows from PurchDB.Orders:

```
EXEC SQL DELETE  FROM   PurchDB.Orders
                 WHERE  OrderNumber: = OrderNumber;
```

The UPDATE Command

In simple data manipulation, you use this syntax of the UPDATE command to change data in one or more columns:

```
UPDATE  TableName
   SET  Columname = ColumnValue
        [,...]
 WHERE  SearchCondition
```

As in the case of the DELETE command, if you omit the WHERE clause, the value of any column specified is changed in *all* rows of the table.

If the WHERE clause is specified, all rows satisfying the search condition are changed:

```
EXEC SQL UPDATE PurchDB.Vendors
            SET ContactName   = :ContactName :ContactNameInd,
                VendorStreet  = :VendorStreet,
                VendorCity    = :VendorCity,
                VendorState   = :VendorState,
                VendorZipCode = :VendorZipCode
          WHERE VendorNumber  = :VendorNumber;
```

In this example, column ContactName can contain a null value. To insert a null value, the program must assign a number less than 0 to the indicator variable for this column, ContactNameInd:

The program prompts the user for new values

```
prompt ('Enter Vendor Street> ');
readln (VendorStreet);

prompt ('Enter Vendor City> ');
readln (VendorCity);

prompt ('Enter Vendor State> ');
readln (VendorState);

prompt ('Enter Vendor Zip Code> ');
readln (VendorZipCode);

prompt ('Enter Contact Name (0 for null)> ');
readln (ContactName);
```

If the user enters a 0 to assign a null value to column ContactName,
the program assigns a -1 to the indicator variable; otherwise, the program
assigns a 0 to this variable:

```
if ContactName = '0' then
   ContactNameInd := -1
else
   ContactNameInd :=  0;
```

The DELETE Command

In simple data manipulation, you use the DELETE command to delete one or more rows from a table, as shown in the following syntax:

DELETE FROM *TableName*
      WHERE *SearchCondition*

The WHERE clause specifies a *SearchCondition* rows must meet to be deleted, for example:

EXEC SQL DELETE FROM PurchDB.Orders
              WHERE OrderDate < :OrderDate;

If the WHERE clause is omitted, *all* rows in the table are deleted.

Transaction Management for Simple Operations

The major objectives of transaction management are to minimize the contention for locks and to ensure logical data consistency. Minimizing lock contention implies short transactions and/or locking small, unique parts of a database. Logical data consistency implies keeping data manipulations that should all occur or all not occur within a single transaction. Defining your transactions should always be made with these two objectives in mind. For in depth transaction management information, refer to the *ALLBASE/SQL Reference Manual* .

Most simple data manipulation applications are for random operations on a minimal number of related rows that satisfy very specific criteria. To minimize lock contention, you should begin a new transaction each time these criteria change. For example, if an application displays order information for random orders, delimit each new query with a BEGIN WORK and a COMMIT WORK command:

The program accepts an order number from the user.

```
EXEC SQL BEGIN WORK;
EXEC SQL SELECT  OrderNumber,
                 VendorNumber,
                 OrderDate
           INTO :OrderNumber,
                :VendorNumber  :VendorNumberInd,
                :OrderDate     :OrderDateInd
           FROM  PurchDB.Orders
          WHERE  OrderNumber = :OrderNumber;
```

Error checking is done here.

```
EXEC SQL COMMIT WORK;
```

The program displays the row, then prompts for another order number.

Because SELECT commands are often executed prior to a related UPDATE, DELETE, or INSERT command, you must decide whether to make each command a separate transaction or combine commands within one transaction. And you must decide which isolation level to use to attain your desired data consistency and to minumize possible lock contention.

- If, for example, you combine SELECT and DELETE operations within one transaction, when the DELETE command is executed, the row deleted is guaranteed to be the same row retrieved and displayed for the user. However, if the program user goes to lunch between SELECT and DELETE commands, and the default isolation level (RR) is in effect, no other users can modify the page or table locked by the SELECT command until the transaction terminates.

- If you put the SELECT and DELETE operations in separate transactions, another transaction may change the target row(s) before the DELETE command is executed. Therefore, the user may delete a row different from that originally intended. One way to handle this situation is as follows:

```
EXEC SQL BEGIN WORK;
```

The SELECT command is executed and the query result displayed.

```
EXEC SQL COMMIT WORK;
```

The program user requests that the row be deleted.

```
EXEC SQL BEGIN WORK;
```

*The SELECT command is re-executed, and the program compares the original
query result with the new one. If the query results match, the DELETE
command is executed.*

```
EXEC SQL COMMIT WORK;
```

*If the new query result does not match the original query result,
the program re-executes the SELECT command to display the query result.*

In the case of some multi-command transactions, you must execute multiple data
manipulation commands within a single transaction for the sake of logical data consistency:

In the following example the DELETE and INSERT commands are used in place of the
UPDATE command to insert null values into the target table.

```
EXEC SQL BEGIN WORK;
```

The DELETE command is executed.

If the DELETE command fails, the transaction can be terminated as follows:

```
EXEC SQL COMMIT WORK;
```

If the DELETE command succeeds, the INSERT command is executed.

If the INSERT command fails, the transaction is terminated as follows:

```
EXEC SQL ROLLBACK WORK;
```

If the INSERT command succeeds, the transaction is terminated as follows:

```
EXEC SQL COMMIT WORK;
```

Logical data consistency is also an issue when an UPDATE, INSERT, or DELETE command
may operate on multiple rows. If one of these commands fails after only *some* of the target
rows have been operated on, you must use a ROLLBACK WORK command to ensure that
any row changes made before the failure are undone:

```
EXEC SQL DELETE FROM PurchDB.Orders
              WHERE OrderDate < :OrderDate;

if SQLCA.SQLCODE <> OK then EXEC SQL ROLLBACK WORK;
```

Program Using Simple DML Operations

The flow chart shown in Figure 7-1 summarizes the functionality of program pasex7. This program uses the four simple data manipulation commands to operate on the PurchDB.Vendors table. Program pasex7 uses a function menu to determine whether to execute one or more SELECT, UPDATE, DELETE, or INSERT operations. Each execution of a simple data manipulation command is done in a separate transaction.

The runtime dialog for program pasex7 appears in Figure 7-2, and the source code in Figure 7-3.

Function ConnectDBE starts a DBE session ⦗48⦘. This function executes the CONNECT command ⦗2⦘ for the sample DBEnvironment, PartsDBE.

The next operation performed depends on the number entered in response to this menu ⦗49⦘:

■ The program terminates if 0 is entered.

■ Procedure Select is executed if 1 is entered.

■ Procedure Update is executed if 2 is entered.

■ Procedure Delete is executed if 3 is entered.

■ Procedure Insert is executed if 4 is entered.

Procedure Select

Procedure Select ⦗9⦘ prompts for a vendor number or a 0 ⦗10⦘. If a 0 is entered, the function menu is re-displayed. If a vendor number is entered, procedure BeginTransaction is executed ⦗11⦘ to issue the BEGIN WORK command ⦗4⦘. Then a SELECT command is executed to retrieve all data for the vendor specified from PurchDB.Vendors ⦗12⦘. The SQLCA.SQLCODE returned is examined to determine the next action:

■ If no rows qualify for the SELECT operation, a message ⦗14⦘ is displayed and the transaction terminated ⦗16⦘. Procedure CommitWork terminates the transaction by executing the COMMIT WORK command ⦗5⦘. The user is then re-prompted for a vendor number or a 0.

■ If more than one row qualifies for the SELECT operation, a different message is displayed and procedure CommitWork ⦗5⦘ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.

■ If the SELECT command execution results in an error condition, procedure SqlStatusCheck is executed ⦗15⦘. This procedure executes SQLEXPLAIN ⦗1⦘ to display all error messages. Then the transaction is terminated ⦗16⦘ and the user re-prompted for a vendor number or a 0.

■ If the SELECT command can be successfully executed, procedure DisplayRow ⦗13⦘ is executed to display the row. This procedure examines the null indicators for each of the three potentially null columns (ContactName, PhoneNumber, and VendorRemarks). If any null indicator contains a value less than 0 ⦗8⦘, a message indicating that the value is null is displayed. After the row is completely displayed, the transaction is terminated ⦗16⦘ and the user re-prompted for a vendor number or a 0.

Procedure Update

Procedure Update ⓐ lets the user UPDATE the value of a column only if it contains a null value. The procedure prompts for a vendor number or a 0 ⓐ. If a 0 is entered, the function menu is re-displayed. If a vendor number is entered, procedure BeginTransaction is executed ⓐ. Then a SELECT command is executed to retrieve data from PurchDB.Vendors for the vendor specified ⓐ. The SQLCA.SQLCODE returned is examined to determine the next action:

- If no rows qualify for the SELECT operation, a message ⓐ is displayed and the transaction is terminated ⓐ. The user is then re-prompted for a vendor number or a 0.

- If more than one row qualifies for the SELECT operation, a different message is displayed and procedure CommitWork ⑤ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.

- If the SELECT command execution results in an error condition, procedure SqlStatusCheck is executed ⓐ. Then the transaction is terminated ⓐ and the user re-prompted for a vendor number or a 0.

- If the SELECT command can be successfully executed, procedure DisplayUpdate ⓐ is executed. This procedure executes procedure DisplayRow to display the row retrieved ⓐ. Function AnyNulls is then executed to determine whether the row contains any null values. This boolean function evaluates to TRUE if the indicator variable for any of the three potentially null columns contains a non-zero value ⑥.

  If function AnyNulls evaluates to FALSE, a message is displayed ⑦ and the transaction is terminated ⓐ; the user is then re-prompted for a vendor number or a 0.

  If function AnyNulls evaluates to TRUE, the null indicators are examined to determine which of them contain a negative value ⓐ. A negative null indicator means the column contains a null value, and the user is prompted for a new value ⓐ. If the user enters a 0, the program assigns a -1 to the null indicator ⓐ so that when the UPDATE command ⓐ is executed, a null value is assigned to that column. If a non-zero value is entered, the program assigns a 0 to the null indicator so that the value specified is assigned to that column. After the UPDATE ⓐ command is executed, the transaction is terminated ⓐ and the user re-prompted for a vendor number or a 0.

Procedure Delete

Procedure Delete ⓐ lets the user DELETE one row. The procedure prompts for a vendor number or a 0 ⓐ. If a 0 is entered, the function menu is re-displayed. If a vendor number is entered, procedure BeginTransaction is executed ⓐ. Then a SELECT command is executed to retrieve all data for the vendor specified from PurchDB.Vendors ⓐ. The SQLCA.SQLCODE returned is examined to determine the next action:

- If no rows qualify for the SELECT operation, a message ⓐ is displayed and the transaction is terminated ⓐ. The user is then re-prompted for a vendor number or a 0.

- If more than one row qualifies for the SELECT operation, a different message is displayed and procedure CommitWork ⑤ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.

- If the SELECT command execution results in an error condition, procedure SqlStatusCheck is executed ⟨39⟩. Then the transaction is terminated ⟨40⟩ and the user re-prompted for a vendor number or a 0.

- If the SELECT command can be successfully executed, procedure DisplayDelete ⟨37⟩ is executed. This procedure executes procedure DisplayRow to display the row retrieved ⟨30⟩. Then the user is asked whether she wants to actually delete the row ⟨31⟩. If not, the transaction is terminated ⟨40⟩ and the user re-prompted for a vendor number or a 0. If so, the DELETE command ⟨32⟩ is executed before the transaction is terminated ⟨40⟩ and the user re-prompted.

Procedure Insert

Procedure Insert ⟨41⟩ lets the user INSERT one row. The procedure prompts for a vendor number or a 0 ⟨42⟩. If a 0 is entered, the function menu is re-displayed. If a vendor number is entered, the user is prompted for values for each column. The user can enter a 0 to specify a null value for potentially null columns ⟨43⟩; to assign a null value, the program assigns a -1 to the appropriate null indicator ⟨44⟩. After a transaction is started ⟨45⟩, an INSERT command ⟨46⟩ is used to insert a row containing the specified values. After the INSERT operation, the transaction is terminated ⟨47⟩, and the user re-prompted for a vendor number or a 0.

When the user enters a 0 in response to the function menu display, the program terminates by executing procedure TerminateProgram ⟨50⟩. This procedure executes the RELEASE command ⟨3⟩.

```
CONNECT TO DBEnvironment

        │
        ▼
Prompt for Function Number
or 0 to Stop                    ◄─── (C)

        │
        ▼
Is User Input 0 ?  ──Yes──►  Terminate Program

        │ No
        ▼
Is User Input 1 (SELECT) ?  ──Yes──►  Prompt for Vendor Number or 0 for Menu
        │                                      │
        │ No                                   ▼
        ▼                             Is User Input 0 ?  ──Yes──►  (loop back to prompt)
Is User Input 2 (UPDATE) ?  ──Yes─►          │ No
        │                                      ▼
        │ No                           Begin Transaction
        ▼                                      │
Is User Input 3 (DELETE) ?  ──No──►  (B)       ▼
        │                             SELECT and Display Vendor's Data
        │ Yes                                  │
        ▼                                      ▼
       (A)                            End Transaction

Is User Input 0 ?  ──No──►  Begin Transaction
        │ Yes                          │
        ▼                              ▼
(back to Prompt for Function Number)  SELECT and Display Vendor's Data
                                       │
                                       ▼
                              UPDATE any Null Values
                                       │
                                       ▼
                              End Transaction

Is User Input 0 ?  (left side)  ──Yes──► Prompt for Vendor Number or 0 for Menu
        │ No
        ▼
Begin Transaction
        │
        ▼
SELECT and Display Vendor's Data  ──►  End Transaction
```

LG200124_007

Figure 7-1. Flow Chart of Program pasex7
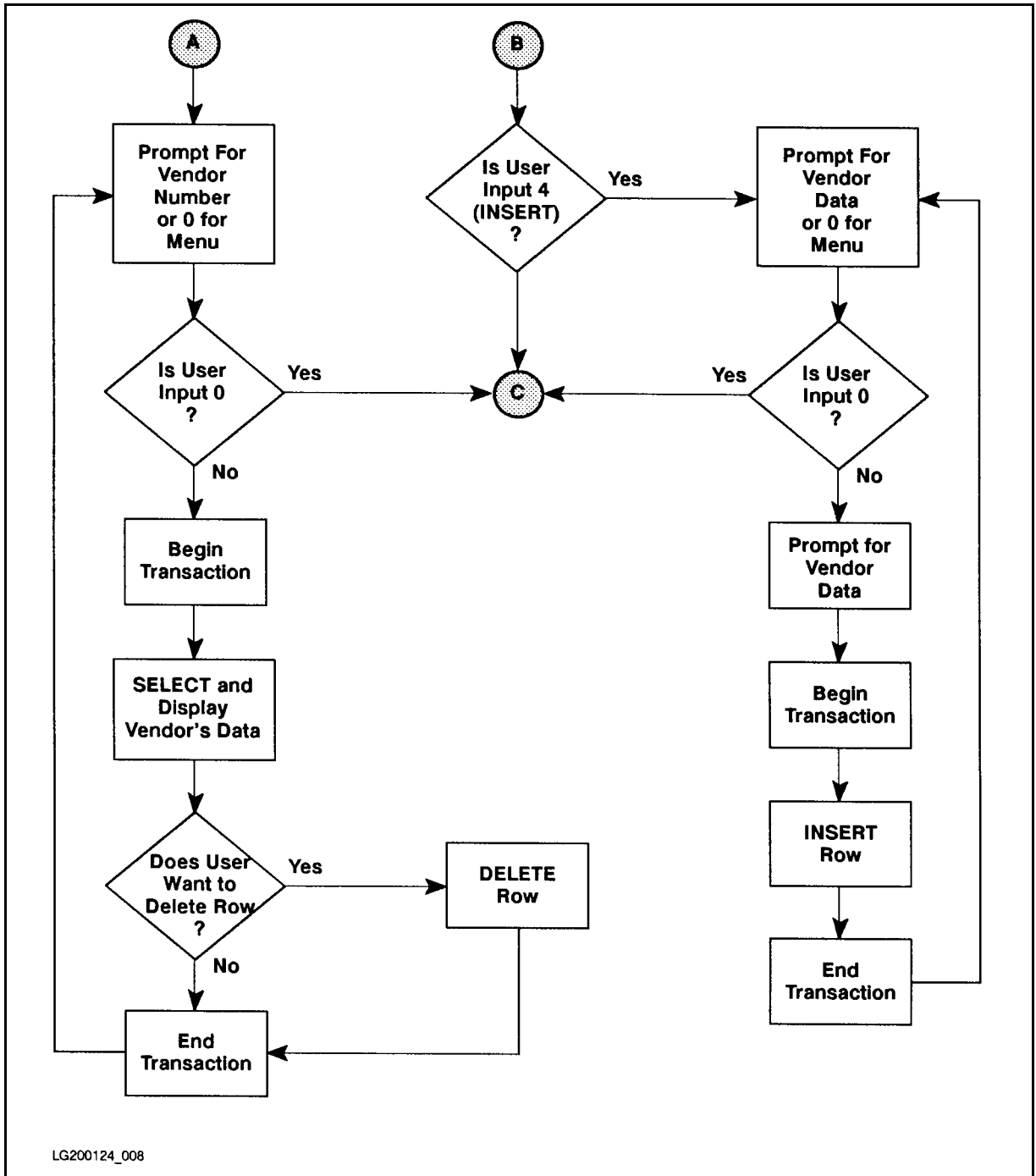
Flow Chart of Program pasex7 (page 2 of 2)

LG200124_008

```
Program for Simple Data Manipulation of Vendors Table - pasex7

Connect to PartsDBE

   1 . . . . SELECT rows from PurchDB.Vendors table
   2 . . . . UPDATE rows with null values in PurchDB.Vendors table
   3 . . . . DELETE rows from PuchDB.Vendors table
   4 . . . . INSERT rows into PurchDB.Vendors table

Enter choice or 0 to stop> 4

  *** Procedure to INSERT rows into PurchDB.Vendors ***

Enter Vendor Number or 0 for MENU> 9016

Enter Vendor Name> Wolfe Works

Enter Contact Name (0 for null)> Stanley Wolfe

Enter Phone Number (0 for null)> 408 975 6061

Enter Vendor Street> 7614 Canine Way

Enter Vendor City> San Jose

Enter Vendor State> CA

Enter Vendor Zip Code> 90016

Enter Vendor Remarks (0 for null)> 0

Begin Work
INSERT row into PurchDB.Vendors
Commit Work

Enter Vendor Number or 0 for MENU> 0

   1 . . . SELECT rows from PurchDB.Vendors table
   2 . . . UPDATE rows with null values in PurchDB.Vendors table
   3 . . . DELETE rows from PurchDB.Vendors table
   4 . . . INSERT rows into PurchDB.Vendors table

Enter choice or 0 to STOP> 1
```

Figure 7-2. Runtime Dialog of Program pasex7

```
   *** Procedure to SELECT rows from PurchDB.Vendors ***


Enter Vendor Number or 0 for MENU> 9016


Begin Work
SELECT * from PurchDB.Vendors

  VendorNumber:              9016
  VendorName:      Wolfe Works
  ContactName:     Stanley Wolfe
  PhoneNumber:     408 975 6061
  VendorStreet:    7614 Canine Way
  VendorCity:      San Jose
  VendorState:     CA
  VendorZipCode:   90016
  VendorRemarks is NULL


Commit Work


Enter Vendor Number or 0 for MENU> 0

   1 . . . SELECT rows from PurchDB.Vendors table
   2 . . . UPDATE rows with null values in PurchDB.Vendors table
   3 . . . DELETE rows from PurchDB.Vendors table
   4 . . . INSERT rows into PurchDB.Vendors table


Enter choice or 0 to STOP> 2

  *** Procedure to UPDATE rows in PurchDB.Vendors ***


Enter Vendor Number or 0 for MENU> 9016


Begin Work
SELECT * from PurchDB.Vendors

  VendorNumber:              9016
  VendorName:      Wolfe Works
  ContactName:     Stanley Wolfe
  PhoneNumber:     408 975 6061
  VendorStreet:    7614 Canine Way
  VendorCity:      San Jose
  VendorState:     CA
  VendorZipCode:   90016
  VendorRemarks is NULL


Enter new VendorRemarks (0 for null)> can expedite shipments
Commit Work
```

Figure 7-2. Runtime Dialog of Program pasex7 (page 2 of 3)

```
   Enter Vendor Number or 0 for MENU> 0

      1 . . . SELECT rows from PurchDB.Vendors table
      2 . . . UPDATE rows with null values in PurchDB.Vendors table
      3 . . . DELETE rows from PurchDB.Vendors table
      4 . . . INSERT rows into PurchDB.Vendors table

   Enter choice or 0 to STOP> 3

     *** Procedure to DELETE rows from PurchDB.Vendors ***

   Enter Vendor Number or 0 for MENU> 9016

   Begin Work
   SELECT * from PurchDB.Vendors

     VendorNumber:            9016
     VendorName:      Wolfe Works
     ContactName:     Stanley Wolfe
     PhoneNumber:     408 975 6061
     VendorStreet:    7614 Canine Way
     VendorCity:      San Jose
     VendorState:     CA
     VendorZipCode:   90016
     VendorRemarks:   can expedite shipments

   Is it OK to DELETE this row (N/Y)? > Y

   DELETE row from PurchDB.Vendors
   Commit Work

   Enter Vendor Number or 0 for MENU> 0

      1 . . . SELECT rows from PurchDB.Vendors table
      2 . . . UPDATE rows with null values in PurchDB.Vendors table
      3 . . . DELETE rows from PurchDB.Vendors table
      4 . . . INSERT rows into PurchDB.Vendors table

   Enter choice or 0 to STOP> 0
```

Figure 7-2. Runtime Dialog of Program pasex7 (page 3 of 3)

```
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* This program illustrates simple data manipulation.  It uses the *)
(* UPDATE command with indicator variables to update any row in    *)
(* Vendors Table that contains null values.  It also uses          *)
(* indicator variables in conjunction with SELECT and INSERT.  The *)
(* DELETE command is also illustrated.                             *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

Program pasex7(input,output);

const
    OK             =        0;
    NotFound       =      100;
    MultipleRows   =  -10002;
    DeadLock       =  -14024;

var
          (* Begin Host Variable Declarations *)
    EXEC SQL BEGIN DECLARE SECTION;
    VendorNumber      : integer;
    VendorName        : packed array[1..30] of char;
    ContactName       : packed array[1..30] of char;
    ContactNameInd    : SqlInd;
    PhoneNumber       : packed array[1..15] of char;
    PhoneNumberInd    : SqlInd;
    VendorStreet      : packed array[1..30] of char;
    VendorCity        : packed array[1..20] of char;
    VendorState       : packed array[1..2]  of char;
    VendorZipCode     : packed array[1..10] of char;
    VendorRemarks     : string[60];
    VendorRemarksInd  : SqlInd;
    SQLMessage        : packed array[1..132] of char;
    EXEC SQL END DECLARE SECTION;
          (* End Host Variable Declarations *)

    SQLCA : SQLCA_type;    (* SQL Communication Area *)

    Abort             : boolean;
    Response          : integer;
    Response1         : packed array[1..3] of char;

procedure TerminateProgram; forward;
procedure SQLStatusCheck; (* Procedure to Display Error Messages *)
begin
```

Figure 7-3. Program pasex7: Using SELECT, UPDATE, DELETE, and INSERT

```
   Abort := FALSE;
   if SQLCA.SQLCODE < DeadLock then Abort := TRUE;

   repeat
   EXEC SQL SQLEXPLAIN :SQLMessage;                                 [1]
   writeln(SQLMessage);
   until SQLCA.SQLCODE = 0;

   if Abort then TerminateProgram;

   end;  (* End SQLStatusCheck Procedure *)
   $PAGE $
   function ConnectDBE: boolean;  (* Function to Connect to PartsDBE *)
   begin

   writeln('Connect to PartsDBE');
   EXEC SQL CONNECT TO '../sampledb/PartsDBE';                      [2]

   ConnectDBE := TRUE;
   if SQLCA.SQLCODE <> OK then
     begin

     ConnectDBE := FALSE;
     SQLStatusCheck;

     end;  (* End if *)
   end;  (* End of ConnectDBE Function *)


   procedure TerminateProgram;   (* Procedure to Release from PartsDBE *)
   begin

   EXEC SQL RELEASE;                                               [3]

   end;                      (* End of TerminateProgramProcedure *)
   $PAGE $

   procedure BeginTransaction;  (* procedure to BEGIN WORK *)

   begin

   writeln('Begin Work');
   EXEC SQL BEGIN WORK;                                            [4]
   if SQLCA.SQLCODE <> OK then
     begin
```

Figure 7-3. Program pasex7: Using SELECT, UPDATE, DELETE, and INSERT (page 2 of 12)

```
   SQLStatusCheck;
   TerminateProgram;
   end;


end;   (* End BeginTransaction procedure *)


procedure CommitWork;    (* Procedure to Commit Work *)
begin


writeln('Commit Work');
EXEC SQL COMMIT WORK;                                                    [5]
if SQLCA.SQLCODE <> OK then
  begin


  SQLStatusCheck;
  TerminateProgram;
  end;
end;   (* End CommitWork Procedure *)


function AnyNulls: boolean; (* Function to test row for null values *)
begin


AnyNulls := TRUE;


if (ContactNameInd = 0) and                                             [6]
   (PhoneNumberInd = 0) and
   (VendorRemarksInd = 0)
then   (* all columns that might be null contain non-null values *)
  begin
    writeln(' No null values exist for this vendor');                   [7]
    AnyNulls := FALSE;
  end;
end; (* End of Null Function *)


procedure DisplayRow;    (* Procedure to Display Vendors Table Rows *)
begin
writeln;
writeln('  VendorNumber:   ', VendorNumber);
writeln('  VendorName:     ', VendorName);
if ContactNameInd <0 then                                               [8]
  writeln('  ContactName is NULL')
else
  writeln('  ContactName:    ', ContactName);
```

Figure 7-3. Program pasex7: Using SELECT, UPDATE, DELETE, and INSERT (page 3 of 12)

```
if PhoneNumberInd <0 then
  writeln('  PhoneNumber is NULL')
else
  writeln('  PhoneNumber:     ', PhoneNumber);
  writeln('  VendorStreet:    ', VendorStreet);
  writeln('  VendorCity:      ', VendorCity);
  writeln('  VendorState:     ', VendorState);
  writeln('  VendorZipCode:   ', VendorZipCode);
if VendorRemarksInd <0 then
  writeln('  VendorRemarks is NULL')
else
  writeln('  VendorRemarks:   ', VendorRemarks);
writeln;
end;  (* End of DisplayRow *)
$PAGE $

procedure Select; (* procedure to select row from Vendors Table *)    ⑨
begin
writeln;
writeln('  *** Procedure to SELECT rows from PurchDB.Vendors ***  ');
writeln;

repeat

writeln;
prompt('Enter Vendor Number or 0 for MENU>  ');                        ⑩
readln(VendorNumber);
writeln;

if VendorNumber <> 0 then
begin
BeginTransaction;                                                      ⑪
writeln('SELECT * from PurchDB.Vendors');
EXEC SQL SELECT VendorNumber,                                          ⑫
                VendorName,
                ContactName,
                PhoneNumber,
                VendorStreet,
                VendorCity,
                VendorState,
                VendorZipCode,
                VendorRemarks
```

Figure 7-3. Program pasex7: Using SELECT, UPDATE, DELETE, and INSERT (page 4 of 12)

```
             INTO :VendorNumber,
                  :VendorName,
                  :ContactName :ContactNameInd,
                  :PhoneNumber :PhoneNumberInd,
                  :VendorStreet,
                  :VendorCity,
                  :VendorState,
                  :VendorZipCode,
                  :VendorRemarks :VendorRemarksInd
           FROM  PurchDB.Vendors
          WHERE  VendorNumber = :VendorNumber;

case SQLCA.SQLCODE of
OK            : DisplayRow;                                     13
NotFound      : begin
                  writeln;
                  writeln('Row not found!');                   14
                end;
MultipleRows:   begin
                  writeln;
                  writeln('WARNING: More than one row qualifies.');   15
                end;
otherwise       begin
                  SQLStatusCheck;                              16
                end;
end;         (* end case *)
CommitWork;                                                    17
end;         (* end if response *)
until VendorNumber = 0;


end;         (* end Select Procedure *)


procedure DisplayUpdate; (* procedure to display and update row *)
begin


DisplayRow;                                                   18
if AnyNulls then
begin


if ContactNameInd < 0 then                                   19
  begin
    writeln;
    prompt('Enter new ContactName (0 for NULL)> ');          20
    readln(ContactName);
  end;
```

Figure 7-3. Program pasex7: Using SELECT, UPDATE, DELETE, and INSERT (page 5 of 12)

```
   if PhoneNumberInd < 0 then
     begin
       writeln;
       prompt('Enter new PhoneNumber (0 for NULL)> ');
       readln(PhoneNumber);
     end;

   if VendorRemarksInd < 0 then
     begin
       writeln;
       prompt('Enter new VendorRemarks (0 for NULL)> ');
       readln(VendorRemarks);
     end;

   if ContactName = '0' then                                         21
     ContactNameInd := -1
   else
     ContactNameInd := 0;

   if PhoneNumber = '0' then
     PhoneNumberInd := -1
   else
     PhoneNumberInd := 0;

   if VendorRemarks = '0' then
     VendorRemarksInd := -1
   else
     VendorRemarksInd := 0;

   EXEC SQL UPDATE PurchDB.Vendors                                   22
             SET ContactName = :ContactName :ContactNameInd,
                 PhoneNumber = :PhoneNumber :PhoneNumberInd,
                 VendorRemarks = :VendorRemarks :VendorRemarksInd
           WHERE VendorNumber = :VendorNumber;

   if SQLCA.SQLCODE <> OK then SQLStatusCheck;

   end;    (* end if AnyNulls *)


   end;    (* end of DisplayUpdate procedure *)


   $PAGE $


   procedure Update;  (* Update a row within the Vendors Table *)    23
```

Figure 7-3. Program pasex7: Using SELECT, UPDATE, DELETE, and INSERT (page 6 of 12)

```
begin
writeln;
writeln('  *** Procedure to UPDATE rows in PurchDB.Vendors ***  ');
writeln;
repeat
writeln;
prompt('Enter Vendor Number or 0 for MENU>  ');                         [24]
readln(VendorNumber);
writeln;

if VendorNumber <> 0 then
begin
  BeginTransaction;                                                      [25]
  writeln ('SELECT * from PurchDB.Vendors');
  EXEC SQL SELECT  VendorNumber,                                         [26]
                   VendorName,
                   ContactName,
                   PhoneNumber,
                   VendorStreet,
                   VendorCity,
                   VendorState,
                   VendorZipCode,
                   VendorRemarks
           INTO  :VendorNumber,
                 :VendorName,
                 :ContactName   :ContactNameInd,
                 :PhoneNumber   :PhoneNumberInd,
                 :VendorStreet,
                 :VendorCity,
                 :VendorState,
                 :VendorZipCode,
                 :VendorRemarks   :VendorRemarksInd
           FROM  PurchDB.Vendors
           WHERE  VendorNumber = :VendorNumber;
  case SQLCA.SQLCODE of
  OK            : begin
                    DisplayUpdate;                                       [27]
                  end;
  NotFound      : begin
                    writeln;
                    writeln('Row not found!');                           [28]
                  end;
  MultipleRows : begin
                    writeln;
```

Figure 7-3. Program pasex7: Using SELECT, UPDATE, DELETE, and INSERT (page 7 of 12)

```
                writeln('WARNING: More than one row qualifies.');          [29]
                    end;
  otherwise       begin
                     SQLStatusCheck;                                        [30]
                    end;
  end;  (* case *)
CommitWork;                                                                 [31]
end;  (* end if response *)
until VendorNumber = 0;


end;  (* End of Update Procedure *)


$PAGE $
procedure DisplayDelete; (* procedure to display and delete a row *)


begin


DisplayRow;                                                                 [32]


prompt('Is it OK to DELETE this row (N/Y)? >  ');                           [33]
readln(Response1);
writeln;
if response1[1] in ['Y','y'] then
  begin
    writeln;
    writeln('DELETE row from PurchDB.Vendors');
    EXEC SQL DELETE FROM PurchDB.Vendors                                    [34]
                   WHERE VendorNumber = :VendorNumber;
    if SQLCA.SQLCODE <> OK then SQLStatusCheck;
  end;
end;    (* end procedure DisplayDelete *)


procedure Delete;                                                          [35]
(* procedure to delete a row from PurchDB.Vendors *)


begin
writeln;
writeln('  *** Procedure to DELETE rows from PurchDB.Vendors ***  ');
writeln;
repeat
writeln;
prompt('Enter Vendor Number or 0 for MENU>  ');                            [36]
readln(VendorNumber);
writeln;
```

Figure 7-3. Program pasex7: Using SELECT, UPDATE, DELETE, and INSERT (page 8 of 12)

```
if VendorNumber <> 0 then
begin
BeginTransaction;                                                    ▢37
writeln('SELECT * from PurchDB.Vendors');
EXEC SQL SELECT VendorNumber,                                        ▢38
               VendorName,
               ContactName,
               PhoneNumber,
               VendorStreet,
               VendorCity,
               VendorState,
               VendorZipCode,
               VendorRemarks
          INTO :VendorNumber,
               :VendorName,
               :ContactName :ContactNameInd,
               :PhoneNumber :PhoneNumberInd,
               :VendorStreet,
               :VendorCity,
               :VendorState,
               :VendorZipCode,
               :VendorRemarks :VendorRemarksInd
          FROM  PurchDB.Vendors
          WHERE  VendorNumber = :VendorNumber;
case SQLCA.SQLCODE of
OK            : DisplayDelete;                                       ▢39
NotFound      : begin
                    writeln;
                    writeln('Row not found!');                       ▢40
                 end;
MultipleRows:  begin
               writeln('WARNING: More than one row qualifies.');     ▢41
                 end;
otherwise      begin
                   SQLStatusCheck;                                   ▢42
                 end;
end;  (* end case *)
CommitWork;                                                          ▢43
end;  (* end if response *)
until VendorNumber = 0;
end;  (* end Delete procedure *)
procedure Insert;                                                    ▢44
(* procedure to insert a row into PurchDB.Vendors *)
begin
```

Figure 7-3. Program pasex7: Using SELECT, UPDATE, DELETE, and INSERT (page 9 of 12)

```
writeln;
writeln(' *** Procedure to INSERT rows into PurchDB.Vendors *** ');
writeln;
repeat
writeln;
prompt('Enter Vendor Number or 0 for MENU> ');                          45
readln(VendorNumber);
writeln;

if VendorNumber <> 0 then
begin
writeln;
prompt('Enter Vendor Name> ');
readln(VendorName);
writeln;

writeln;
prompt('Enter Contact Name (0 for null)> ');                            46
readln(ContactName);
if ContactName = '0' then
   ContactNameInd := -1                                                 47
else
   ContactNameInd :=  0;
writeln;

prompt('Enter Phone Number (0 for null)> ');
readln(PhoneNumber);
if PhoneNumber = '0' then
   PhoneNumberInd := -1
else
   PhoneNumberInd :=  0;
writeln;

prompt('Enter Vendor Street> ');
readln(VendorStreet);
writeln;

prompt('Enter Vendor City> ');
readln(VendorCity);
writeln;

prompt('Enter Vendor State> ');
readln(VendorState);
writeln;
```

Figure 7-3. Program pasex7: Using SELECT, UPDATE, DELETE, and INSERT (page 10 of 12)

```
prompt('Enter Vendor Zip Code>  ');
readln(VendorZipCode);
writeln;
prompt('Enter Vendor Remarks (0 for null)>  ');
readln(VendorRemarks);
if VendorRemarks = '0' then
   VendorRemarksInd := -1
else
   VendorRemarksInd := 0;
BeginTransaction;                                                    [48]
writeln('INSERT row into PurchDB.Vendors');
EXEC SQL INSERT                                                      [49]
           INTO  PurchDB.Vendors
               (VendorNumber,
                VendorName,
                ContactName,
                PhoneNumber,
                VendorStreet,
                VendorCity,
                VendorState,
                VendorZipCode,
                VendorRemarks)
       VALUES (:VendorNumber,
                :VendorName,
                :ContactName :ContactNameInd,
                :PhoneNumber :PhoneNumberInd,
                :VendorStreet,
                :VendorCity,
                :VendorState,
                :VendorZipCode,
                :VendorRemarks :VendorRemarksInd);
if SQLCA.SQLCODE <> OK then SQLStatusCheck;
CommitWork;                                                          [50]
end; (* end if response *)
until VendorNumber = 0;
end;    (* end of insert procedure *)


begin  (* Beginning of Program *)


writeln('Program for Simple Data Manipulation of Vendors Table');
writeln;


if ConnectDBE then                                                  [51]
begin
  repeat
```

Figure 7-3. Program pasex7: Using SELECT, UPDATE, DELETE, and INSERT (page 11 of 12)

```
    writeln;
    writeln('  1 . . . SELECT rows from PurchDB.Vendors table');
    writeln('  2 . . . UPDATE rows with null values in PurchDB.Vendors table
    writeln('  3 . . . DELETE rows from PurchDB.Vendors table');
    writeln('  4 . . . INSERT rows into PurchDB.Vendors table');
    writeln;
    prompt('Enter choice or 0 to STOP>  ');
    readln(Response);
    writeln;

    if Response <> 0 then
    begin
        case Response of
        1  :  Select;
        2  :  Update;
        3  :  Delete;
        4  :  Insert;
        otherwise   writeln('Enter 0-4 only, please');

      end;   (* end case *)
      end;   (* end if Response *)
  until Response = 0;
  TerminateProgram;
end   (* end if connect *)
else
  writeln('Cannot connect to PartsDBE');
end.    (* end of program *)
```

52

53

Figure 7-3. Program pasex7: Using SELECT, UPDATE, DELETE, and INSERT (page 12 of 12)

8

Processing with Cursors

Processing with cursors gives you the option of operating on a multiple-row query result, *one* row at a time. The query result is referred to as an **active set**. You use a pointer called a **cursor** to move through the active set, retrieving a row at a time into host variables and optionally updating or deleting the row. Reporting applications may find this technique useful. Update applications such as those that periodically operate on tables not being concurrently accessed (e.g., inventory adjustments) may also find this technique useful.

This chapter presents:

- SQL Cursor Commands.

- Transaction Management for Cursor Operations (Further discussion of transaction management is found in the *ALLBASE/SQL Reference Manual* .

- Sample Program Using Cursor Operations.

The emphasis in this chapter is on FETCHing one row at a time. For an example of using the FETCH command with the BULK option, see the "BULK FETCH" section of Chapter 9.

SQL Cursor Commands

The following ALLBASE/SQL commands are used in cursor processing:

- DECLARE CURSOR defines a cursor and associates it with a query.

- OPEN defines the active set.

- FETCH retrieves one row of the active set into host variables; when a row resides in host variables it is known as the **current row**. When a row is current and the active set is a query result derived from a single table, you can use one of the following two commands to change the row.

- UPDATE WHERE CURRENT updates the current row.

- DELETE WHERE CURRENT deletes the current row.

- CLOSE terminates access to the active set and frees up ALLBASE/SQL buffer space used to handle the cursor.

For a given cursor, the commands listed above (with the exception of DECLARE CURSOR) should be contained within the same transaction. Refer to the *ALLBASE/SQL Reference Manual* for the complete syntax and semantics of these commands.

DECLARE CURSOR

The DECLARE CURSOR command syntax names a cursor and associates with it a particular SELECT command:

```
DECLARE CursorName
        [IN DBEFileSetName]
        CURSOR FOR
        SelectCommand
        [FOR UPDATE OF ColumnName [,ColumnName...]
```

This command does not retrieve rows from a table.

In the physical order of your source program statements, the DECLARE CURSOR command must precede any command that references the cursor; for example, the OPEN command.

Note that the DECLARE CURSOR command has two optional clauses:

- The IN clause defines the DBEFileSet in which the section generated by the preprocessor for this command is stored. If no IN clause is specified, file space in the SYSTEM DBEFileSet is used.

- The FOR UPDATE OF clause is used when you use the UPDATE WHERE CURRENT command to update a current row. This command may offer the simplest way to update a current row, but it imposes certain restrictions on the SelectCommand. Updating a current row is fully discussed later in this chapter under "UPDATE WHERE CURRENT."

The SELECT command syntax for cursor declarations that do not include the FOR UPDATE clause can consist of any of the SELECT command clauses *except* the INTO clause:

```
  SELECT SelectList
    FROM TableNames
   WHERE SearchCondition1
GROUP BY ColumnNames
  HAVING SearchCondition2
ORDER BY ColumnIdentifiers
```

A SELECT command associated with a cursor does not name output host variables, but may name input host variables in the select list, the WHERE clause, or the HAVING clause. In the following example, the rows qualifying for the query result will be those with a COUNTCYCLE matching that specified by the user in input host variable *COUNTCYCLE*:

```
EXEC SQL DECLARE Inventory
         CURSOR FOR
         SELECT PartNumber,
                BinNumber,
                QtyOnHand,
                AdjustmentQty
           FROM PurchDB.Inventory
          WHERE CountCycle = :CountCycle
       ORDER BY BinNumber;
```

When performing cursor processing, the ORDER BY clause may be useful. In the previous example, the rows in the query result will be in order by ascending bin number, to help the program user, who will be moving from bin to bin, taking a physical inventory.

The DECLARE CURSOR command is actually a preprocessor directive. When the Pascal preprocessor parses this command, it stores a section in the target DBEnvironment. At run time, the section is not executed when the DECLARE CURSOR command is encountered, but when the OPEN command is executed. Because the DECLARE CURSOR command is not executed at run time, you do not need to perform status checking in your program following this command.

OPEN

The OPEN command examines any input host variables, determines the active set, and allocates internal buffer space for the active set. (See the "Using KEEP CURSOR" section of this chapter for more information.) The syntax is as follows:

$$\text{OPEN } CursorName \left[\text{KEEP CURSOR} \right] \begin{Bmatrix} \text{WITH LOCKS} \\ \text{WITH NOLOCKS} \end{Bmatrix}$$

The following command opens the cursor defined earlier:

```
EXEC SQL OPEN Inventory;
```

Once the active set is defined, the FETCH command will retrieve data from it, one row at a time.

You can use the KEEP CURSOR WITH NOLOCKS option for a cursor that involves sorting, whether through the use of a DISTINCT, GROUP BY, or ORDER BY clause, or as the result of a union or a join operation. However, for kept cursors involving sorting, ALLBASE/SQL does not ensure data integrity. See the *ALLBASE/SQL Reference Manual* for more information on ensuring data integrity.

FETCH

The FETCH command defines a current row and delivers the row into output host variables with the following syntax:

```
FETCH CursorName INTO OutputHostVariables
```

Remember to include indicator variables when one or more columns in the query result may contain a null value:

```
EXEC SQL FETCH   Inventory
         INTO  :PartNumber,
               :BinNumber,
               :QtyOnHand     :QtyOnHandInd,
               :AdjustmentQty :AdjustmentQtyInd;
```

The first time you execute the FETCH command, the first row in the query result becomes the current row. With each subsequent execution of the FETCH command, each succeeding row in the query result becomes current. After the last row in the query result has been fetched, ALLBASE/SQL sets SQLCODE to 100. ALLBASE/SQL also sets SQLCODE to 100

if no rows qualify for the active set. You should test for an SQLCODE value of 100 after each execution of the FETCH command to determine whether to re-execute this command:

```
var
⋮
   DoFetch   : boolean;
⋮
procedure GetARow;
begin


do
Fetch := TRUE;
while DoFetch = TRUE do
```

The FETCH command appears here.

```
case SQLCA.SQLCODE of
0          : DisplayRow;
100        : begin
                 DoFetch := FALSE;
                 CloseCursor;
                 CommitWork;
             end;
otherwise    begin
                 DoFetch := FALSE;
                 SqlStatusCheck;
                 CloseCursor;
                 RollBackWork;
             end;
end;
```

When a row is current, you can update it by using the UPDATE WHERE CURRENT command or delete it by using the DELETE WHERE CURRENT command.

UPDATE WHERE CURRENT

This command can be used to update the current row when the SELECT command associated with the cursor does *not* contain one of the following:

- A DISTINCT clause in the select list.
- An aggregate function in the select list.
- A FROM clause with more than one table.
- An ORDER BY clause.
- A GROUP BY clause.

The UPDATE WHERE CURRENT command syntax identifies the active set to be updated by naming the cursor and the column(s) to be updated.

```
        UPDATE  TableName
           SET  ColumnName = ColumnValue
                [,...]
         WHERE CURRENT OF CursorName
```

Any columns you name in this command must also have been named in a FOR UPDATE
clause in the related DECLARE CURSOR command:

```
EXEC SQL DECLARE  AdjustQtyOnHand
            CURSOR FOR
            SELECT PartNumber,
                   BinNumber,
                   QtyOnHand,
                   AdjustmentQty
             FROM PurchDB.Inventory
            WHERE QtyOnHand IS NOT NULL
              AND AdjustmentQty IS NOT NULL
      FOR UPDATE OF QtyOnHand,
                    AdjustmentQty;

EXEC SQL OPEN AdjustQtyOnHand;
```

The output host variables do not need to include indicator variables,
because the SELECT command associated with the cursor eliminates any rows
having null values from the active set:

```
EXEC SQL FETCH   AdjustQtyOnHand
            INTO :PartNumber,
                 :BinNumber,
                 :QtyOnHand,
                 :AdjustmentQty;
   .
   .
   .
EXEC SQL UPDATE  PurchDB.Inventory
            SET QtyOnHand      = :QtyOnHand + :AdjustmentQty,
                AdjustmentQty = 0
            WHERE CURRENT OF AdjustQtyOnHand;
```

In this example, the order of the rows in the query result is not important. Therefore the
SELECT command associated with cursor ADJUSTQTYONHAND does not need to contain
an ORDER BY clause and the UPDATE WHERE CURRENT command can be used.

In cases where order *is* important and the ORDER BY clause must be used, you can use the UPDATE command with the WHERE clause to update values in the current row *as well as* any other rows that qualify for the search condition.

```
EXEC SQL DECLARE Inventory
         CURSOR FOR
         SELECT PartNumber,
                BinNumber,
                QtyOnHand,
                AdjustmentQty
           FROM PurchDB.Inventory
          WHERE CountCycle = :CountCycle
       ORDER BY BinNumber;
  .
  .
  .
EXEC SQL FETCH  Inventory
         INTO :PartNumber,
              :BinNumber,
              :QtyOnHand      :QtyOnHandInd,
              :AdjustmentQty :AdjustmentQtyInd;
```

*The program displays the current row. If the QtyOnHand value is not
null, the program prompts the user for an adjustment quantity. Adjustment
quantity is the difference between the quantity actually in the bin and
the QtyOnHand in the row displayed. If the QtyOnHand value is null, the
program prompts the user for both QtyOnHand and AdjustmentQty. Any value
entered is used in the following UPDATE command:*

```
EXEC SQL UPDATE  PurchDB.Inventory
            SET QtyOnHand = :QtyOnHand :QtyOnHandInd,
                AdjustmentQty = :AdjustmentQty :AdjustmentQtyInd
          WHERE  PartNumber = :PartNumber
            AND BinNumber  = :BinNumber;
```

After either the UPDATE WHERE CURRENT or the UPDATE command is executed, the current row remains the same until the FETCH command is re-executed.

If you want to execute UPDATE commands inside the FETCH loop, remember that more than one row in the active set may qualify for the UPDATE operation, as when the WHERE clause in the UPDATE command does not specify a unique key. When more than one row qualifies for the UPDATE, you may not see a changed row unless you CLOSE and re-OPEN the cursor. To avoid this problem, either ensure your UPDATE commands change only one row (the current row) or perform the UPDATE operations outside the FETCH loop.

DELETE WHERE CURRENT

This command can be used to delete the current row when the SELECT command associated with the cursor does *not* contain one of the following:

- A DISTINCT clause in the select list.
- An aggregate function in the select list.
- A FROM clause with more than one table.
- An ORDER BY clause.
- A GROUP BY clause.

The DELETE WHERE CURRENT command has a very simple syntax:

```
DELETE FROM TableName
        WHERE CURRENT OF CursorName
```

The DELETE WHERE CURRENT command can be used in conjunction with a cursor declared with *or* without the FOR UPDATE clause:

> *The program displays the current row and asks the user whether to update or delete it. If the user wants to delete the row, the following command is executed.*

```
EXEC SQL DELETE FROM PurchDB.Inventory
          WHERE CURRENT OF AdjustQtyOnHand
```

Even though the SELECT command associated with cursor INVENTORY names only some of the columns in table PURCHDB.INVENTORY, the entire current row is deleted.

After the DELETE WHERE CURRENT command is executed, there is no current row. You must re-execute the FETCH command to obtain another current row.

As in the case of the UPDATE WHERE CURRENT command, if the SELECT command associated with the cursor contains an ORDER BY clause or other components listed earlier, you can use the DELETE command with the WHERE clause to delete a row:

```
EXEC SQL DELETE FROM PurchDB.Inventory
              WHERE PartNumber = :PartNumber
              AND BinNumber  = :BinNumber
```

If you use the DELETE command to delete a row while using a cursor to examine an active set, remember that more than one row will be deleted if multiple rows satisfy the conditions specified in the WHERE clause of the DELETE command. In addition, the row that is current when the DELETE command is executed remains the current row until the FETCH command is re-executed.

CLOSE

When you no longer want to operate on the active set, use the CLOSE command with the
following syntax:

```
CLOSE CursorName
```

The CLOSE command frees up ALLBASE/SQL internal buffers used to handle cursor
operations. This command does *not* release any locks obtained since the cursor was opened; to
release locks, you must terminate the transaction with a COMMIT WORK or a ROLLBACK
WORK:

> *The program opens a cursor and operates on the active set. After the last*
> *row has been operated on, the cursor is closed:*

```
EXEC SQL CLOSE Inventory;
```

> *Additional SQL commands are executed, then the transaction is terminated:*

```
EXEC SQL COMMIT WORK;
```

When a transaction terminates, any cursors opened during that transaction are automatically
closed, unless you ar using the KEEP CURSOR option of the OPEN command. To avoid
possible confusion, it is good programming practice to *always* use the CLOSE command
followed by a COMMIT WORK to explicitly close any open cursors before ending a
transaction. Refer to the "Using KEEP CURSOR" section of this chapter for more
information on closing a kept cursor.

Transaction Management for Cursor Operations

The time at which ALLBASE/SQL obtains locks during cursor processing depends on whether ALLBASE/SQL uses an index scan or a sequential scan to retrieve the query result.

When a cursor is based on a SELECT command for which ALLBASE/SQL can use an *index scan*, locks are obtained when the FETCH command is executed. In the following example, an index scan can be used, because the predicate is optimizable and an index exists on column ORDERNUMBER:

```
EXEC SQL DECLARE OrderReview
          CURSOR FOR
          SELECT OrderNumber,
                 ItemNumber,
                 OrderQty,
                 ReceivedQty
            FROM PurchDB.OrderItems
            WHERE OrderNumber = :OrderNumber;
```

When the cursor is based on a SELECT command for which ALLBASE/SQL will use a *sequential scan*, locks are obtained when the OPEN command is executed. A sequential scan would be used in conjunction with the following cursor:

```
EXEC SQL DECLARE OrderReview
          CURSOR FOR
          SELECT OrderNumber,
                 ItemNumber
                 OrderQty,
                 ReceivedQty
            FROM PurchDB.OrderItems
           WHERE OrderNumber > :OrderNumber;
```

The scope and strength of any lock obtained depends in part on the automatic locking mode of the target table(s). If the lock obtained is a *shared* lock, as for PUBLIC or PUBLICREAD tables, ALLBASE/SQL elevates the lock to an *exclusive* lock when you update or delete a row in the active set.

The use of lock types, lock granularities, and isolation levels is discussed in the *ALLBASE/SQL Reference Manual* .

As mentioned in the previous section, when a transaction terminates, any cursors opened during that transaction are either automatically closed, or they remain open if you are using the KEEP CURSOR option of the OPEN command. To avoid possible confusion, it is good programming practice to *always* use the CLOSE command to explicitly close any open cursors before ending a transaction with the COMMIT WORK or ROLLBACK WORK command.

When the transaction terminates, any changes made to the active set during the transaction are either *all* committed or *all* rolled back, depending on how you terminate the transaction.

Using KEEP CURSOR

Cursor operations in an application program let you manipulate data in an *active set* associated with a SELECT command. The cursor is a pointer to a row in the active set. The KEEP CURSOR option of the OPEN command lets you maintain the cursor position in an active set beyond transaction boundaries. This means you can scan and update a large table without holding locks for the duration of the entire scan. You can also design transactions that avoid holding any locks around terminal reads. In general, use the KEEP CURSOR option when you wish to release locks periodically in long or complicated transactions.

After you specify KEEP CURSOR in an OPEN command, a COMMIT WORK does not close the cursor, as it normally does. Instead, COMMIT WORK releases all locks not associated with the kept cursor and begins a new transaction while maintaining the current (kept) cursor position. This makes it possible to update tuples in a large active set, releasing locks as the cursor moves from page to page, instead of requiring you to reopen and manually reposition the cursor before the next FETCH.

Locks held on pages corresponding to the current kept cursor are either held until after the transaction ends (the default) or released depending on whether you specify WITH LOCKS or WITH NOLOCKS. (Pages held include data and system pages.)

If you use the KEEP CURSOR WITH NOLOCKS option for a cursor that involves sorting, whether through the use of a DISTINCT, GROUP BY, or ORDER BY clause, or as the result of a union or a join operation, ALLBASE/SQL does not ensure data integrity.

It is your responsibility to ensure data integrity by verifying the continued existence of a row before updating it or using it as the basis for updating some other table. For an updatable cursor, you can use either the REFETCH or SELECT command to verify the continued existence of a row. For a cursor that is non-updatable, you must use the SELECT command.

A warning (DBWARN 2056) regarding the kept cursor on a sort with no locks is generated. You *must* check for this warning if you want to detect the execution of this type of cursor operation.

KEEP CURSOR and Isolation Levels

The KEEP CURSOR option retains the current isolation level that you have specified in the BEGIN WORK command. Moreover, the exact pattern of lock retention and release for cursors opened using KEEP CURSOR WITH LOCKS depends on the current isolation level, for example:

- With the CS isolation level, KEEP CURSOR maintains locks until the next FETCH is completed. See Figure 8-2.

- With the RC isolation level, KEEP CURSOR maintains locks only until the current FETCH is completed; no locks are maintained across transactions. Therefore, KEEP CURSOR WITH LOCKS does not retain locks at the RC isolation level.

For additional information on isolation levels, refer to the *ALLBASE/SQL Reference Manual* .

KEEP CURSOR and Declaring for Update

When you DECLARE a cursor for UPDATE, SIX locks are obtained at the page level rather than share locks. There is less concurrency and less chance of deadlock because lock promotion is unnecessary. Although concurrency is reduced, throughput is often improved due to the reduction in deadlock recovery overhead.

OPEN Command Without KEEP CURSOR

Figure 8-1 shows the operation of cursors when you do *not* select the KEEP CURSOR option.
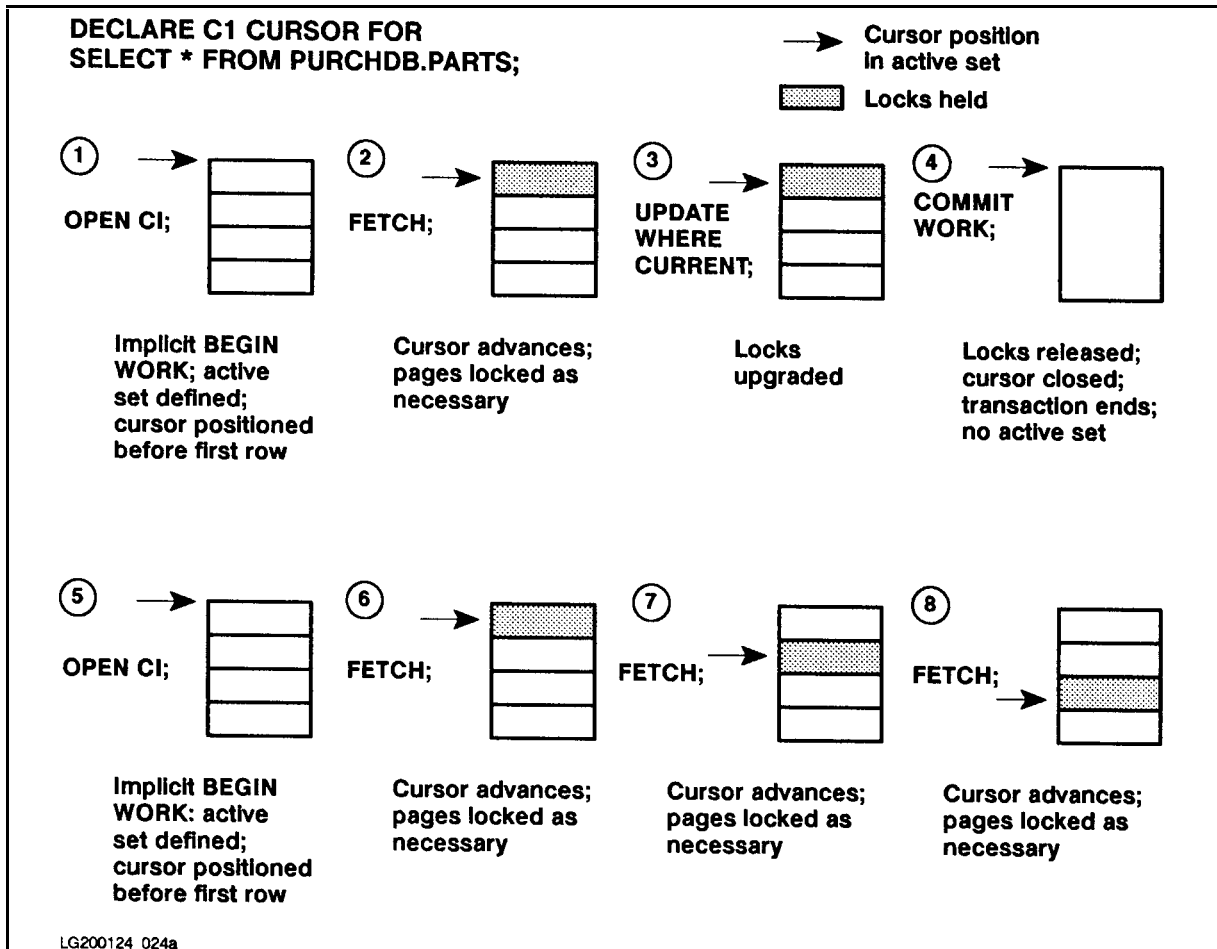
Figure 8-1. Cursor Operation without the KEEP CURSOR Feature

After the cursor is opened, successive FETCH commands advance the cursor position. Any exclusive locks acquired along the way are retained until the transaction ends. If you have selected the Cursor Stability option in the BEGIN WORK command, locks on pages that have not been updated are released when the cursor moves to a tuple on a new data page. Exclusive locks are not released until a COMMIT WORK, which also closes the cursor.

OPEN Command Using KEEP CURSOR WITH LOCKS and CS Isolation Level

The feature has the following effects:

- A COMMIT WORK command does not close the cursor. Instead, it ends the current transaction and immediately starts another one.

- When you issue a COMMIT WORK, locks associated with the cursor are not released.

- Successive FETCHES advance the cursor position, which is retained in between transactions until the cursor is explicitly closed with the CLOSE command.

- After the CLOSE command, you use an additional COMMIT WORK command. This step is *essential*. The final COMMIT after the CLOSE is necessary to end the KEEP state, release all locks associated with the cursor, and prevent a new implicit BEGIN WORK.

Figure 8-2 shows the effect of the KEEP CURSOR WITH LOCKS.
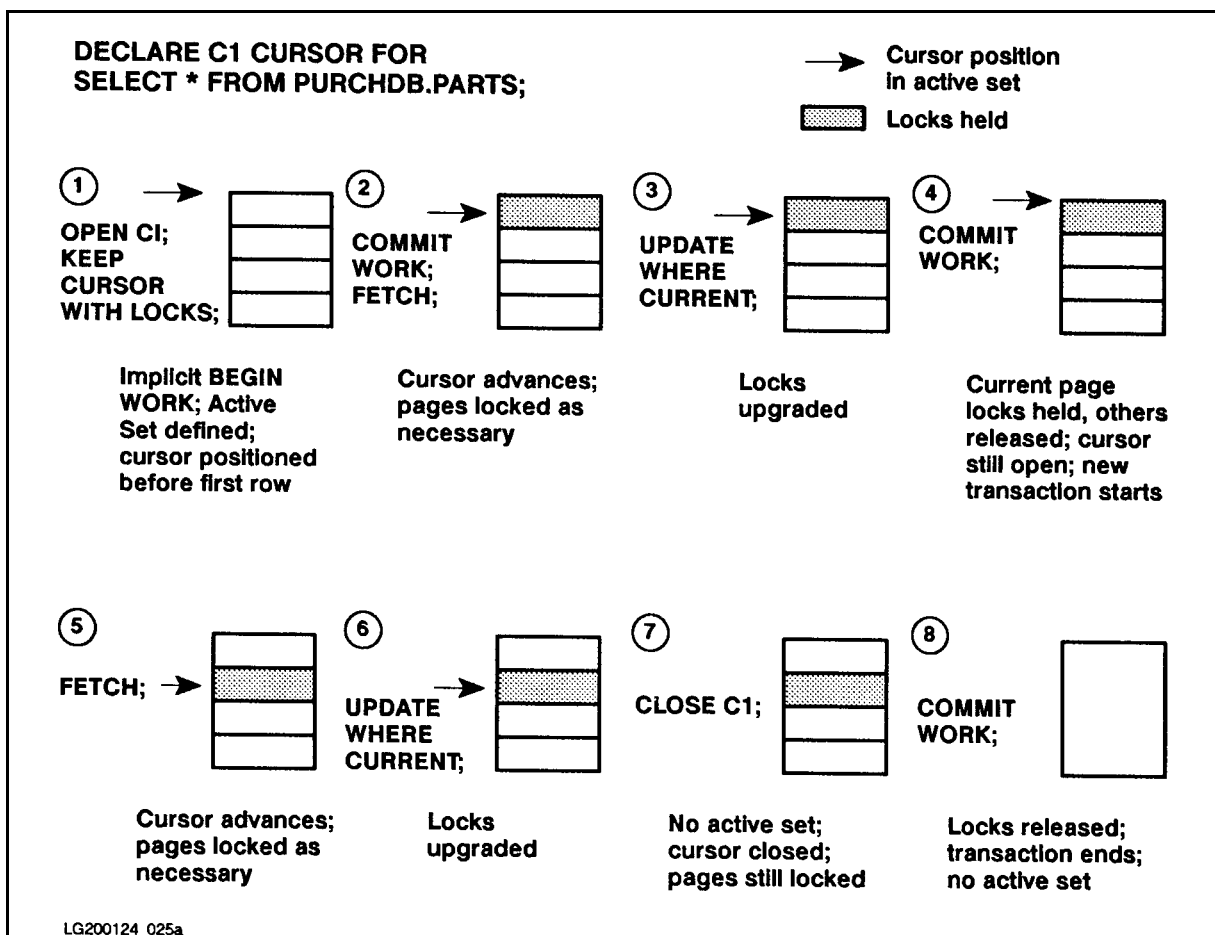
Figure 8-2. Cursor Operation Using KEEP CURSOR WITH LOCKS

OPEN Command Using KEEP CURSOR WITH NOLOCKS

The feature has the following effects:

- A COMMIT WORK command does not close the cursor. Instead, it ends the current transaction and immediately starts another one.

- When you issue a COMMIT WORK, all locks associated with the cursor are released. This means that another transaction may delete or modify the next tuple in the active set before you have the chance to FETCH it.

- Successive FETCHES advance the cursor position, which is retained in between transactions until the cursor is explicitly closed with the CLOSE command.

- After the CLOSE command, you use an additional COMMIT WORK command. This step is *essential*. The final COMMIT after the CLOSE is necessary to end the KEEP state and prevent a new implicit BEGIN WORK.

- When using KEEP CURSOR WITH NOLOCKS, be aware that data at the cursor position may be lost before the next FETCH:

  - If another transaction deletes the current row, ALLBASE/SQL will return the next row. No error message is displayed.

  - If another transaction deletes the table being accessed, the user will see the following message: `TABLE NOT FOUND  (DBERR 137)`

Figure 8-3 shows the effect of KEEP CURSOR WITH NOLOCKS.

DECLARE C1 CURSOR FOR
SELECT * FROM PURCHDB.PARTS;

→ Cursor position in active set

▨ Locks held

① OPEN CI;
KEEP CURSOR WITH NOLOCKS;

Implicit BEGIN WORK; Active Set defined; cursor positioned before first row

② COMMIT WORK; FETCH;

Cursor advances; pages locked as necessary

③ UPDATE WHERE CURRENT;

Locks upgraded

④ COMMIT WORK;

All locks released; cursor still open; new transaction starts

⑤ FETCH; →

Cursor advances; pages locked as necessary

⑥ UPDATE WHERE CURRENT;

Locks upgraded

⑦ CLOSE C1;

No active set; cursor closed; pages still locked

⑧ COMMIT WORK;

Locks released; transaction ends; no active set

LG200124_026a

Figure 8-3. Cursor Operation Using KEEP CURSOR WITH NOLOCKS

KEEP CURSOR and BEGIN WORK

■ ALLBASE/SQL automatically begins a transaction whenever you issue a command if a transaction is not already in progress. Thus, although you can code an explicit BEGIN WORK to start transactions, it is not necessary to do so unless you wish to specify an isolation level other than RR.

■ With KEEP CURSOR, an implicit BEGIN WORK follows immediately after you perform a COMMIT WORK, so if you do an explicit BEGIN WORK, ALLBASE/SQL returns an error message stating that a transaction is already in progress. If this problem should arise, re-code to eliminate the BEGIN WORK from the loop.

KEEP CURSOR and COMMIT WORK

- When the KEEP CURSOR option of the OPEN command is activated for a cursor, COMMIT WORK may or may not release locks associated with the cursor depending on the setting of the WITH LOCKS/WITH NOLOCKS option.

- COMMIT WORK does not close cursors opened with the KEEP CURSOR option. COMMIT WORK does end the previous implicit transaction and starts an implicit transaction with the same isolation level as that specified with the BEGIN WORK command.

- Remember that COMMIT WORK will still close all cursors opened *without* the KEEP CURSOR option.

KEEP CURSOR and ROLLBACK WORK

- When the KEEP CURSOR option is activated for an opened cursor, all locks are released when you ROLLBACK WORK, whether or not you have specified WITH LOCKS or WITH NOLOCKS. The position of the cursor is restored to what it was at the beginning of the transaction being rolled back. The current transaction is ended and a new transaction is implicitly started with the same isolation level as specified in the BEGIN WORK command.

- Remember that ROLLBACK WORK closes all cursors that you opened during the current transaction, unless the cursor was opened with the KEEP CURSOR option and its position saved with a COMMIT WORK immediately following the the OPEN command.

- When a cursor is opened with the KEEP CURSOR option, ROLLBACK WORK TO SavePoint is not allowed.

KEEP CURSOR and Aborted Transactions

- When a transaction is aborted by ALLBASE/SQL, the cursor position is retained, and a new transaction begins, as with ROLLBACK WORK.

- Remember that when a transaction aborts all cursors that you opened during the current transaction are closed unless the cursor was opened with the KEEP CURSOR option and its position saved with a COMMIT WORK immediately following the the OPEN command.

- The use of multiple cursors may require frequent examination of several system catalog tables. This means acquiring exclusive locks, which creates the potential for deadlock. However, the behavior of aborted transactions with KEEP CURSOR lets you create automatic deadlock handling routines. Simply repeat the operation until deadlock does not occur. The technique is shown under "Examples," below.

Writing Keep Cursor Applications

The next pages show a skeleton outline of a KEEP CURSOR application. Specific code examples appear in the following pages.

Because of the potential for deadlock, you must be careful to test for that condition frequently in applications using KEEP CURSOR. An aborted transaction results when a deadlock is encountered. (There is no need to test for deadlock following a COMMIT WORK or a BEGIN WORK command.) Use the following steps to create your code:

1. Declare all cursors to be used in the application.

2. Use a loop to test for a deadlock condition as you open all cursors that will use the KEEP CURSOR option. Start the loop with a BEGIN WORK statement that specifies the isolation level, then include a separate test for non-deadlock errors for each OPEN statement. Create an S100-SQL-STATUS-CHECK routine to display all error messages and RELEASE the DBEnvironment in the event of fatal errors. See the "Examples" section below.

3. Use the COMMIT WORK command. If you do not COMMIT at this point, an aborted transaction will roll back all the OPEN statements, and you will lose the cursor positions. The COMMIT starts a new transaction and keeps the cursor positions.

4. Use a loop to scan your data until all rows have been processed, as follows.

   ■ First, open any non-kept cursors. Do *not* include a COMMIT WORK after opening the non-kept cursors. If a deadlock is detected and the transaction aborted, the program reapplies the transaction.

   ■ Next, execute any FETCH, UPDATE WHERE CURRENT, or DELETE WHERE CURRENT commands. Be sure to test for unexpected errors and branch to S100-SQL-STATUS-CHECK to display messages and RELEASE in the event of a non-deadlock error. Again, if a deadlock is detected and the transaction aborted, the program reapplies the transaction.

   ■ At the end of the loop, include a COMMIT WORK. This will commit your data to the database, and it will close any non-kept cursors opened so far in the program. It will also start a new transaction and maintain the cursor position of all kept cursors.

   ■ Place any terminal or file I/O *after* this COMMIT, in order to prevent duplicate messages from appearing in the event of a rollback because of deadlock.

5. Once the program is finished scanning the tables, you should close all kept cursors within a final loop which tests for a deadlock condition. Once again, test for unexpected errors and branch to S100-SQL-STATUS-CHECK if necessary.

6. Execute a final COMMIT WORK to release the KEEP state.

Examples

This code is intended as a guide; you will want to customize it for your specific needs.

The code illustrates status checking techniques with emphasis on deadlock detection. The following four generalized code segments are presented:

- Using status checking routine in conjunction with the other code segments.

- Using a single kept cursor with locks.

- Using multiple cursors and cursor stability.

- Avoiding locks on terminal reads.

Common StatusCheck Procedure

```
PROCEDURE SQLStatusCheck;
BEGIN
   CASE SQLCA.SQLCODE OF

(****************************************************************)
(* Deadlock did not occur; Set DeadLockFree to TRUE            *)
(****************************************************************)
           0: DeadLockFree := TRUE;

(****************************************************************)
(* Deadlock occurred; set DeadLockFree to FALSE.              *)
(* Exit status checking routine without displaying a message. *)
(****************************************************************)
      -14024: BEGIN
               DeadLockFree := FALSE;
(****************************************************************)
(* If your program monopolizes CPU time by repeatedly         *)
(* reapplying a transaction, you could include a call          *)
(* to the HP-UX 'sleep' utility at this point.                *)
(****************************************************************)
               END;

(****************************************************************)
(* No more rows found; Set EndOfScan-Flag to EndOfScan.       *)
(* Exit status checking routine without displaying a message. *)
(****************************************************************)
         100: EndOfScan := TRUE;
```

```
(*****************************************************************)
(* For other fatal errors:                                     *)
(*   PERFORM S200-SQLEXPLAIN to display messages               *)
(*   RELEASE the DBE                                           *)
(*   Stop the program                                         *)
(*                                                            *)
(* Some errors which could be considered fatal are:            *)
(*     -3040   DBA issued a STOP DBE command                   *)
(*     -3043   DBA issued a terminate user command             *)
(*    -14046   log full error                                 *)
(*    -14047   system clock/timestamp error                   *)
(*    -14074   DBCore internal error                          *)
(*    -14075   DBCore internal error                          *)
(*    -15048   DBCore internal error                          *)
(*****************************************************************)
      OTHERWISE
              REPEAT
                 EXEC SQL SQLEXPLAIN :SQLMessage;
                 writeln(SQLMessage);
              UNTIL SQLCA.SQLCODE = 0;

              EXEC SQL RELEASE;

              halt;

   END;  (* CASE Statement *)
END;    (* Procedure SQLStatusCheck *)
```

Single Cursor WITH LOCKS

```
(****************************************************************)
(* Declare cursor C1.                                          *)
(****************************************************************)
    EXEC SQL DECLARE C1 CURSOR FOR
              SELECT PartName, SalesPrice FROM PurchDB.Parts
              WHERE SalesPrice > 500.00;

(****************************************************************)
(* Open cursor C1 using KEEP CURSOR WITH LOCKS option,         *)
(* testing for deadlocks.                                      *)
(****************************************************************)
    DeadLockFree := FALSE;
    REPEAT
       EXEC SQL OPEN C1 KEEP CURSOR WITH LOCKS;
       SQLStatusCheck;
    UNTIL DeadLockFree;

(****************************************************************)
(* COMMIT WORK in order to preserve initial cursor position.  *)
(****************************************************************)
     EXEC SQL COMMIT WORK;
     SQLStatusCheck;

(****************************************************************)
(* BULK FETCH qualifying rows from the Parts table using      *)
(* cursor C1 until there is no more data, testing for         *)
(* deadlocks.                                                  *)
(****************************************************************)
    EndOfScan := FALSE;
    REPEAT
       DeadLockFree := FALSE;
       REPEAT
          EXEC SQL BULK FETCH C1 INTO :PriceList, 1, 20;
          SQLStatusCheck;
       UNTIL DeadLockFree OR EndOfScan;

       IF DeadLockFree
          BEGIN

(****************************************************************)
(* Execute COMMIT WORK to release all page locks held by      *)
(* cursor C1 except the current page.                         *)
(****************************************************************)
          EXEC SQL COMMIT WORK;
          SQLStatusCheck;
```

```
(******************************************************************)
(* Display qualifying rows.  SQLERRD[3] contains the actual    *)
(* number of qualified rows.  BUFFEREND contains the maximum   *)
(* number of rows declared in the buffer which receives data   *)
(* from the BULK FETCH command.                                *)
(******************************************************************)
        IF SQLERRD[3] > BUFFEREND THEN
            NUMROWS := BUFFEREND
        ELSE
            NUMROWS := SQLERRD[3];

        FOR i := 1 TO NUMROWS DO
            BEGIN
            writeln('    Part Name: ',PriceList[i].PartName);
            writeln('  Sales Price: ',PriceList[i].SalesPrice);
            writeln;
            END;
        END;
   UNTIL EndOfScan;


(******************************************************************)
(* CLOSE cursor C1, testing for deadlocks.                     *)
(******************************************************************)
   DeadLockFree := FALSE;
   REPEAT
      EXEC SQL CLOSE C1;
      SQLStatusCheck;
   UNTIL DeadLockFree;


(******************************************************************)
(* Execute final COMMIT WORK to release all locks held by      *)
(* cursor C1.                                                  *)
(******************************************************************)
   EXEC SQL COMMIT WORK;
   SQLStatusCheck;
```

Multiple Cursors and Cursor Stability

```
(*****************************************************************)
(* Declare cursor C1 and cursor C2.                            *)
(*****************************************************************)
    EXEC SQL DECLARE C1 CURSOR FOR
            SELECT BranchNo FROM Tellers WHERE TellerNo > 15000
            FOR UPDATE OF Status;

    EXEC SQL DECLARE C2 CURSOR FOR
            SELECT BranchNo FROM Branches
            FOR UPDATE OF Credit;

(*****************************************************************)
(* Open cursor C1 using KEEP CURSOR WITH LOCKS option,         *)
(* testing for deadlocks.  Use an explicit BEGIN WORK CS       *)
(* command in the loop to ensure that ALLBASE/SQL will use     *)
(* the CURSOR STABILITY isolation level if a deadlock occurs.  *)
(*****************************************************************)
    DeadLockFree := FALSE;
    REPEAT
       EXEC SQL BEGIN WORK CS;
       IF SQLCA.SQLCODE = 0 THEN
          EXEC SQL OPEN C1 KEEP CURSOR WITH LOCKS;
       SQLStatusCheck;
    UNTIL DeadLockFree;

(*****************************************************************)
(* COMMIT WORK in order to preserve initial cursor position.   *)
(*****************************************************************)
    EXEC SQL COMMIT WORK;
    SQLStatusCheck;

(*****************************************************************)
(* FETCH and UPDATE data in qualifying rows of the Tellers     *)
(* table and Branches table using cursors C1 and C2 until      *)
(* no more rows are found.                                     *)
(*****************************************************************)
    EndOfScan := FALSE;
    REPEAT

(*****************************************************************)
(* FETCH data from Tellers table using cursor C1.              *)
(*****************************************************************)
        EXEC SQL FETCH C1 INTO :HostBranchNo1;

(*****************************************************************)
(* OPEN cursor C2 (without the KEEP CURSOR option).            *)
(*****************************************************************)
```

```
        IF SQLCODE = 0 THEN
            BEGIN
            EXEC SQL OPEN C2;
(***************************************************************)
(* For each qualifying row in the Tellers table:              *)
(*   FETCH and UPDATE rows in the Branches table using cursor *)
(*   C2 until no more rows are found, testing for deadlocks.  *)
(***************************************************************)
            IF SQLCODE = 0 THEN
                BEGIN
                DeadLockFree := TRUE;

(***************************************************************)
(* FETCH data from the Branches table using cursor C2.        *)
(***************************************************************)
                REPEAT
                    EXEC SQL FETCH C2 INTO :HostBranchNo2;

(***************************************************************)
(* Update Branches table if:                                  *)
(*   FETCH was successful (SQLCODE = 0), and                  *)
(*   Teller.BranchNo = Branches.BranchNo                      *)
(***************************************************************)
                    IF SQLCODE  0 THEN
                        SQLStatusCheck
                    ELSE
                        IF HostBranchNo1 = HostBranchNo2 THEN
                            BEGIN
                            EXEC SQL UPDATE Branches
                                    SET Credit = Credit * 0.005
                                    WHERE CURRENT OF C2;
                            SQLStatusCheck;
                            END;
                UNTIL EndOfScan OR NOT DeadLockFree;

            IF EndOfScan THEN
                BEGIN
                EndOfScan := FALSE;
                EXEC SQL CLOSE C2;

(***************************************************************)
(* After successfully completing the FETCH and UPDATE of data *)
(* in qualifying rows of the Branches table using cursor C2,  *)
(* UPDATE the Tellers table using cursor C1.                  *)
(***************************************************************)
                IF SQLCODE = 0 THEN
                    BEGIN
                    EXEC SQL UPDATE TELLERS
                            SET Status = :NewStatus
                            WHERE CURRENT OF C1;
```

```
(***************************************************************)
(* Execute COMMIT WORK to:                                   *)
(*   Save UPDATEs to Branches table using cursor C2          *)
(*   Release all page locks held by cursor C2                *)
(*   Save UPDATES to Tellers table using cursor C1           *)
(*   Release pages locked by cursor C1 except current page   *)
(***************************************************************)
                 IF SQLCODE = 0 THEN
                     EXEC SQL COMMIT WORK;
                 END;
             END;
          END;
       END;

    SQLStatusCheck;

  UNTIL EndOfScan;


(***************************************************************)
(* CLOSE cursor C1, testing for deadlocks.                   *)
(***************************************************************)
    DeadLockFree := FALSE;
    REPEAT
       EXEC SQL CLOSE C1;
       SQLStatusCheck;
    UNTIL DeadLockFree;


(***************************************************************)
(* Execute final COMMIT WORK to release all locks held by    *)
(* cursor C1.                                                *)
(***************************************************************)
    EXEC SQL COMMIT WORK;
    SQLStatusCheck;
```

Avoiding Locks on Terminal Reads

```
(***************************************************************)
(* Declare cursor C1.                                        *)
(***************************************************************)
    EXEC SQL DECLARE C1 CURSOR FOR
             SELECT PartName, SalesPrice FROM PurchDB.Parts;


(***************************************************************)
(* Open cursor C1 using KEEP CURSOR WITH NOLOCKS option,     *)
(* testing for deadlocks.                                    *)
(***************************************************************)
    DeadLockFree := FALSE;
    REPEAT
       EXEC SQL OPEN C1 KEEP CURSOR WITH NOLOCKS;
       SQLStatusCheck;
    UNTIL DeadLockFree;
```

```
(*****************************************************************)
(* COMMIT WORK in order to preserve initial cursor position.  *)
(*****************************************************************)
   EXEC SQL COMMIT WORK;
   SQLStatusCheck;


(*****************************************************************)
(* FETCH and DISPLAY data in qualifying rows of the Parts      *)
(* table using cursors C1 until no more rows are found,        *)
(* testing for deadlocks.                                      *)
(*****************************************************************)
   EndOfScan := FALSE;
   REPEAT


(*****************************************************************)
(* FETCH data from the Parts table using cursor C1, testing    *)
(* for deadlocks.                                              *)
(*****************************************************************)
      DeadLockFree := FALSE;
      REPEAT
         EXEC SQL FETCH C1 INTO :PartNumber, :PresentPrice;
         SQLStatusCheck;
      UNTIL DeadlockFree;


(*****************************************************************)
(* Execute COMMIT WORK to release all page locks held by       *)
(* cursor C1.                                                  *)
(*****************************************************************)
      EXEC SQL COMMIT WORK;
      SQLStatusCheck;
(*****************************************************************)
(* Display values from Parts.PartNumber and Parts.SalesPrice, *)
(* and prompt user for a new sales price.                      *)
(*****************************************************************)
      writeln('   Part Number: ',PartNumber);
      writeln('   Sales Price: ',PresentPrice);
      prompt ('Enter new sales price: ');
      readln (NewPrice);


(*****************************************************************)
(* Re-select data from the Parts table and verify that the     *)
(* SalesPrice has not changed.  If unchanged, update the row   *)
(* with the value in NewPrice.                                 *)
(*****************************************************************)
      DeadLockFree := FALSE;
      REPEAT
```

```
(****************************************************************)
(* Re-select data from the Parts table.                       *)
(****************************************************************)
      EXEC SQL SELECT SalesPrice INTO :SalesPrice
               FROM PurchDB.Parts WHERE PartNumber = :PartNumber;
      SQLStatusCheck;

      IF EndOfScan THEN
          writeln('Part number no longer in database. Not updated')
      ELSE
          IF SalesPrice NOT = PresentPrice
              writeln('Current price has changed. Not updated')
          ELSE

(****************************************************************)
(* If Parts.SalesPrice has not changed, update the qualifying *)
(* row with the value in NewPrice.                            *)
(****************************************************************)
             BEGIN
             EXEC SQL UPDATE PurchDB.Parts
                      SET SalesPrice = :NewPrice
                      WHERE PartNumber = :PartNumber;
             SQLStatusCheck;
             END;

      UNTIL DeadLockFree;
   UNTIL EndOfScan;

(****************************************************************)
(* CLOSE cursor C1, testing for deadlocks.                    *)
(****************************************************************)
   DeadLockFree := FALSE;
   REPEAT
      EXEC SQL CLOSE C1;
      SQLStatusCheck;
   UNTIL DeadLockFree;

(****************************************************************)
(* Execute final COMMIT WORK to release all locks held by     *)
(* cursor C1.                                                 *)
(****************************************************************)
   EXEC SQL COMMIT WORK;
   SQLStatusCheck;
```

Program Using UPDATE WHERE CURRENT

The flow chart in Figure 8-4 summarizes the functionality of program pasex8. This program uses a cursor and the UPDATE WHERE CURRENT command to update column ReceivedQty in table PurchDB.OrderItems. The runtime dialog for pasex8 appears in Figure 8-5, and the source code in Figure 8-6.

The program first executes procedure DeclareCursor (26), which contains the DECLARE CURSOR command (7). This command is a preprocessor directive and is not executed at run time. At run time, procedure DeclareCursor only displays the message, Declare Cursor. The DECLARE CURSOR command defines a cursor named OrderReview. The cursor is associated with a SELECT command that retrieves the following columns for all rows in table PurchDB.OrderItems having a specific order number but no null values in column VendPartNumber:

```
OrderNumber (defined NOT NULL)
ItemNumber  (defined NOT NULL)
VendPartNumber
ReceivedQty
```

Cursor OrderReview has a FOR UPDATE clause naming column ReceivedQty to allow the user to change the value in this column.

To establish a DBE session, program pasex8 executes function ConnectDBE (27). This function evaluates to TRUE when the CONNECT command (1) for the sample DBEnvironment, PartsDBE, is successfully executed.

The program then executes procedure FetchUpdate until the Done flag is set to TRUE (28).

Procedure FetchUpdate

Procedure FetchUpdate prompts for an order number or a 0 (17). When the user enters a 0, the Done flag is set to TRUE (25), and the program terminates. When the user enters an order number, the program begins a transaction by executing procedure BeginTransaction (18), which executes the BEGIN WORK command (3).

Cursor OrderReview is then opened by invoking function OpenCursor (19). This function, which executes the OPEN command (8), evaluates to TRUE when the command is successful.

A row at a time is retrieved and optionally updated until the DoFetch flag is set to FALSE (20). This flag becomes false when:

- The FETCH command fails; this command fails when no rows qualify for the active set, when the last row has already been fetched, or when ALLBASE/SQL cannot execute this command for some other reason.

- The program user wants to stop reviewing rows from the active set.

The FETCH command (21) names an indicator variable for ReceivedQty, the only column in the query result that may contain a null value. If the FETCH command is successful, the program executes procedure DisplayUpdate (22) to display the current row and optionally update it.

Procedure DisplayUpdate

Procedure DisplayUpdate executes procedure DisplayRow (10) to display the current row. The user is asked whether he wants to update the current ReceivedQty value (11). If so, the user is prompted for a new value. The value accepted is used in an UPDATE WHERE CURRENT command (12). If the user entered a 0, a null value is assigned to this column.

The program then asks whether to FETCH another row (13). If so, the FETCH command is re-executed. If not, the program asks whether the user wants to make permanent any updates he may have made in the active set (14). To keep any row changes, the program executes procedure CommitWork (16), which executes the COMMIT WORK command (4). To undo any row changes, the program executes procedure RollBackWork (15), which executes the ROLLBACK WORK command (5).

The COMMIT WORK command is also executed when ALLBASE/SQL sets SQLCA.SQLCODE to 100 following execution of the FETCH command (23). SQLCA.SQLCODE is set to 100 when no rows qualify for the active set or when the last row has already been fetched. If the FETCH command fails for some other reason, the ROLLBACK WORK command is executed instead (24).

Before any COMMIT WORK or ROLLBACK WORK command is executed, cursor OrderReview is closed (9). Although the cursor is automatically closed whenever a transaction is terminated, it is good programming practice to use the CLOSE command to close open cursors prior to terminating transactions.

When the program user enters a 0 in response to the order number prompt (17), the program terminates by executing procedure TerminateProgram (29), which executes the RELEASE command (2).

Explicit status checking is used throughout this program. After each embedded SQL command is executed, SQLCA.SQLCode is checked. If SQLCode is less than 0, the program executes procdure SQLStatusCheck, which executes the SQLEXPLAIN command.
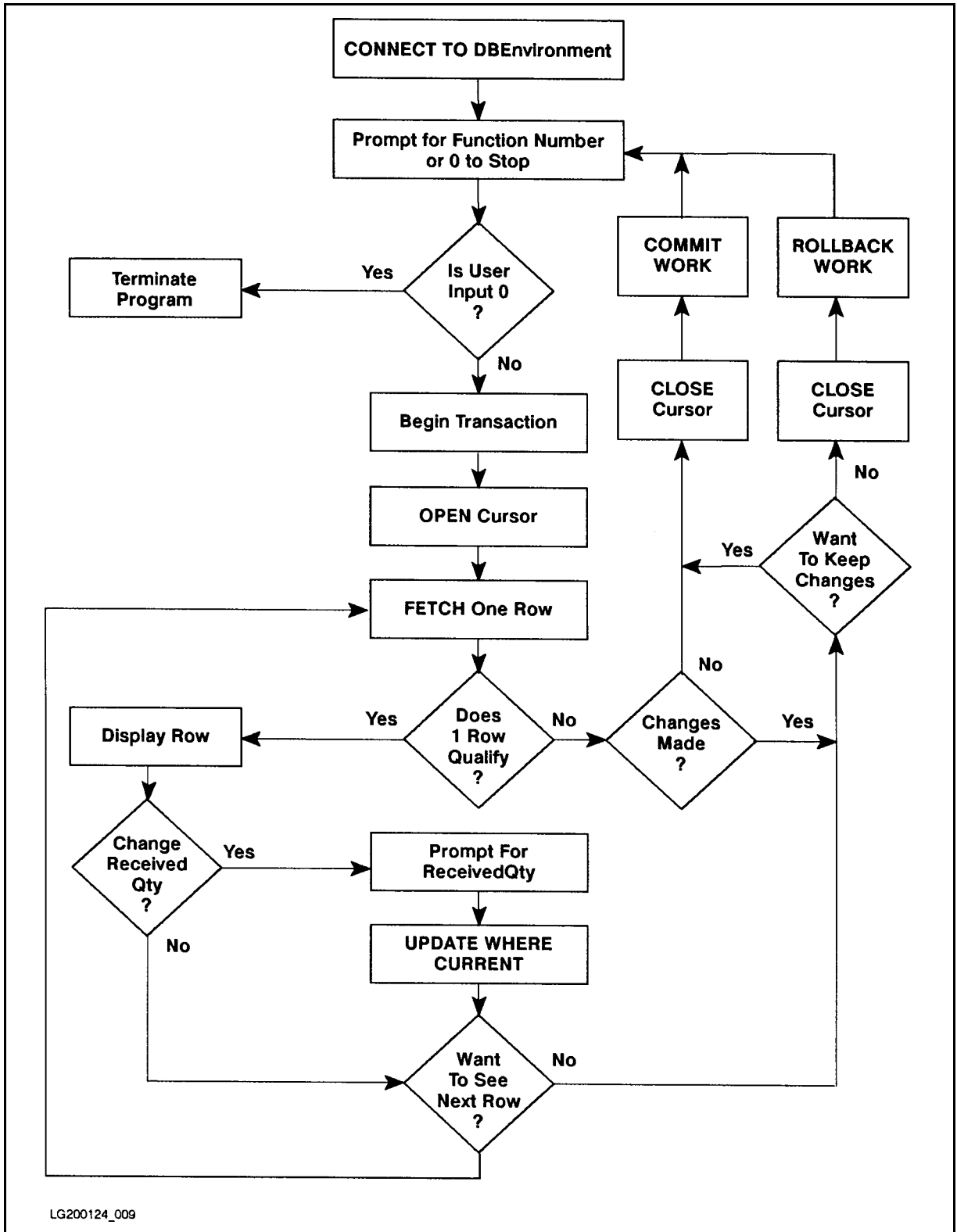
Figure 8-4. Flow Chart of Program pasex8

LG200124_009

```
Program to UPDATE OrderItems Table via a CURSOR - pasex8

Event List:
  Connect to PartsDBE
  Prompt for Order Number
  Begin Work
  Open Cursor
  FETCH a row
  Display the retrieved row
  Prompt for new Received Quantity
  Update row within OrderItems table
  FETCH the next row, if any, with the same Order Number
  Repeat the above five steps until there are no more rows
  Close Cursor
  End Transaction
  Repeat the above eleven steps until user enters 0
  Release PartsDBE

Declare Cursor
Connect to PartsDBE

Enter OrderNumber or 0 to STOP >  30520

Begin Work
Open Cursor

  OrderNumber:            30520
  ItemNumber:                1
  VendPartNumber:  9375
  ReceivedQty                9

Do you want to change ReceivedQty (Y/N)? >  n

Do you want to see another row (Y/N)? >  y

  OrderNumber:            30520
  ItemNumber:                2
  VendPartNumber:  9105
  ReceivedQty is             3
```

Figure 8-5. Runtime Dialog of Program pasex8

```
Do you want to change ReceivedQty (Y/N)? >  y

Enter New ReceivedQty (O for NULL)>  15
Update PurchDB.OrderItems Table

Do you want to see another row (Y/N)? >  y

  OrderNumber:          30520
  ItemNumber:              3
  VendPartNumber:  9135
  ReceivedQty              3

Do you want to change ReceivedQty (Y/N)? >  n

Do you want to see another row (Y/N)? >  y

Row not found or no more rows
Want to save your changes (Y/N)? > y

Close Cursor
Commit Work
   1 row(s) changed.

Enter OrderNumber or 0 to STOP >  30510

Begin Work
Open Cursor

  OrderNumber:          30510
  ItemNumber:              1
  VendPartNumber:  1001
  ReceivedQty              3

Do you want to change ReceivedQty (Y/N)? >  n

Do you want to see another row (Y/N)? >  n

Close Cursor
Commit Work

Enter OrderNumber or 0 to STOP >  0
```

Figure 8-5. Runtime Dialog of Program pasex8 (page 2 of 2)

```
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* This program illustrates the use of UPDATE WHERE CURRENT        *)
(* with a Cursor to update a single row at a time.                 *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

Program pasex8(input, output);

const
    OK            =      0;
    NotFound      =    100;
    DeadLock      = -14024;

var
          (* Begin Host Variable Declarations *)
    EXEC SQL BEGIN DECLARE SECTION;
    OrderNumber       : integer;
    ItemNumber        : integer;
    VendPartNumber    : packed array [1..16] of char;
    ReceivedQty       : SmallInt;
    ReceivedQtyInd    : SqlInd;
    SQLMessage        : packed array[1..132] of char;
    EXEC SQL END DECLARE SECTION;
          (* End Host Variable Declarations *)

    SQLCA : SQLCA_type;   (* SQL Communication Area *)

    Abort             : boolean;
    Done              : boolean;
    DoFetch           : boolean;

    Response          : packed array [1..3] of char;
    RowCounter        : integer;

procedure TerminateProgram; forward;

procedure SQLStatusCheck;  (* Procedure to Display Error Messages *)
begin

Abort := FALSE;
if SQLCA.SQLCODE < DeadLock then Abort := TRUE;
```

Figure 8-6. Program pasex8: Using UPDATE WHERE CURRENT

```
repeat
EXEC SQL SQLEXPLAIN :SQLMessage;
writeln(SQLMessage);
until SQLCA.SQLCODE = 0;
if Abort then TerminateProgram;
end;  (* End SQLStatusCheck Procedure *)
$PAGE $
function ConnectDBE: boolean;  (* Function to Connect to PartsDBE *)
begin

writeln('Connect to PartsDBE');
EXEC SQL CONNECT TO '../sampledb/PartsDBE';                               1

ConnectDBE := TRUE;
if SQLCA.SQLCODE <> OK then
  begin

  ConnectDBE := FALSE;
  SQLStatusCheck;

  end;  (* End if *)
end;  (* End of ConnectDBE Function *)

procedure TerminateProgram;   (* Procedure to Release PartsDBE *)
begin

EXEC SQL RELEASE;                                                         2
Done := TRUE;

end;  (* End TerminateProgram Procedure *)
$PAGE $
procedure BeginTransaction;                (* Procedure to Begin Work *)
begin
writeln;
writeln('Begin Work');
EXEC SQL BEGIN WORK;                                                      3
if SQLCA.SQLCODE <> OK then
  begin
  SQLStatusCheck;
  TerminateProgram;
  end;
end;  (* End BeginTransaction Procedure *)
```

Figure 8-6. Program pasex8: Using UPDATE WHERE CURRENT (page 2 of 7)

```
procedure CommitWork;  (* Procedure to Commit Work *)
begin

writeln('Commit Work');
EXEC SQL COMMIT WORK;                                              4
if SQLCA.SQLCODE <> OK then
  begin
  SQLStatusCheck;
  TerminateProgram;
  end;


end;  (* End CommitWork Procedure *)



procedure RollBackWork;  (* Procedure to RollBack Work *)
begin

writeln('Rollback Work');
EXEC SQL ROLLBACK WORK;                                           5
if SQLCA.SQLCODE <> OK then
  begin
  SQLStatusCheck;
  TerminateProgram;
  end;


end;  (* End RollBackWork Procedure *)

procedure DisplayRow;    (* Procedure to Display OrderItems Rows *)  6
begin

writeln;
writeln('  OrderNumber:     ', OrderNumber);
writeln('  ItemNumber:      ', ItemNumber);
writeln('  VendPartNumber:  ', VendPartNumber);
if ReceivedQtyInd < 0 then
  writeln('  ReceivedQty is NULL')
else
  writeln('  ReceivedQty      ', ReceivedQty);


end;  (* End of DisplayRow *)
$PAGE $


Procedure DeclareCursor;
begin
```

Figure 8-6. Program pasex8: Using UPDATE WHERE CURRENT (page 3 of 7)

```
writeln('Declare Cursor');
EXEC SQL DECLARE OrderReview                                                    [7]
            CURSOR FOR
            SELECT OrderNumber,
                   ItemNumber,
                   VendPartNumber,
                   ReceivedQty
              FROM PurchDB.OrderItems
             WHERE OrderNumber = :OrderNumber
               AND VendPartNumber IS NOT NULL
     FOR UPDATE OF ReceivedQty;
end;  (* End of DeclareCursor Procedure *)
function OpenCursor: boolean;    (* Function to Open Cursor *)
begin
writeln('Open Cursor');
EXEC SQL OPEN OrderReview;                                                      [8]
if SQLCA.SQLCODE <> OK then
  begin
  OpenCursor := FALSE;
  SQLStatusCheck;
  RollBackWork;
  end
else
  OpenCursor := TRUE;
end;  (* End OpenCursor Function *)
procedure CloseCursor;     (* Procedure to Close Cursor *)
begin
writeln;
writeln('Close Cursor');
EXEC SQL CLOSE OrderReview;                                                     [9]
if SQLCA.SQLCODE <> OK then
  begin
  SQLStatusCheck;
  TerminateProgram;
  end;
end;  (* End CloseCursor Procedure *)
$PAGE $
procedure DisplayUpdate;   (* Display & Update row in Parts Table *)
begin
DisplayRow;                                                                    [10]
writeln;                                                                       [11]
prompt('Do you want to change ReceivedQty (Y/N)? >  ');
readln(Response);
if Response[1] in ['Y','y'] then
```

Figure 8-6. Program pasex8: Using UPDATE WHERE CURRENT (page 4 of 7)

```
      begin
      writeln;
      prompt('Enter New ReceivedQty (0 for NULL)>  ');
      readln(ReceivedQty);
      writeln('Update PurchDB.OrderItems Table');
      if ReceivedQty = 0 then ReceivedQtyInd := -1
        else ReceivedQtyInd := 0;
        EXEC SQL UPDATE PurchDB.OrderItems
                    SET ReceivedQty = :ReceivedQty :ReceivedQtyInd     12
                  WHERE CURRENT OF OrderReview;
      if SQLCA.SQLCODE <> OK then SQLStatusCheck
      else RowCounter := RowCounter+1;
      end;
  writeln;
  prompt('Do you want to see another row (Y/N)? >  ');               13
  readln(Response);
  if Response[1] in ['N','n'] then
  begin
  if RowCounter > 0 then
    begin
    writeln;
    prompt('Do you want to save any changes you made (Y/N)?> ');     14
    readln(Response);
    if Response[1] in ['N','n'] then
      begin
      CloseCursor;
      RollBackWork;                                                  15
      DoFetch := FALSE;
      end
    else
      begin
      CloseCursor;
      CommitWork;                                                    16
      writeln(RowCounter, ' row(s) changed.');
      DoFetch := FALSE;
      end;
    end;     (* end if RowCounter *)
    if RowCounter = 0 then
      begin
      CloseCursor;
      CommitWork;
      DoFetch := FALSE;
      end;
  end;
```

Figure 8-6. Program pasex8: Using UPDATE WHERE CURRENT (page 5 of 7)

```
end;    (* End of DisplayUpdate Procedure *)
$PAGE$
procedure FetchUpdate;
begin
RowCounter := 0;
writeln;
prompt('Enter OrderNumber or 0 to STOP >  ');             [17]
readln(OrderNumber);
if OrderNumber <> 0 then
  begin
  BeginTransaction;                                        [18]
  if OpenCursor then                                       [19]
    begin
    DoFetch := TRUE;
    while DoFetch = TRUE do                                [20]
      begin
      EXEC SQL FETCH OrderReview                           [21]
                INTO :OrderNumber,
                     :ItemNumber,
                     :VendPartNumber,
                     :ReceivedQty :ReceivedQtyInd;
      case SQLCA.SQLCODE of
      OK        : DisplayUpdate;                           [22]
      NotFound : begin
                    DoFetch := FALSE;
                    writeln;
                    writeln('Row not found or no more rows');
                    if RowCounter > 0 then
                      begin
                      prompt('Want to save your changes (Y/N)? > ');
                      readln(Response);
                      if Response[1] in ['N','n'] then
                        begin
                        CloseCursor;
                        RollBackWork;
                        end
                      else
                        begin
                        CloseCursor;
                        CommitWork;                        [23]
                        writeln(RowCounter ,' row(s) changed.');
                        end;
                  end;
                if RowCoounter = 0 then
                  begin
```

Figure 8-6. Program pasex8: Using UPDATE WHERE CURRENT (page 6 of 7)

```
                        CloseCursor;
                        CommitWork;
                        end;
                    end;
        otherwise   begin
                        DoFetch := FALSE;
                        SQLStatusCheck;
                        CloseCursor;
                        RollbackWork;                           [24]
                    end;
        end; (* case *)
        end;    (* while *)
        end; (* if OpenCursor *)
    end (* end if OrderNumber *)
    else
      Done := TRUE;                                            [25]
end;  (* End of FetchUpdate Procedure *)
$PAGE $
begin  (* Beginning of Program *)
writeln('Program to UPDATE OrderItems Table via a CURSOR - pasex8');
writeln;
writeln('Event List:');
writeln('  Connect to PartsDBE');
writeln('  Prompt for Order Number');
writeln('  Begin Work');
writeln('  Open Cursor');
writeln('  FETCH a row');
writeln('  Display the retrieved row');
writeln('  Prompt for new Received Quantity');
writeln('  Update row within OrderItems table');
writeln('  FETCH the next row, if any, with the same Order Number');
writeln('  Repeat the above five steps until there are no more rows');
writeln('  Close Cursor');
writeln('  End Transaction');
writeln('  Repeat the above eleven steps until user enters 0');
writeln('  Release PartsDBE');
writeln;
DeclareCursor;                                                 [26]
if ConnectDBE then                                            [27]
  begin
  Done := FALSE;
  repeat                                                      [28]
    FetchUpdate
  until Done;
  end;
TerminateProgram;                                             [29]
end.  (* End of Program *)
```

Figure 8-6. Program pasex8: Using UPDATE WHERE CURRENT (page 7 of 7)

9

Bulk Table Processing

BULK table processing is the programming technique you use to SELECT, FETCH, or INSERT *multiple* rows at a time. This chapter describes the following aspects of BULK processing:

- Variables Used in BULK Processing.

- SQL BULK Commands.

- Transaction Management for BULK Operations.

- Program Using BULK Processing.

Variables Used in BULK Processing

Rows are retrieved into or inserted from host variables declared as an array of records. Any column that may contain a null value *must* have an indicator variable immediately following the declaration for the column in the array. For example, the indicator variable for Column2Name is Column2IndVar, as shown in the following syntax:

```
ArrayName                  : packed or unpacked array [1..n]
of packed record
  Column1Name              : Valid data type;
  Column2Name              : Valid data type;
  Col2IndVar               : SqlInd;
  ⋮
  ColumnName               : Valid data type;
end;
```

You reference the name of the array in the BULK SQL command:

```
EXEC SQL BEGIN DECLARE SECTION;
PartsArray              : packed array[1..25]
 of packed record
   PartNumber           : packed array [1..16] of char;
   PartName             : packed array [1..30] of char;
   PartNameInd          : SqlInd;
 end;
SalesPrice              : longreal;
  .
  .
EXEC SQL BULK SELECT  PartNumber, PartName
                 INTO :PartsArray
                 FROM  PurchDB.Parts
                WHERE  SalesPrice < :SalesPrice;
```

Two additional host variables may be specified in conjunction with the array:

- A StartIndex variable: a SMALLINT or INTEGER variable that specifies an array
  subscript. The subscript identifies where in the array ALLBASE/SQL should store the
  first row in a group of rows retrieved. In the case of an INSERT operation, the subscript
  identifies where in the array the first row to be inserted is stored. If not specified, the
  assumed subscript is one.

- A NumberOfRows variable: a SMALLINT or INTEGER variable that indicates to
  ALLBASE/SQL how many rows to transfer into or take from the array, starting at the
  array record designated by StartIndex. If not specified for an INSERT operation, the
  assumed number of rows is the number of records in the array from the StartIndex to the
  end of the array. If not specified for a SELECT operation, the assumed number of rows is
  the smaller of two values: the number of records in the array or the number of rows in the
  query result. NumberOfRows can be specified only if you specify the StartIndex variable.

In the BULK SELECT example shown earlier, these two variables would be declared and
referenced as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
PartsArray              : packed array[1..25]
 of packed record
   PartNumber           : packed array [1..16] of char;
   PartName             : packed array [1..30] of char;
   PartNameInd          : SqlInd;
 end;
SalesPrice              : longreal;
StartIndex              : SmallInt;
NumberOfRows            : SmallInt;
  .
  .
EXEC SQL BULK SELECT  PartNumber, PartName
                 INTO :PartArray,
                      :StartIndex,
                      :NumberOfRows
                 FROM  PurchDB.Parts
                WHERE  SalesPrice < :SalesPrice;
```

Note that StartIndex and NumberOfRows must be referenced in that order and immediately following the array reference.

SQL Bulk Commands

The SQL commands used for BULK table processing are:

```
BULK SELECT
BULK FETCH
BULK INSERT
```

BULK SELECT

The BULK SELECT command is useful when the maximum number of rows in the query result is known at programming time and when the query result is not too large. For example, this command might be used in an application that retrieves a query result containing a row for each month of the year.

The syntax of the BULK SELECT command is:

```
BULK SELECT SelectList
       INTO ArrayName [,StartIndex [,NumberOfRows]]
       FROM TableNames
      WHERE SearchCondition1
   GROUP BY ColumnName
     HAVING SearchCondition2
   ORDER BY ColumnID
```

Remember, the WHERE, GROUP BY, HAVING, and ORDER BY clauses are optional. Note that the order of the select list items *must* match the order of the corresponding host variables in the array.

In the following example, parts are counted at one of three frequencies or cycles: 30, 60, or 90 days. The host variable array needs to contain only three records, since the query result will never exceed three rows.

```
EXEC SQL BEGIN DECLARE SECTION;
PartsPerCycle               : packed array[1..3]
  of packed record
    CountCycle              : SmallInt;
    PartCount               : integer;
  end;
.
.
.
EXEC SQL BULK SELECT   CountCycle, COUNT(PartNumber)
                INTO :PartsPerCycle
                FROM  PurchDB.Inventory;
```

The query result is a three row table that describes how many parts are counted per count cycle.

Multiple query results can be retrieved into the same host variable array by using StartIndex and NumberOfRows values and executing a BULK SELECT command multiple times:

```
EXEC SQL BEGIN DECLARE SECTION;
PartsPerCycle               : packed array[1..15]
  of packed record
    CountCycle              : SmallInt;
    PartCount               : integer;
  end;
StartIndex                  : SmallInt;
NumberOfRows                : SmallInt;
LowBinNumber                : packed array [1..16] of char;
HighBinNumber               : packed array [1..16] of char;
.
.
.
EXEC SQL END DECLARE SECTION;
LessThanFive                : boolean;
.
.
.

procedure DisplayRows;
var
  i : integer;
begin

for i = 1 to (StartIndex  - 1) do
with PartsPerCycle[i] do
  begin
  writeln('CountCycle:  ', Countcycle);
  writeln('PartCount:   ', PartCount);
  end;   (* end for *)
end;   (* end of procedure DisplayRows *)
.
.
.
```

Several variables are initialized:

```
StartIndex      := 1;
NumberOfRows    := 3;
LessThanFive    := TRUE;
while LessThanFive = TRUE do
```

```
begin
```
*The user is prompted for a range of bin numbers or a 0. If bin numbers
are entered, they are used in a BETWEEN predicate in the BULK SELECT command.
This WHILE loop can be executed as many as five times, at which time
the array would be filled.*

```
prompt('Enter a low bin number or 0 to STOP> ');
readln(LowBinNumber);

if LowBinNumber <> 0 then
  begin
  prompt('Enter a high bin number> ');
  readln(HighBinNumber);

  EXEC SQL BULK SELECT  CountCycle, COUNT(PartNumber)
                  INTO :PartsPerCycle,
                       :StartIndex,
                       :NumberOfRows
                  FROM  PurchDB.Inventory
                 WHERE  BinNumber
               BETWEEN :LowBinNumber AND :HighBinNumber;

  StartIndex     := StartIndex + 3;
  if StartIndex = 16 then
    LessThanFive := FALSE;
  end  (* if LowBinNumber *)
else
  LessThanFive := FALSE;
end;  (* while *)
```

*The final StartIndex value can be used to display the final contents of
the host variable array.*

```
if StartIndex > 1 then DisplayRows;
```

The following example illustrates the use of SQLERRD(3) to display rows stored in the host variable array. It also checks SQLCODE in conjunction with SQLERRD(3), to determine whether or not the BULK SELECT executed without error and whether there may be additional qualified rows for which there was not room in the array. In each case, an appropriate message is displayed.

```
procedure DisplayRows;
var
  i : integer;
begin
for i := 1 to SQLCA.SQLERRD[3] do
with OrdersArray[i] do
  begin
  writeln ('OrderNumber:   ', OrderNumber);
  writeln ('VendorNumber:  ', VendorNumber);
  end;
end;     (* end of procedure DisplayRows *)
.
.
```

The variable MaximumRows is set to the number of records in the host variable array.

```
MaximumRows  := 25;
.
.
EXEC SQL BULK SELECT  OrderNumber, VendorNumber
                INTO :OrdersArray
                FROM  PurchDB.Orders;

case SQLCA.SQLCODE of
  0           : begin
                if SQLCA.SQLERRD[3] = MaximumRows  then
                  begin
                  write('There may be additional rows ');
                  writeln('that cannot be displayed.');
                DisplayRows;
                end;
  100         : writeln('No rows were found.');
  otherwise     begin
                if SQLCA.SQLERRD[3] > 0 then
                  begin
                  write('The following rows were retrieved ');
                  writeln('before an error occurred:');
                  DisplayRows;
                  end;
                SqlStatusCheck;
                end;
    end;
```

BULK FETCH

The BULK FETCH command is useful for reporting applications that operate on large query results whose maximim size is unknown at programming time. The syntax of the BULK FETCH command is:

```
BULK FETCH CursorName
      INTO ArrayName [,StartIndex [,NumberOfRows]]
```

You use this command in conjunction with the following cursor commands:

- DECLARE CURSOR: defines a cursor and associates with it a query. The cursor declaration should not contain a FOR UPDATE clause, however, because the BULK FETCH command is designed to be used for active set *retrieval* only. The order of the select list items in the embedded SELECT command must match the order of the corresponding host variables in the host variable array.

- OPEN: defines the active set.

- BULK FETCH: delivers rows into the host variable array and advances the cursor to the last row delivered. If a single execution of this command does not retrieve the entire active set, you re-execute it to retrieve subsequent rows in the active set.

- CLOSE: releases ALLBASE/SQL internal buffers used to handle cursor operations.

To retrieve all the rows in an active set larger than the host variable array, you can test for a value of 100 in SQLCODE to determine when you have fetched the last row in the active set:

```
EXEC SQL BEGIN DECLARE SECTION;
  ⋮
SupplierBuffer         : packed array[1..20]
  of packed record
    PartNumber         : packed array[1..16] of char;
    VendorName         : packed array[1..30] of char;
    DeliveryDays       : SmallInt;
    DeliveryDaysInd    : SqlInd;
  end;
EXEC SQL END DECLARE SECTION;
DoFetch                : boolean;
Response               : packed array[1..3] of char;
  ⋮
procedure DisplayRows;
var
  i : integer;
begin
for i := 1 to SQLCA.SQLERRD[3] do
with SupplierBuffer[i] do
  begin
        The values in each row returned by the BULK FETCH command are displayed here.
  end;
```

```
  if SQLCA.SQLCODE = 0 then
    begin
    prompt('Do you want to see additional rows? (YES/NO)> ');
    readln('Response');
    if Response[1] in [N','n'] then DoFetch := FALSE;
    end;
end;    (* end of DisplayRows procedure *)
.
.
EXEC SQL DECLARE SupplierInfo
           CURSOR FOR
           SELECT PartNumber,
                  VendorName,
                  DeliveryDays
             FROM PurchDB.Vendors,
                  PurchDB.SupplyPrice
            WHERE PurchDB.Vendors.VendorNumber =
                  PurchDB.SupplyPrice.Vendornumber
         ORDER BY PartNumber;

EXEC SQL OPEN SupplierInfo;

DoFetch = TRUE;
while DoFetch = TRUE do
  begin
  EXEC SQL BULK FETCH SupplierInfo
                 INTO SupplierBuffer;
  case SQLCA.SQLCODE of
    0          : DisplayRows;
    100        : begin
                 writeln('No rows were found');
                 DoFetch := FALSE;
                 end;
    otherwise    begin
                 DisplayRows;
                 SqlStatusCheck;
                 DoFetch := FALSE;
                 end;
    end;   (* end case *)

  EXEC SQL CLOSE SupplierInfo;
```

Each time the BULK FETCH command is executed, the CURRENT row is the last row put
by ALLBASE/SQL into the host variable array. When the last row in the active set has been
fetched, ALLBASE/SQL sets SQLCODE to 100 the next time the BULK FETCH command
is executed.

BULK INSERT

The BULK INSERT command is useful for multiple-row insert operations. The syntax of the
BULK INSERT command is:

```
BULK INSERT INTO TableName
               (ColumnNames)
        VALUES (ArrayName [,StartIndex [,NumberOfRows]])
```

As in the case of the simple INSERT command you can omit ColumnNames when you provide
values for all columns in the target table. ALLBASE/SQL attempts to assign a null value to
any unnamed column.

In the following example, a user is prompted for multiple rows. When the host variable array
is full and/or when the user is finished specifying values, the BULK INSERT command is
executed:

```
EXEC SQL BEGIN DECLARE SECTION;
  ⋮
NewParts                   : packed array[1..20]
  of packed record
    PartNumber             : packed array[1..16] of char;
    PartName               : packed array[1..30] of char;
    PartNameInd            : SqlInd;
    SalesPrice             : longreal;
    SalesPriceInd          : SqlInd;
  end;
StartIndex                 : SmallInt;
NumberOfRows               : SmallInt;
EXEC SQL END DECLARE SECTION;


DoneEntry                  : boolean;
Response                   : packed array[1..4] of char;
  ⋮
procedure BulkInsert;

EXEC SQL BULK INSERT INTO  PurchDB.Parts
                         (PartNumber,
                          PartName,
                          SalesPrice)
                  VALUES (:NewParts,
                          :StartIndex,
                          :NumberOfRows);
  .
  .
  .
end;     (* end of procedure BulkInsert *)
procedure PartEntry;
```

The user is prompted for three column values, and the values are assigned to the
appropriate record in the host variable array; then the array row counter
(NumberOfRows)is incremented and the user asked whether the user wants to specify
another line item.

```
NumberOfRows := NumberOfRows + 1;
prompt('Do you want to specify another line item (Y/N)?> ');
readln(Response);

if Response[1] in ['N','n'] then
  begin
  DoneEntry := TRUE;
  BulkInsert;
  end
else
  begin
  if NumberOfRows = 20 then
    begin
    BulkInsert;
    NumberOfRows := 0;
    end
  else
    BulkInsert;
  end;    (* end else *)
end;      (* end of procedure PartEntry *)
.
.

.
StartIndex      := 1;
NumberOfRows    := 0;
DoneEntry       := FALSE;
repeat PartEntry until DoneEntry;
```

Transaction Management for BULK Operations

Bulk processing, by using only one ALLBASE/SQL command to operate on multiple rows, provides a way of minimizing the time page or table locks are held. Locks are only held while moving rows between database tables and an array defined by the program, and operations can be done while holding data in that array without holding locks against the database.

Because the BULK FETCH command may need to be executed several times before an entire active set is retrieved, locks obtained to execute this command may be held longer than locks needed to execute the other BULK commands. Therefore this command is most useful for applications running when multi-user DBEnvironment access is minimal or when concurrent transactions do not need to update the table that is the target of the BULK FETCH.

Transaction management is further discussed in the *ALLBASE/SQL Reference Manual* .

Program Using BULK INSERT

The flowchart in Figure 9-1 summarizes the functionality of program pasex9. This program creates orders in the sample DBEnvironment, PartsDBE. Each order is placed with a specific vendor, to obtain one or more parts supplied by that vendor.

An order consists of a row in table PurchDB.Orders, which comprises the order header:

    OrderNumber (defined NOT NULL)
    VendorNumber
    OrderDate

An order usually also consists of one or more line items, represented by one or more rows in table PurchDB.OrderItems, such as the following:

    OrderNumber (defined NOT NULL)
    ItemNumber (defined NOT NULL)
    VendPartNumber
    PurchasePrice (defined NOT NULL)
    OrderQty
    ItemDueDate
    ReceivedQty

Program pasex9 uses a simple INSERT command to create the order header and, optionally, a BULK INSERT command to insert line items.

The runtime dialog for pasex9 appears in Figure 9-2, and the source code in Figure 9-3.

To establish a DBE session, pasex9 executes function ConnectDBE ⟨ 54 ⟩. This function evaluates to TRUE when the CONNECT command ⟨ 5 ⟩ is successfully executed.

The program then executes procedure CreateOrder until the Done flag is set to TRUE ⟨ 55 ⟩.

Procedure CreateOrder prompts for a vendor number or a 0 ⟨ 48 ⟩. When the user enters a 0, Done is set to TRUE ⟨ 53 ⟩ and the program terminates. When the user enters a vendor number, pasex9 performs the tasks in the following list.

- Validates the number entered.

- Creates an order header if the vendor number is valid.

- Optionally inserts line items if the order header has been successfully created; the part number for each line item is validated to ensure the vendor actually supplies the part.

- Displays the order created.

To validate the vendor number, procedure ValidateVendor is executed ⌊ 49 ⌋. Procedure ValidateVendor starts a transaction by invoking procedure BeginTransaction ⌊ 9 ⌋, which executes the BEGIN WORK command ⌊ 6 ⌋. Then a SELECT command ⌊ 10 ⌋ is processed to determine whether the vendor number exists in column VendorNumber of table PurchDB.Vendors:

- If the number exists in table PurchDB.Vendors, the vendor number is valid. Flag VendorOK is set to TRUE, and the transaction is terminated by invoking procedure CommitWork ⌊ 11 ⌋. CommitWork executes the COMMIT WORK command ⌊ 7 ⌋.

- If the vendor number is not found, COMMIT WORK is executed and a message is displayed to inform the user that the number entered is invalid ⌊ 12 ⌋. Several flags are set to FALSE so that when control returns to procedure CreateOrder, the user is again prompted for a vendor number.

- If the SELECT command fails, procedure SQLStatusCheck is invoked ⌊ 13 ⌋ to display any error messages ⌊ 4 ⌋. Then the COMMIT WORK command is executed, and the appropriate flags set to FALSE.

If the vendor number is valid, pasex9 invokes procedure CreateHeader to create the order header ⌊ 50 ⌋. The order header consists of a row containing the vendor number entered, plus two values computed by the program: OrderNumber and OrderDate.

Procedure CreateHeader starts a transaction ⌊ 34 ⌋, then obtains an exclusive lock on table PurchDB.Orders ⌊ 35 ⌋. Exclusive access to this table ensures that when the row is inserted, no row having the same number will have been inserted by another transaction. The unique index that exists on column OrderNumber prevents duplicate order numbers in table PurchDB.Orders. Therefore an INSERT operation fails if it attempts to insert a row having an order number with a value already in column OrderNumber.

In this case, the exclusive lock does not threaten concurrency. No operations conducted between the time the lock is obtained and the time it is released involve operator intervention:

- Procedure CreateHeader invokes procedure ComputeOrderNumber ⌊ 36 ⌋ to compute the order number and the order date.

- Procedure ComputeOrderNumber executes a SELECT command to retrieve the highest order number in PurchDB.Orders ⌊ 30 ⌋. The number retrieved is incremented by one ⌊ 31 ⌋ to assign a number to the order.

- Procedure ComputeOrderNumber then executes procedure SystemDate ⌊ 32 ⌋. This procedure uses the HP-UX system call (time) and library call (nl_time) ⌊ 2 ⌋ to retrieve and format the current date. The date retrieved is converted into YYYYMMDD format, the format in which dates are stored in the sample DBEnvironment.

- Procedure ComputeOrderNumber then executes procedure InsertRow ⌊ 33 ⌋. This procedure executes a simple INSERT command ⌊ 22 ⌋ to insert a row into PurchDB.Orders. If the INSERT command succeeds, the transaction is terminated with a COMMIT WORK command, and the HeaderOK flag is set to TRUE ⌊ 24 ⌋. If the INSERT command fails,

the transaction is terminated with COMMIT WORK, but the HeaderOK flag is set to FALSE $\boxed{23}$ so that the user is prompted for another vendor number when control returns to procedure CreateOrder.

To create line items, procedure CreateOrder executes procedure CreateOrderItems until the DoneItems flag is set to TRUE $\boxed{51}$. Procedure CreateOrderItems asks the user whether she wants to specify line items $\boxed{44}$.

If the user wants to create line items, CreateOrderItems executes procedure ItemEntry until the DoneItems flag is set to TRUE $\boxed{46}$, then executes procedure BulkInsert $\boxed{47}$:

- ItemEntry assigns values to host variable array OrderItems $\boxed{1}$; each record in the array corresponds to one line item, or row in PurchDB.OrderItems. The procedure first assigns the order number and a line number to each row $\boxed{37}$, beginning at one. ItemEntry then prompts fo a vendor part number $\boxed{38}$, which is validated by invoking procedure ValidatePart $\boxed{39}$.

  ValidatePart starts a transaction $\boxed{14}$. Then it executes a SELECT command $\boxed{15}$ to determine whether the part number entered matches any part number known to be supplied by the vendor. If the part number is valid, the COMMIT WORK command is executed $\boxed{16}$ and the PartOK flag set to TRUE. If the part number is invalid, COMMIT WORK is executed $\boxed{17}$, and the user informed that the vendor does not supply any part having the number specified; then the PartOK flag is set to FALSE so that the user is prompted for another part number when control returns to procedure ItemEntry.

  If the part number is valid, procedure ItemEntry completes the line item. It prompts for values to assign to columns PurchasePrice, OrderQty, and ItemDueDate $\boxed{40}$. The procedure then assigns a negative value to the indicator variable for column ReceivedQty $\boxed{41}$ in preparation for inserting a null value into this column.

  ItemEntry terminates when the user indicates that she does not want to specify any more line items $\boxed{42}$ or when the host variable array is full $\boxed{43}$.

- Procedure BulkInsert starts a transaction $\boxed{25}$, then executes the BULK INSERT command $\boxed{27}$. The line items in array OrderItems are inserted into table PurchDB.OrderItems, starting with the first record and continuing for as many records as there were line items specified $\boxed{26}$. If the BULK INSERT command succeeds, the COMMIT WORK command is executed $\boxed{29}$ and the ItemsOK flag set to TRUE. If the BULK INSERT command fails, procedure RollBackWork is executed $\boxed{28}$ to process the ROLLBACK WORK command $\boxed{8}$ so that any rows inserted prior to the failure are rolled back.

If the user does not want to create line items, procedure CreateOrderItems displays the order header by invoking procedure DisplayHeader $\boxed{45}$. DisplayHeader displays the row inserted earlier in PurchDB.Orders $\boxed{18}$.

If line items were inserted into PurchDB.OrderItems, procedure DisplayOrder is invoked $\boxed{52}$ to display the order created. DisplayOrder invokes procedure DisplayHeader $\boxed{20}$ to display the order header. Then it executes procedure DisplayItems $\boxed{21}$ to display each row inserted into PurchDB.OrderItems. DisplayItems displays values from array OrderItems $\boxed{19}$.

When the program user enters a 0 in response to the vendor number prompt, the program terminates by executing procedure TerminateProgram $\boxed{56}$, which executes the RELEASE command $\boxed{3}$.
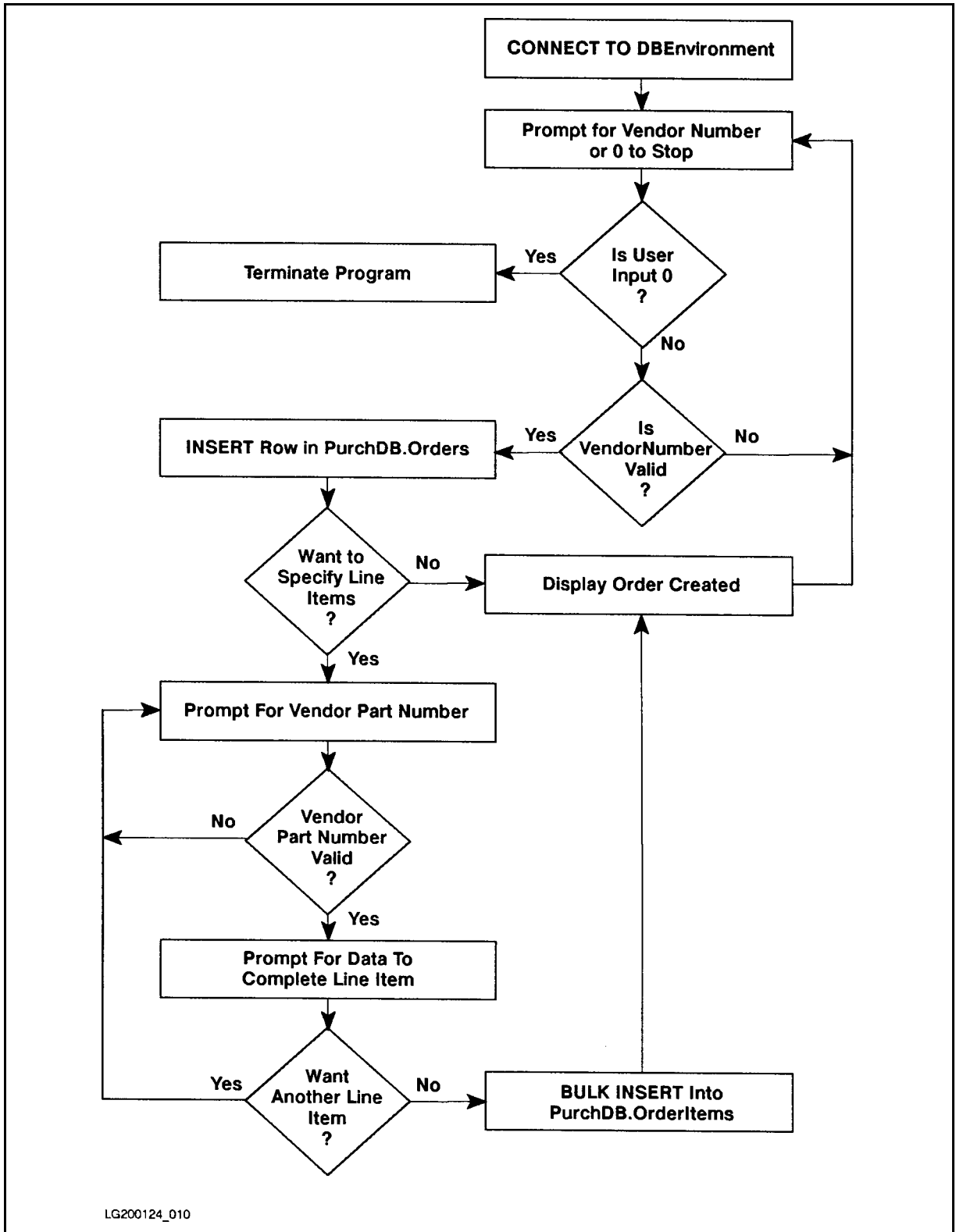
Figure 9-1. Flow Chart of Program pasex9

```
Program to Create an Order - pasex9
Event List:
  Connect to PartsDBE
  Prompt for VendorNumber
  Validate VendorNumber
  INSERT a row into PurchDB.Orders
  Prompt for line items
  Validate VendPartNumber for each line item
  BULK INSERT rows into PurchDB.OrderItems
  Repeat the above six steps until the user enters 0
  Release PartsDBE

Connect to PartsDBE

Enter VendorNumber or 0 to STOP> 9015

Begin Work
Validating VendorNumber
Commit Work

Begin Work
Calculating OrderNumber
Calculating OrderDate
INSERT INTO PurchDB.Orders
Commit Work

Do you want to specify line items (Y/N)?> y

You can specify as many as 25 line items.

Enter data for ItemNumber  1:
 VendPartNumber> 9040

Begin Work
Validating VendPartNumber
Commit Work

 PurchasePrice> 1500
 OrderQty> 5
 ItemDueDate (YYYYMMDD)> 19870630

Do you want to specify another line item (Y/N)?> y

You can specify as many as 25 line items.

Enter data for ItemNumber  2:
 VendPartNumber> 9055
```

Figure 9-2. Runtime Dialog of Program pasex9

```
Begin Work
Validating VendPartNumber
Commit Work

The vendor has no part with the number you specified.

You can specify as many as 25 line items.

Enter data for ItemNumber  2:
 VendPartNumber> 9050

Begin Work
Validating VendPartNumber
Commit Work

 PurchasePrice> 345
 OrderQty> 2
 ItemDueDate (YYYYMMDD)> 19870801

Do you want to specify another line item (Y/N)?> n

Begin Work
BULK INSERT INTO PurchDB. OrderItems
Commit Work

The following order has been created:

 OrderNumber:            30524
 VendorNumber:            9015
 OrderDate:     19870603

 ItemNumber:                   1
   VendPartNumber:  9040
   PurchasePrice:       1500.00
   OrderQty:                   5
   ItemDueDate:     19870630
   ReceivedQty:  NULL

 ItemNumber:                   2
   VendPartNumber:  9050
   PurchasePrice:        345.00
   OrderQty:                   2
   ItemDueDate:     19870801
   ReceivedQty:  NULL

Enter VendorNumber or 0 to STOP> 0
```

Figure 9-2. Runtime Dialog of Program pasex9 (page 2 of 2)

```
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* This program illustrates the use of BULK INSERT                *)
(* to insert multiple rows at a time.                             *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

Program pasex9(input, output);

const
    OK             =        0;
    NotFound       =      100;
    DeadLock       = -14024;

type
  DateType      = packed array[1..8] of char;
  DatePtr       = DateType;
  FormType      = packed array[1..10] of char;

var
          (* Begin Host Variable Declarations *)
    EXEC SQL BEGIN DECLARE SECTION;
    OrderNumber1            : integer;
    VendorNumber            : integer;
    OrderDate               : packed array[1..8] of char;

    PartSpecified           : packed array[1..16] of char;
    MaxOrderNumber          : integer;

    OrderItems              : array[1..25]                    [1]
       of record
          OrderNumber2      : integer;
          ItemNumber        : integer;
          VendPartNumber    : packed array [1..16] of char;
          PurchasePrice     : longreal;
          OrderQty          : SmallInt;
          ItemDueDate       : packed array[1..8] of char;
          ReceivedQty       : SmallInt;
          ReceivedQtyInd    : SqlInd;
       end;

    StartIndex              : SmallInt;
    NumberOfRows            : SmallInt;

    SQLMessage              : packed array[1..132] of char;
    EXEC SQL END DECLARE SECTION;
          (* End Host Variable Declarations *)
```

Figure 9-3. Program pasex9: Using BULK INSERT

```
    SQLCA : SQLCA_type;    (* SQL Communication Area *)


    Done                    : boolean;
    DoneItems               : boolean;
    VendorOK                : boolean;
    HeaderOK                : boolean;
    PartOK                  : boolean;
    ItemsOK                 : boolean;
    Abort                   : boolean;


    Response                : packed array [1..4] of char;


    counter1                : integer;
    counter2                : integer;


    i,j         : integer;

(* Intrinsic to get today's date from the system *)
procedure time (var sec:integer);external;


function nl_ctime(var clock:integer; var Format:FormType;
                  Lang:integer):DatePtr; external;


procedure SystemDate;                                        ⌐2⌐
var
    sec     : integer;
    Date    : DatePtr;
    Format  : FormType;


begin


Time(sec)
Format := '19%y%m%d';
Format[10] := chr(0);
Date := nl_ctime(sec,Format,0);
OrderDate := Date;


end; (* SystemDate procedure *)
```

Figure 9-3. Program pasex9: Using BULK INSERT (page 2 of 11)

```
procedure TerminateProgram;    (* Procedure to Release PartsDBE *)
begin
EXEC SQL RELEASE;                                                         ③
Done := TRUE;
end;  (* End TerminateProgram Procedure *)
$PAGE $


procedure SQLStatusCheck; (*Procedure to Display Error Messages*)         ④
begin


Abort := FALSE;
if SQLCA.SQLCODE < DeadLock then Abort := TRUE;


repeat
EXEC SQL SQLEXPLAIN :SQLMessage;
writeln(SQLMessage);
until SQLCA.SQLCODE = 0;


if Abort then TerminateProgram;
end;  (* End SQLStatusCheck Procedure *)
$PAGE $
function ConnectDBE: boolean;  (* Function to Connect to PartsDBE *)
begin
writeln('Connect to PartsDBE');
EXEC SQL CONNECT TO '../sampledb/PartsDBE';                               ⑤


ConnectDBE := TRUE;
if SQLCA.SQLCODE <> OK then
  begin
  ConnectDBE := FALSE;
  SQLStatusCheck;
  end;  (* End if *)
end;  (* End of ConnectDBE Function *)


procedure BeginTransaction;                 (* Procedure to Begin Work *)
begin


EXEC SQL BEGIN WORK;                                                      ⑥
if SQLCA.SQLCODE <> OK then
  begin
  SQLStatusCheck;
  TerminateProgram;
  end;
end;  (* End BeginTransaction Procedure *)
```

Figure 9-3. Program pasex9: Using BULK INSERT (page 3 of 11)

```
procedure CommitWork;  (* Procedure to Commit Work *)
begin
writeln('Commit Work');
EXEC SQL COMMIT WORK;                                              [7]
if SQLCA.SQLCODE <> OK then
  begin
  SQLStatusCheck;
  TerminateProgram;
  end;
end;  (* End CommitWork Procedure *)


procedure RollBackWork;  (* Procedure to RollBack Work *)
begin
writeln('Rollback Work');
EXEC SQL ROLLBACK WORK;                                           [8]
if SQLCA.SQLCODE <> OK then
  begin
  SQLStatusCheck;
  TerminateProgram;
  end;
end;  (* End RollBackWork Procedure *)


procedure ValidateVendor;(* procedure ensures VendorNumber is valid*)
begin
writeln;
writeln('Begin Work');
writeln('Validating VendorNumber');
BeginTransaction;                                                 [9]
EXEC SQL SELECT  VendorNumber                                     [10]
          INTO :VendorNumber
          FROM  PurchDB.Vendors
         WHERE  VendorNumber = :VendorNumber;
case SQLCA.SQLCODE of
OK        : begin
                CommitWork;                                      [11]
                VendorOK := TRUE;
            end;
NotFound  : begin
                CommitWork;                                      [12]
                writeln;
                writeln('No vendor has the VendorNumber you specified.')
                VendorOK := FALSE;
                HeaderOK := FALSE;
                ItemsOK  := FALSE;
            end;
```

Figure 9-3. Program pasex9: Using BULK INSERT (page 4 of 11)

```
Otherwise        begin
                    SQLStatusCheck;                                    (13)
                    CommitWork;
                    VendorOK := FALSE;
                    HeaderOK := FALSE;
                    ItemsOK  := FALSE;
                end;
end;      (* case *)
end;      (* End of Procedure  ValidateVendor *)

procedure ValidatePart;(*procedure ensures VendorPart number is valid*)

var
    i : integer;
begin

writeln;
writeln('Begin Work');
writeln('Validating VendPartNumber');

BeginTransaction;                                                     (14)

i := counter1;
PartSpecified := OrderItems[i].VendPartNumber;
EXEC SQL SELECT  VendPartNumber                                       (15)
           INTO :PartSpecified
           FROM  PurchDB.SupplyPrice
          WHERE  VendorNumber   = :VendorNumber
            AND  VendPartNumber = :PartSpecified;
case SQLCA.SQLCODE of
OK              : begin
                    CommitWork;                                       (16)
                    PartOK := TRUE;
                end;
NotFound        : begin
                    CommitWork;                                       (17)
                    writeln;
                    write('The vendor has no part with the number ');
                    writeln('you specified.');
                    PartOK  := FALSE;
                end;
Otherwise        begin
                    SQLStatusCheck;
                    CommitWork;
                    PartOK  := FALSE;
                end;
end;      (* case *)
end;      (* End of Procedure  ValidatePart *)
```

Figure 9-3. Program pasex9: Using BULK INSERT (page 5 of 11)

```
procedure DisplayHeader; (*Procedure displays row from PurchDB.Orders*)
begin

writeln;
writeln('The following order has been created:');
writeln;
writeln(' OrderNumber:    ' ,OrderNumber1);                        [18]
writeln(' VendorNumber:   ' ,VendorNumber);
writeln(' OrderDate:      ' ,OrderDate);

end;   (* End of Procedure DisplayHeader *)

procedure DisplayItems;
    (*Procedure displays rows from PurchDB.OrderItems*)
var
    j : integer;

begin

j := counter2;
writeln;
writeln(' ItemNumber:        ' ,OrderItems[j].ItemNumber);         [19]
writeln('   VendPartNumber: ' ,OrderItems[j].VendPartNumber);
writeln('   PurchasePrice:  ' ,OrderItems[j].PurchasePrice:10:2);
writeln('   OrderQty:       ' ,OrderItems[j].OrderQty);
writeln('   ItemDueDate:    ' ,OrderItems[j].ItemDueDate);
writeln('   ReceivedQty:  NULL');
counter2 := j + 1;

end;   (* End of Procedure DisplayItems *)

procedure DisplayOrder; (* Procedure to Display Order Created *)

var
    i : integer;
    j : integer;

begin

DisplayHeader;                                                    [20]
writeln;
i := counter1;
counter2 := 1;

for j := 1 to i do DisplayItems;                                  [21]

end;  (* End of Procedure DisplayOrder *)
```

Figure 9-3. Program pasex9: Using BULK INSERT (page 6 of 11)

```
procedure InsertRow; (* procedure to insert row in PurchDB.Orders *)
begin
writeln('INSERT INTO PurchDB.Orders');
EXEC SQL INSERT INTO    PurchDB.Orders                              ⌧22⌧
                    ( OrderNumber,
                      VendorNumber,
                      OrderDate )
          VALUES (:OrderNumber1,
                   :VendorNumber,
                   :OrderDate );
if SQLCA.SQLCODE <> 0 then
  begin
  SQLStatusCheck;                                                   ⌧23⌧
  CommitWork;
  HeaderOK := FALSE;
  end
else
  begin
  CommitWork;                                                       ⌧24⌧
  HeaderOK := TRUE;
  end;
end;                      (* End of Procedure InsertRow *)


procedure BulkInsert;
   (* procedure to BULK INSERT into PurchDB.OrderItems *)
begin
writeln;
writeln('Begin Work');
BeginTransaction;                                                  ⌧25⌧
NumberOfRows := counter1;                                          ⌧26⌧
StartIndex   := 1;


writeln('BULK INSERT INTO PurchDB. OrderItems');
EXEC SQL BULK INSERT INTO    PurchDB.OrderItems                    ⌧27⌧
                         ( OrderNumber,
                           ItemNumber,
                           VendPartNumber,
                           PurchasePrice,
                           OrderQty,
                           ItemDueDate,
                           ReceivedQty )
              VALUES (:OrderItems,
                       :StartIndex,
                       :NumberOfRows );
```

Figure 9-3. Program pasex9: Using BULK INSERT (page 7 of 11)

```
  if SQLCA.SQLCODE <> 0 then
    begin
    SQLStatusCheck;
    RollBackWork;                                                    [28]
    ItemsOK := FALSE;
    end
  else
    begin
    CommitWork;                                                      [29]
    ItemsOK := TRUE;
    end;
end;        (* End of Procedure BulkInsert *)
procedure ComputeOrderNumber;  (* procedure to assign number to order *)
begin
EXEC SQL SELECT  MAX(OrderNumber)                                    [30]
            INTO :MaxOrderNumber
            FROM  PurchDB.Orders;
if SQLCA.SQLCODE <> 0 then
    begin
    SQLStatusCheck;
    CommitWork;
    HeaderOK := FALSE;
    end
  else
    begin
    writeln('Calculating OrderNumber');
    OrderNumber1 := MaxOrderNumber + 1;                             [31]
    writeln('Calculating OrderDate');
    SystemDate;                                                      [32]

    InsertRow;                                                       [33]
    end;
end;                         (* End of ComputeOrderNumber Procedure *)
procedure CreateHeader;   (* procedure to create order header *)
begin
writeln;
writeln('Begin Work');
BeginTransaction;                                                    [34]

EXEC SQL LOCK TABLE PurchDB.Orders IN EXCLUSIVE MODE;                [35]
if SQLCA.SQLCODE <> OK then
    begin
    SQLStatusCheck;
    CommitWork;
    HeaderOK := FALSE;
    end
```

Figure 9-3. Program pasex9: Using BULK INSERT (page 8 of 11)

```
else
  ComputeOrderNumber;                                               36
end;                       (* End of CreateHeader Procedure *)
procedure ItemEntry; (* procedure puts line items in OrderItems array
var
    i : integer;
begin
i := counter1;
OrderItems[i].OrderNumber2 := OrderNumber1;                         37
OrderItems[i].ItemNumber   := i;
writeln;
writeln('You can specify as many as 25 line items.');
writeln;
writeln('Enter data for ItemNumber ',OrderItems[i].ItemNumber:2 ,':');
writeln;
prompt(' VendPartNumber> ');                                        38
readln(OrderItems[i].VendPartNumber);

ValidatePart;                                                       39
if PartOK then
  begin
  writeln;

  prompt(' PurchasePrice> ');                                       40
  readln(OrderItems[i].PurchasePrice);
  prompt(' OrderQty> ');
  readln(OrderItems[i].OrderQty);
  prompt(' ItemDueDate (YYYYMMDD)> ');
  readln(OrderItems[i].ItemDueDate);

  OrderItems[i].ReceivedQtyInd := -1;                               41
  if i < 25 then
    begin
    writeln;
    prompt('Do you want to specify another line item (Y/N)?> '); 42
    readln(Response);
    if Response[1] in ['N','n'] then
      DoneItems := TRUE
    else
      counter1 := i + 1;
    end    (* end if i < 25 *)
  else
    DoneItems := TRUE;  (* host variable array is full *)          43
  end;      (* end if PartOK *)
end;        (* End of Procedure ItemEntry *)
```

Figure 9-3. Program pasex9: Using BULK INSERT (page 9 of 11)

```
procedure CreateOrderItems;  (* procedure to create line items *)
begin
writeln;
prompt('Do you want to specify line items (Y/N)?> ');              44
readln(Response);
if Response[1] in ['N','n'] then
  begin
  DoneItems := TRUE;
  ItemsOK := FALSE;
  DisplayHeader;                                                   45
  end
else
  begin
  counter1 := 1;
  repeat                                                          46
    ItemEntry
  until DoneItems;
  BulkInsert;                                                     47
  end;
end;           (* End of procedure CreateOrderItems *)


procedure CreateOrder;   (* Procedure to create an order *)
begin
writeln;
prompt('Enter VendorNumber or 0 to STOP> ');                     48
readln(VendorNumber);
if VendorNumber <> 0 then
  begin
  ValidateVendor;                                                49

  if VendorOK then CreateHeader;                                 50
  if HeaderOK then
    begin
    DoneItems := FALSE;
    while DoneItems = FALSE do                                   51
      begin
      CreateOrderItems;
      end;  (* while *)
    end;    (* if HeaderOK *)
  if ItemsOK then DisplayOrder;                                  52
  end        (* end if VendorNumber *)
else
  Done := TRUE;                                                  53
end;          (* end of CreateOrder Procedure *)
```

Figure 9-3. Program pasex9: Using BULK INSERT (page 10 of 11)

```
$PAGE $

begin  (* Beginning of Program *)

writeln('Program to Create an Order - pasex9');
writeln('Event List:');
writeln('  Connect to PartsDBE');
writeln('  Prompt for VendorNumber');
writeln('  Validate VendorNumber');
writeln('  INSERT a row into PurchDB.Orders');
writeln('  Prompt for line items');
writeln('  Validate VendPartNumber for each line item');
writeln('  BULK INSERT rows into PurchDB.OrderItems');
writeln('  Repeat the above six steps until the user enters 0');
writeln('  Release PartsDBE');
writeln;


if ConnectDBE then                                            ⌊54⌋

  begin
  Done := FALSE;
  repeat
    CreateOrder                                               ⌊55⌋
  until Done;
  end;

TerminateProgram;                                             ⌊56⌋

end.  (* End of Program *)
```

Figure 9-3. Program pasex9: Using BULK INSERT (page 11 of 11)

10

Using Dynamic Operations

Dynamic operations are used to execute SQL commands that are not preprocessed until run time. Such commands, known as **dynamic** SQL commands, are submitted to ALLBASE/SQL through several special SQL statements: PREPARE, DESCRIBE, EXECUTE, and EXECUTE IMMEDIATE.

This chapter contrasts dynamic with non-dynamic operations and introduces the techniques used to handle dynamic operations from a program. It then focuses on dynamic non-queries and queries. The following topics are considered:

- Review of Preprocessing Events.
- Differences between Dynamic and Non-Dynamic Preprocessing.
- Preprocessing of Dynamic Non-Queries.
- Preprocessing of Dynamic Queries.
- Preprocessing of Dynamic Commands That May or May Not be Queries.
- Programs Using Dynamic Command Operations.

Review of Preprocessing Events

All embedded SQL statements must be preprocessed before they can be executed. Preprocessing may be done by running the Pascal preprocessor during application development, or it may be done for dynamic commands when the program is run. Preprocessing does the following:

- Checks syntax: The syntax of SQL commands and host variable declarations must be correct.

- Verifies the existence of objects: Any object named in an SQL command must exist.

- Optimizes data access: If the statement accesses data, the fastest way to access the data must be determined.

- Checks authorizations: Both the program owner and the executor must have the required authorities.

- Creates sections: ALLBASE/SQL creates sections for SQL commands when this is appropriate. At run time, the section is executed.

These preprocessing events take place for all *non-dynamic* SQL commands when you run the ALLBASE/SQL preprocessor. Non-dynamic commands are fully defined in the source code and are preprocessed *before* run time. So far, most of the examples in this manual have shown non-dynamic preprocessing.

ALLBASE/SQL completes the preprocessing of dynamic commands at run time, in an event known as **dynamic** preprocessing. Any SQL command except the following, which do not require sections for execution, can be preprocessed at run time:

```
BEGIN DECLARE SECTION          FETCH
CLOSE CURSOR                   INCLUDE
DECLARE CURSOR                 OPEN CURSOR
DELETE WHERE CURRENT           PREPARE
DESCRIBE                       SQLEXPLAIN
END DECLARE SECTION            UPDATE WHERE CURRENT
EXECUTE                        WHENEVER
EXECUTE IMMEDIATE
```

Dynamic commands that are not queries can be preprocessed at run time using the PREPARE and EXECUTE statements or the EXECUTE IMMEDIATE statement. Dynamic queries are preprocessed using the PREPARE and DESCRIBE commands in conjunction with the SQLDA or SQL Description Area (**SQLDA**). These statements and data structures, used with a cursor, are described further in a later section.

Differences between Dynamic and Non-Dynamic Preprocessing

The authorization checking and section creation activities for non-dynamic and dynamic ALLBASE/SQL commands differ in the following ways:

■ Authorization checking. A non-dynamic command is executed if the owner of the program module has the proper authority at run time. A dynamic command is executed if the program executor has the proper authority at run time.

■ Section creation. Any section created for a non-dynamic command becomes part of a module permanently stored in a DBEnvironment by the Pascal preprocessor. The module remains in that system catalog until you execute the DROP MODULE command or invoke the preprocessor with the -d option. Any section created for a dynamic command is temporary. The section is created at run time, temporarily stored, then deleted at the end of the transaction in which it was created.

Permanently Stored vs. Temporary Sections

In some instances, you could code the same SQL statement as either dynamic or non-dynamic, depending on whether you wanted to store permanent sections. A program that has permanently stored sections associated with it can be executed only against DBEnvironments containing those sections. Figure 10-1 illustrates how you create and use such programs. Note that the sections can be permanently stored either by the preprocessor or by using the ISQL INSTALL command.
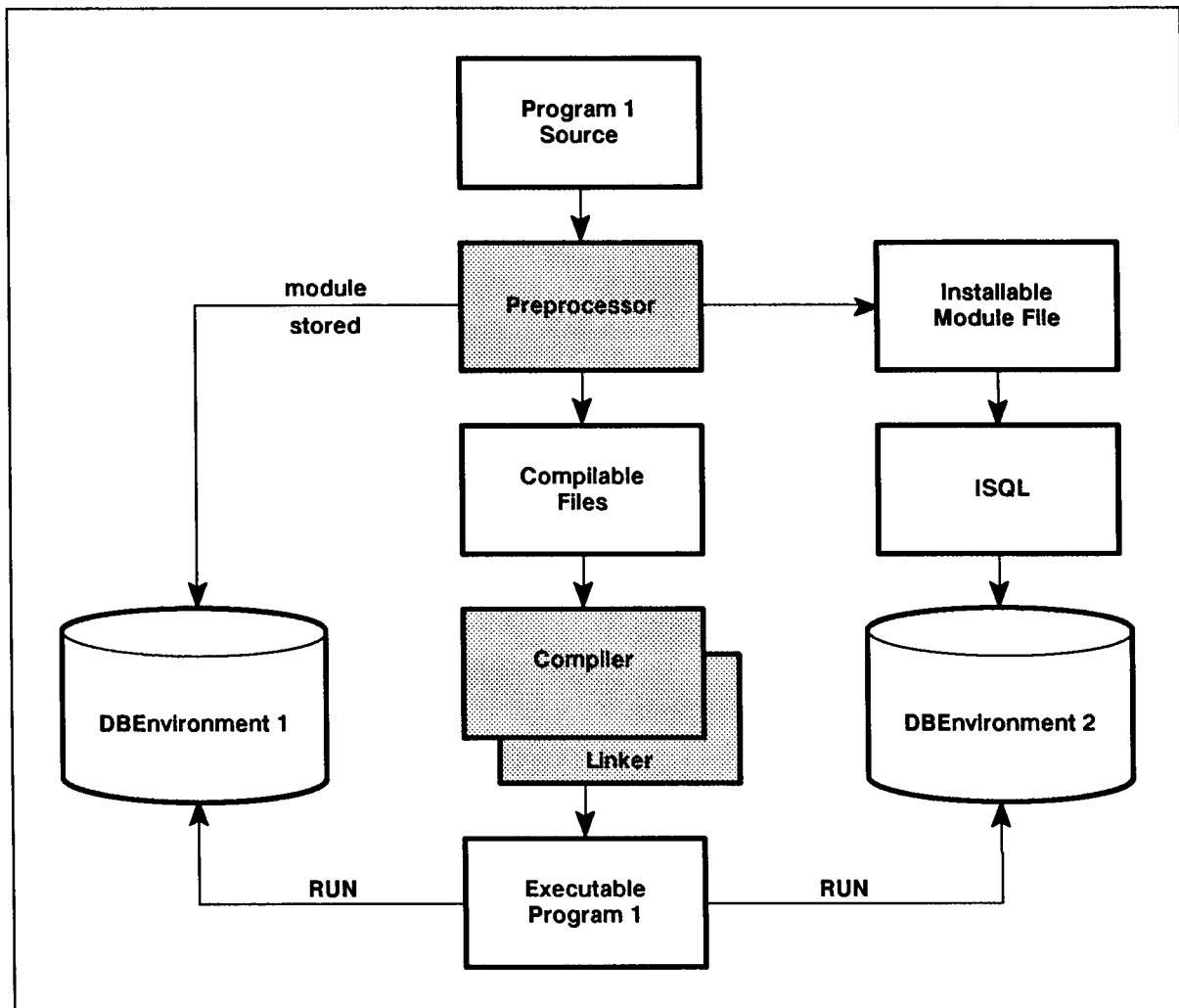
LG200125_011a

Figure 10-1. Creation and Use of a Program that has a Stored Module

Programs that contain only SQL commands that do not have permanently stored sections can be executed against *any* DBEnvironment without the prerequisite of storing a module in the DBEnvironment. Figure 10-2 illustrates how you create and use programs in this category. Note that the program must still be preprocessed in order to create compilable files and generate ALLBASE/SQL external procedure calls.

LG200125_012a
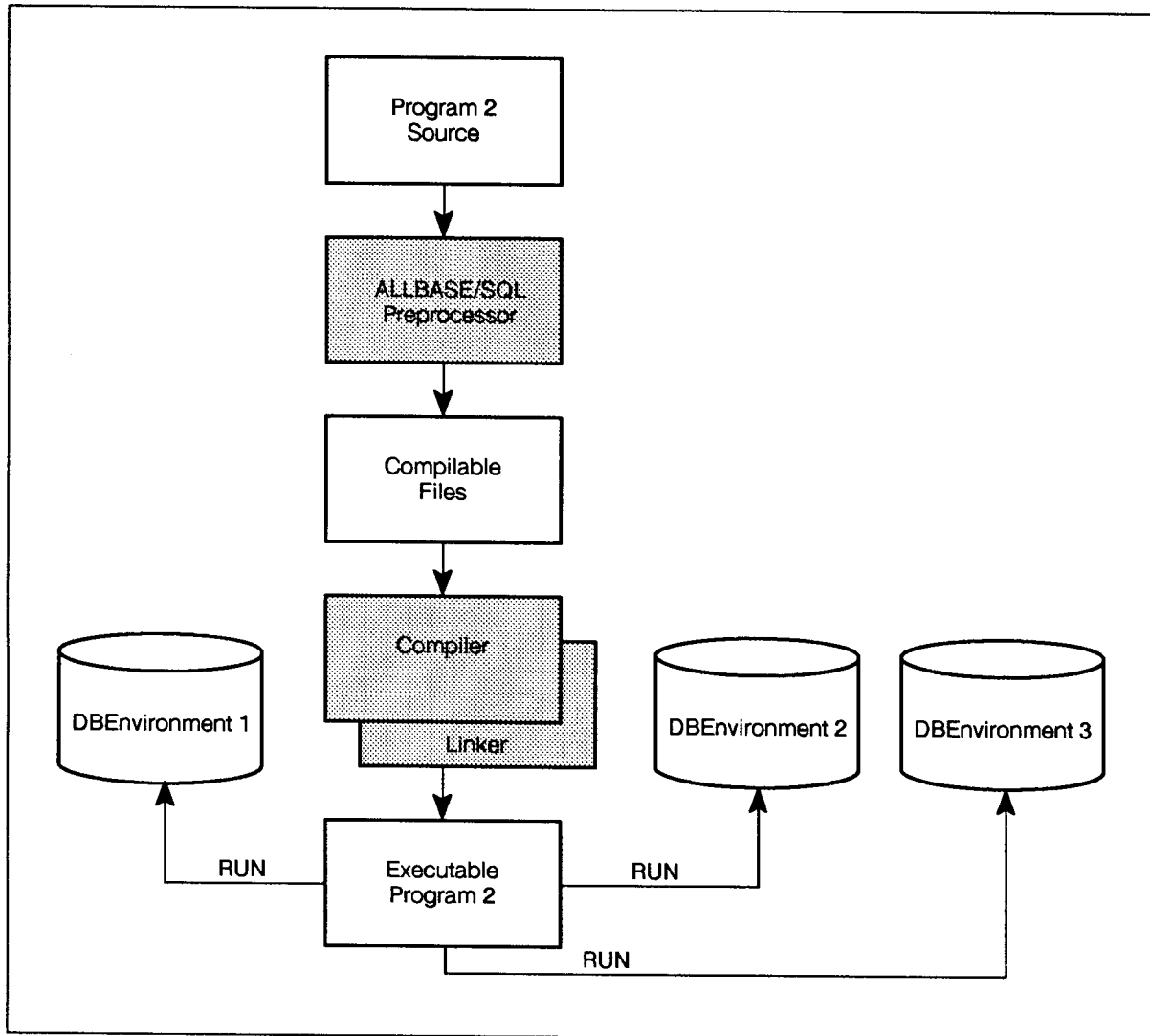
Figure 10-2. Creation and Use of a Program that has No Stored Module

Examples of Non-Dynamic and Dynamic SQL Statements

The following example shows an embedded SQL statement that is coded so as to generate a stored section before run time:

```
EXEC SQL UPDATE STATISTICS FOR TABLE PurchDB.Parts;
```

When you run the preprocessor on a source file containing this statement, a permanent section will be stored in the appropriate DBEnvironment.

The following example shows an SQL statement that is coded so as to generate a temporary section at run time:

```
DynamicCommand := 'UPDATE STATISTICS FOR TABLE PurchDB.Parts;';
EXEC SQL PREPARE MyCommand FROM :DynamicCommand;
EXEC SQL EXECUTE MyCommand;
```

In this case, the SQL statement is stored in a host variable which is passed to ALLBASE/SQL in the PREPARE statement at run time. A temporary section is then created and executed, and the section is not stored in the DBEnvironment.

Why Use Dynamic Preprocessing?

In some cases, it may *not* be desirable to preprocess an SQL command before run time:

- You may need to code an application that permits ad hoc queries requiring that SQL commands be entered by the user at run time. (ISQL is an example of an ad hoc query facility in which the command the user will submit is completely unknown at programming time.)

- You may need more specialized applications requiring SQL commands that are defined partly at programming time and partly by the user at run time. An application may, for example, perform UPDATE STATISTICS operations on tables the user specifies at run time.

- You may wish to run an application on different DBEnvironments at different times without the need to permanently store sections in those DBEnvironments.

- You may wish to code only one dynamic command (a CONNECT, for instance) and then preprocess or install the same application in several different DBEnvironments.

Passing Dynamic Commands to ALLBASE/SQL

A dynamic command is passed to ALLBASE/SQL either as a string literal or as a host variable containing a string. It must be terminated with a semicolon. The maximum length for such a string is 2048 bytes.

To pass a dynamic command that can be completely defined at programming time, you can use a delimited string:

```
EXEC SQL
PREPARE MyCommand FROM 'UPDATE STATISTICS FOR TABLE PurchDB.Parts;';
```

or

```
EXEC SQL
EXECUTE IMMEDIATE 'UPDATE STATISTICS FOR TABLE PurchDB.Parts;';
```

To pass a dynamic command that cannot be completely defined at programming time, you use a host variable declared as an array of char:

```
DynamicHostVar    : packed array[2048] of char;
 .
 .
 .
EXECUTE IMMEDIATE :DynamicHostVar;
```

Understanding the Types of Dynamic Operations

Dynamic operations in ALLBASE/SQL are of two major types:

- Dynamic Non-Queries: dynamic operations that do not retrieve rows from the database. Note that dynamic non-queries either do or do not require the use of sections at execution time. For example, a CONNECT does not require a section, but a DELETE does.

- Dynamic Queries: dynamic operations that do retrieve rows. Note that dynamic queries may have a query result whose format is known to you at programming time, or they may have a query result whose format is unknown. Dynamic queries always use sections at execution time.

It is sometimes necessary to define dynamic data structures that can accomodate *either* non-queries or queries at run time. An example is shown later in this chapter in program pas10a (Figures 10-7, 10-8, 10-9).

The following paragraphs examine each type of dynamic operation and present information on how to determine whether or not a dynamic command is a query.

Preprocessing of Dynamic Non-Queries

There are two methods for dynamic preprocessing of a non-query:

- Using EXECUTE IMMEDIATE.
- Using PREPARE and EXECUTE.

The first method can be used with any non-query; the second is only for those non-query commands that use sections at execution time.

Using EXECUTE IMMEDIATE

If you know in advance that a dynamic command will not be a query, you can dynamically preprocess and execute the command in one step, using the EXECUTE IMMEDIATE command. Figure 10-3 illustrates a procedure hosting a dynamic UPDATE STATISTICS command that can be handled in this fashion.

Procedure UpdateStatistics (1) prompts the user for a table name (2). The table name entered is assigned to the host variable CmdLine (3) to complete the UPDATE STATISTICS command. After the command is prepared and executed (4), the transaction is terminated with a COMMIT WORK command (5) or a ROLLBACK WORK command (6), depending on the value in SQLCA.SQLCODE. Terminating the transaction before accepting another

table name and re-executing the UPDATE STATISTICS command releases any locks obtained and improves concurrency.

If you do not know in advance whether a dynamic command will be a query or a non-query, you must use the PREPARE command to dynamically preprocess the command, the DESCRIBE command to distinguish between queries and non-queries, and the EXECUTE or EXECUTE IMMEDIATE command to execute the dynamic non-query. The program examined later in this chapter under "Program Using Dynamic Commands of Unknown Format" illustrates how to handle this situation.

```
   .
   .
   .
  var
    EXEC SQL BEGIN DECLARE SECTION;
    CmdLine         : String[100];
    EXEC SQL END DECLARE SECTION;

    TableName       : String[50];
   .
   .
   .
  procedure UpdateStatistics;                                      1
  begin

  repeat

  prompt('Enter name of table or / to terminate > ');             2
  readln(TableName);
  if TableName[1] <> '/' then
    begin
      CmdLine := 'UPDATE STATISTICS FOR TABLE ' + TableName + ';'; 3

      EXEC SQL EXECUTE IMMEDIATE :CmdLine;                         4

      if SQLCA.SQLCODE = 0 then
        EXEC SQL COMMIT WORK;                                      5
      else
        EXEC SQL ROLLBACK WORK;                                    6

    end;                (* end of if TableName *)
  until TableName[1] = '/';

  end;                  (* end of UpdateStatistics procedure *)
   .
   .
   .
```

Figure 10-3. Procedure Hosting Dynamic Non-Query Commands

Using **PREPARE** and **EXECUTE**

Use PREPARE command syntax to create and store a temporary section for the dynamic command:

    PREPARE *CommandName* FROM *CommandSource*

Because the PREPARE command operates only on sections, it can be used to dynamically preprocess only SQL commands executed by using sections. The DBE session management and transaction management commands can only be dynamically preprocessed by using EXECUTE IMMEDIATE.

With PREPARE, ALLBASE/SQL creates a temporary section for the command that you can execute *one or more times in the same transaction* by using the EXECUTE command:

    EXEC SQL PREPARE MyNonQuery FROM :DynamicCommand;

    for i := 1 to MaxIterations do
    EXEC SQL EXECUTE MyNonQuery;

As soon as you process a COMMIT WORK or ROLLBACK WORK command, the temporary section is deleted.

Preprocessing of Dynamic Queries

Processing of dynamic queries requires setting up a buffer to receive the query result and extracting the items you want from the buffer. For these operations, you use three special data structures:

- SQL Description Area (SQLDA). The SQLDA is a record used to pass information on the location and contents of the other two dynamic data structures, the format array and the data buffer. You set some fields in the SQLDA and pass them to ALLBASE/SQL; and ALLBASE/SQL passes values back to you in other fields.

- SQL Format Array. The format array is an array of records with one record for each select list item (column). The attributes of a column in the query result are described in a format array record. When you do not know the format of a query result at programming time, you use format array information to identify where in the data buffer to find each column value and how to interpret it.

- Data Buffer. The data buffer is an array for holding rows in a query result. ALLBASE/SQL puts rows into the data buffer each time you execute the FETCH command.

Figure 10-4 summarizes the relationships among the special data structures and when data is assigned to them. Note that status checking information for each SQL command can be found in the sqlca data structure. See the chapter "Runtime Status Checking and the SQLCA" for more details.
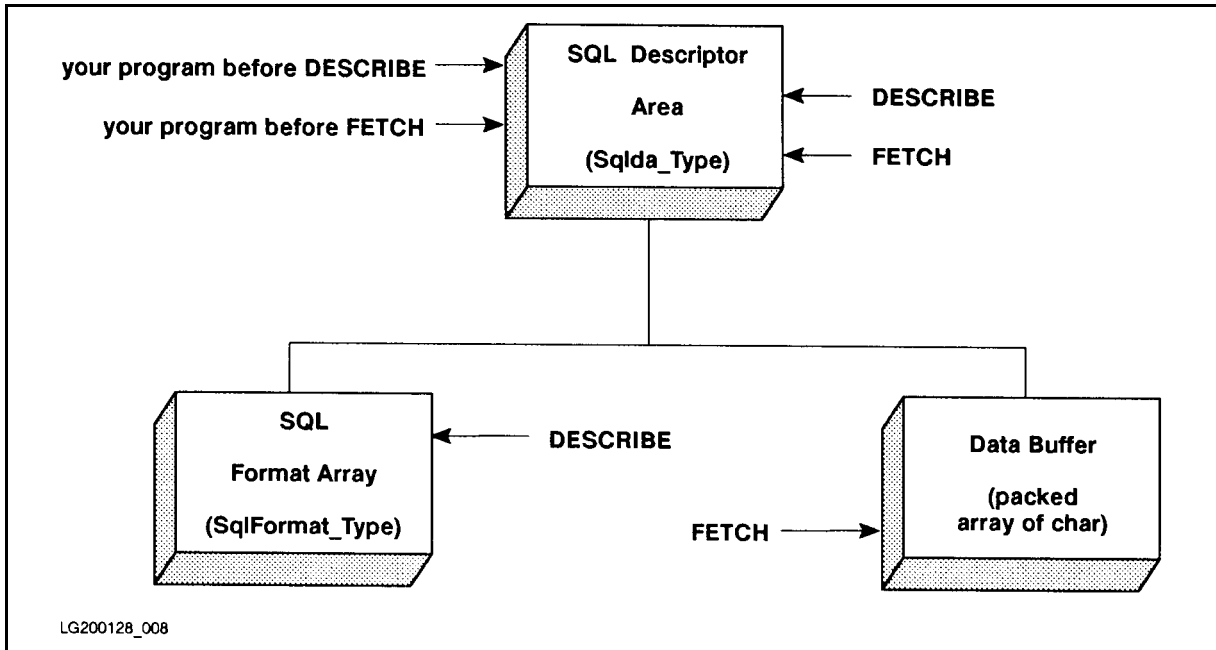
Figure 10-4. Dynamic Query Data Structures and Data Assignment

Though some specific details differ depending on the query type, in general you handle all types of dynamic queries as follows:

■ A host variable (a string) is defined to hold the SELECT statement to be used by the PREPARE command.

■ The PREPARE command dynamically preprocesses the query. ALLBASE/SQL defines a temporary section, which includes a run tree for the SELECT command specified in the PREPARE command:

```
EXEC SQL PREPARE MyQuery FROM :DynamicCommand;
```

■ The DESCRIBE command makes available to your program information about each column in a query result:

```
EXEC SQL DESCRIBE MyQuery INTO SQLDA
```

■ The DECLARE CURSOR command maps the temporary section to a cursor so that the other cursor manipulation commands can be used:

```
EXEC SQL DECLARE DynamicCursor CURSOR FOR MyQuery;
```

■ The OPEN command allocates ALLBASE/SQL buffer space for holding qualifying rows and defines the active set:

```
EXEC SQL OPEN DynamicCursor;
```

■ The FETCH command evaluates any predicates in the query and transfers rows from the ALLBASE/SQL buffer into host variables:

```
EXEC SQL FETCH DynamicCursor USING DESCRIPTOR SQLDA;
```

The USING DESCRIPTOR clause indicates to ALLBASE/SQL that rows should be formatted in accord with a format array identified in the SQLDA and returned to a data buffer identified in the SQLDA. The SQLDA, the format array, and the data buffer are discussed later in this chapter under "Using the Dynamic Query Data Structures."

Although you can fetch multiple rows with each execution of the FETCH command, you do *not* specify the BULK option when fetching rows that qualify for dynamic queries. Instead, you set a field in the SQLDA as shown later in this chapter to communicate to ALLBASE/SQL how many rows to fetch. You can repeatedly execute the FETCH command until ALLBASE/SQL sets sqlca.sqlcode to 100.

■ The CLOSE command closes the cursor and frees previously allocated buffer space:

```
EXEC SQL CLOSE DynamicCursor;
```

The COMMIT WORK and ROLLBACK WORK commands also close any open cursors, unless you are using the KEEP CURSOR option of the OPEN command (see the chapter "Processing with Cursors"). In addition, these commands release locks obtained to execute the dynamic query. Therefore, to improve concurrency when repeatedly preparing dynamic queries, issue one of these commands before executing the PREPARE command for the second and each subsequent time.

Dynamically Updating and Deleting Data

You have the option of dynamically updating or deleting a row in conjunction with a dynamic FETCH statement. Any dynamic UPDATE WHERE CURRENT or DELETE WHERE CURRENT statement must be hard coded in your program just as you would code it for a non-dynamic FETCH statement. The statements *cannot* be defined at run time and prepared.

Whether your SELECT statement is completely user specified at run time, supplied by your program based on related user input, or completely defined by your program, here are some things to keep in mind:

■ If you are using a dynamic cursor to update, be sure your SELECT statement contains a FOR UPDATE OF clause.

■ An UPDATE WHERE CURRENT command must map to an appropriate SELECT statement. Be sure all of the columns you might possibly want to update are specified in the FOR UPDATE OF clause.

For example, if the host variable or string from which you prepare contains the following statement, you can use the UPDATE WHERE CURRENT command to change the content of all the columns in qualifying rows of PurchDB.Parts.

```
      SELECT PartNumber FROM PurchDB.Parts
        WHERE PartNumber BETWEEN 9000 AND 9999
  FOR UPDATE OF PartNumber, PartName, SalesPrice
```

However, if your prepared command is based on a host variable or string containing the following statement, you will *only* be able to use UPDATE WHERE CURRENT to change column SalesPrice in any qualifying rows of PurchDB.Parts.

```
      SELECT PartNumber FROM PurchDB.Parts
        WHERE PartNumber BETWEEN 9000 AND 9999
  FOR UPDATE OF SalesPrice
```

- Your error checking strategy might include routines to parse user input for an acceptable SELECT statement and/or routines to test specific sqlca field values and invoke SQLEXPLAIN. This error checking strategy may need to be modified, if the syntax of the SELECT statement has changed for a particular ALLBASE/SQL release.

Setting Up the SQLDA

You use the INCLUDE command to declare the SQLDA in the declaration section of your program:

```
EXEC SQL INCLUDE SQLDA;
```

When the Pascal preprocessor parses this command, it inserts a type declaration for this data structure into the modified source code file:

```
$Skip_Text ON$
  EXEC SQL INCLUDE SQLDA;
$Skip_Text OFF$
sqlda: Sqlda_Type;
```

Alternatively, you can include the above type declaration in your source file and omit the INCLUDE command.

The Sqlda_Type record is defined as follows in the full preprocessor generated include file named SQLTYPE:

```
Sqlda_Type = Record
  SqldaId  : Packed Array[1..8] of Char;  reserved for ALLBASE/SQL
  Sqldabc  : Integer;                      reserved for ALLBASE/SQL
  Sqln     : Integer;                      number of format array records
  Sqld     : Integer;                      number of columns
  SqlFmtArr: Integer;                      format array address
  SqlNRow  : Integer;                      number of rows to FETCH
  SqlRRow  : Integer;                      number of rows fetched
  SqlRowLen: Integer;                      bytes in each row
  SqlBufLen: Integer;                      bytes in data buffer
  SqlRowBuf: Integer;                      data buffer address
  end;
```

Values are assigned to SQLDA fields by you or by ALLBASE/SQL, as summarized in Table 10-1.

Table 10-1. SQLDA Fields

FIELD NAME	FIELD DESCRIPTION	Pascal DATA TYPE	YOU SET BEFORE DESCRIBE	YOU SET BEFORE FETCH	ALLBASE/ SQL SETS AT DESCRIBE	ALLBASE/ SQL SETS AT FETCH
sqldaid	reserved	Packed Array [1..8] of char				
sqldabc	reserved	Integer				
sqln	number of format array records (one record (column) per select list item)	Integer	X			
sqld	number of columns in query result (0 if non-query)	Integer			X	
sqlfmtarr	address of format array	Integer	X			
sqlnrow	number of rows to FETCH into the data buffer	Integer		X		
sqlrrow	number of rows put into the data buffer	Integer				X
sqlrowlen	number of bytes in each row	Integer			X	
sqlbuflen	number of bytes in the data buffer	Integer		X		
sqlrowbuf	address of data buffer	SmallInt		X		

Setting Up the Format Array

You declare the format array as an array of records having the type SqlFormat_Type:

```
var
   SqlFmts    : Array[1..NbrFmtRecords] of SqlFormat_Type;
```

Set the number of records in the format array (NbrFmtRecords in this example) to the largest number of select list items you expect. If you do not know this value at programming time, you can allow for as many as 1024 records, since 1024 is the maximum number of columns any query result can contain, as follows:

```
const
   NbrFmtRecords = 1024;
```

On the other hand, if you know at programming time the maximum number of columns to expect, you may be able to declare a smaller format array:

```
const
   NbrFmtRecords = 6;
```

The definition for the type SqlFormat_Type appears in the full preprocessor generated type include file:

```
SqlFormat_Type = Packed Record
   SqLnty, SqlType, SqlPrec, SqlScale: SmallInt;
   SqlTotalLen, SqlValLen, SqlIndLen: integer;
   SqlVof, SqlNof: integer;
   SqlName: Packed Array [1..20] of Char;
   end;
```

Each record in the format array describes one of the columns in the query result. The first record describes the first column, the second record describes the second column, and so forth. Table 10-2 explains the meaning of each field in a format array record. Under "MEANING OF FIELD" for the sqltype field in the table, DATE, TIME, DATETIME, and INTERVAL each have different code numbers, but they all are formatted with an sqltype of code 2, CHAR, externally.

Table 10-2. Fields in a Format Array Record

FIELD NAME	MEANING OF FIELD	Pascal DATA TYPE
sqlnty	reserved; always set to 111	SmallInt

sqltype	data type of column:
	
	    `0 = SMALLINT or INTEGER`
	    `1 = BINARY*`
	    `2 = CHAR*`
	    `3 = VARCHAR*`
	    `4 = FLOAT`
	    `5 = DECIMAL †`
	    `8 = NATIVE CHAR *`
	    `9 = NATIVE VARCHAR *`
	  `10 = DATE*`
	  `11 = TIME*`
	  `12 = DATETIME*`
	  `13 = INTERVAL*`
	  `14 = VARBINARY*`
	  `15 = LONG BINARY *`
	  `16 = LONG VARBINARY*`
	
	* Native CHAR or VARCHAR is what SQLCore uses internally when a CHAR or VARCHAR column is defined with a LANG = ColumnLanguageName clause. They possess the same characteristics as the related types CHAR and VARCHAR, except that data stored in native columns will be sorted, compared, or truncated using local language rules. Native, character, and Date/Time types are compatible with regular character types.
	† When you use the DECIMAL data type with BULK processing or with format array record in dynamic processing, you must convert the DECIMAL value to its ASCII representation. Refer to the routine BCDToString in the program cex10a later in this chapter.
		SmallInt

sqlprec	precision of DECIMAL data	SmallInt
sqlscale	scale of DECIMAL data	SmallInt
sqltotallen	byte sum of sqlvallen, sqlindlen, indicator alignment bytes, and next data value alignment bytes	Integer
sqlvallen	number of bytes in data value, including a 4-byte prefix containing actual length of VARCHAR data	Integer

sqlindlen	number of bytes null indicator occupies in the data buffer:
	
	    `0 bytes:  column defined NOT NULL`
	    `2 bytes:  column allows null values`
		SmallInt

sqlvof	byte offset of value from the beginning of a row	Integer
sqlnof	byte offset of null indicator from the beginning of a row, dependent on the value of sqlindlen	Integer
sqlname	defined name of column or, for computed expression, EXPR	Packed Array [1..20] of char

Setting up the Data Buffer

You use different approaches to setting up the data buffer depending on whether your dynamic query result has an unknown format or a known format. If the query result has an unknown format, you may not know the number of columns or their data types. If the query result has a *known* format, you know in advance the number of columns in the query result and the data type of each column.

Setting up a Buffer for Query Results of Unknown Format

For query results of unknown format, you declare the data buffer as a character array:

```
const
  MaxDataBuff = 2500;
.
.
.
var
  DataBuffer : packed array[1..MaxDataBuff] of char;
```

The data buffer must be large enough to hold all the rows ALLBASE/SQL retrieves each time you execute the FETCH command, i.e., the number of rows you specify in SQLDA.SqlNRow. The data buffer defined above can hold as many as 2500 bytes of data.

Although the data buffer above can hold 2500 bytes, it would not be able to hold 2500 bytes of column values if any of the values were null and/or VARCHAR:

- If a column can contain null values, ALLBASE/SQL appends a 2-byte suffix to the data value when it puts the data into the data buffer. This suffix, referred to as a **null indicator**, contains a 0 when the data value is *not* null and a negative number when the value *is* null. You use the sqlindlen field of the format array record to determine whether ALLBASE/SQL returned this suffix with the data.

- When ALLBASE/SQL puts VARCHAR data into the data buffer, it prefixes the data with 4 bytes containing the actual length of the VARCHAR string. You use the sqltype field of the format array record to identify VARCHAR values. This field is set to 3 when data returned to the data buffer has this prefix.

You can use the SQLDA.SqlRowLen value to compute how many rows will fit into the data buffer. Dividing SQLDA.SqlRowLen into SQLDA.SqlBufLen gives you the number of rows, including any VARCHAR prefixes and null indicator suffixes accompanying data values in the row:

```
SqlNRow := SqlBufLen DIV SqlRowLen
```

The data buffer declaration shown above is an array of char, because the format of the query result is unknown at programming time.

Setting up a Buffer for Query Results of Known Format

When you know the query result format in advance, you can declare a data buffer as an array of records having the expected format. When a column can contain null values, you *must* declare a 2-byte indicator variable, immediately following the variable for that column. The indicator variable will hold the 2-byte suffix ALLBASE/SQL returns with the data value. In the following example, Column3Ind is an indicator variable for Column3.

```
DataBuffer   : Packed Array[1..MaxDataBuff] of Packed Record
               Column1      : String[20]; (* for VARCHAR data *)
               Column2      : SmallInt;
               Column3      : Integer;
               Column3Ind   : SmallInt;   (* indicator variable *)
               Column4      : Packed Array[1..60] of Char; (* for CHAR data *)
             End;
```

When a column contains a VARCHAR data type, you use a *string* data type for the variable length data, as shown above. The string data type includes a 4-byte prefix for the length of the data.

The data types you declare for a query result of known format need not be equivalent to the data types of their corresponding columns, but they should be compatible. (DATE, TIME, DATETIME, and INTERVAL values are treated like CHAR values.) Refer to the *ALLBASE/SQL Reference Manual* for the rules governing data type compatibility and conversion for complete information on this topic. The *ALLBASE/SQL Reference Manual* also addresses type conversion that may occur when a select list item is an expression containing data of different types. When you expect truncation, the column must allow nulls in order to detect the truncation.

Using the Dynamic Query Data Structures

You use the SQLDA, the format array, and the data buffer in the following sequence of operations:

■ Include the SQLDA at the beginning of your program with an INCLUDE statement:

```
EXEC SQL INCLUDE SQLDA;
```

■ Declare a data buffer to hold the query result. This may be structured or not, depending on whether you know the format of the query result in advance. The following is unstructured:

```
const
  MaxDataBuff = 2500;
.
.
.
var
  DataBuffer : packed array[1..MaxDataBuff] of char;
```

When the select list is known, you can define the data buffer as an array of records having the expected format:

```
var
  DataBuffer: packed array[1..MaxNbrRows] of packed record
                column1  : Column1DataType;
                column2  : Column2DataType;
              end;
```

■ Declare a format array as sqlformat_type. This type is defined for you in the preprocessor generated type include file. The number of records in the format array in this example is 1024, which allows for the maximum size query result of 1024 columns.

```
const
  NbrFmtRecords = 1024;              (*columns expected*)
var
     (*SQLFmts is the format array*)
  SQLFmts          : array[1..NbrFmtRecords] of SQLFormat_Type;
```

■ Use a host variable for the SELECT command, and pass it to ALLBASE/SQL in the PREPARE command:

```
EXEC SQL BEGIN DECLARE SECTION;
DynamicCommand       : string[1024];
EXEC SQL END DECLARE SECTION;
 .
 .
 .
EXEC SQL PREPARE Cmd1 FROM :DynamicCommand;
```

■ Initialize two SQLDA fields, sqln and sqlfmtarr. sqln is set to the number of records of the format array, and sqlfmtarr is set to its address.

```
with SQLDA do
  begin
    sqln       := NbrFmtRecords;       (* columns expected*)
    sqlfmtarr := waddress(SQLFmts);   (* format array address*)
  end;
```

■ Execute the DESCRIBE command:

```
EXEC SQL DESCRIBE Cmd1 INTO SQLDA;
```

During the execution of the DESCRIBE command, ALLBASE/SQL returns to the format array and to the SQLDA the information you need later to parse and handle the query result. You use format array information to parse the data buffer when you do not know in advance the format of a query result.

Note	When you know the format of the query result in advance, you can define a data buffer having the format you expect, and you do not need to use format array information to parse it. However, you still need to declare the format array.

■ Declare and open a cursor for the prepared query:

```
EXEC SQL DECLARE Cursor1 CURSOR FOR Cmd1;
EXEC SQL OPEN Cursor1;
```

■ Before retrieving rows into the data buffer, initialize three SQLDA fields. These fields identify your data buffer and specify how many rows you want retrieved into the data buffer each time the FETCH command is executed:

```
with SQLDA do
  begin
    sqlbuflen := sizeof(DataBuffer);     (* bytes in data buffer *)
    sqlrowbuf := waddress(DataBuffer);   (* data buffer address *)
    sqlnrow   := sqlbuflen DIV sqlrowlen; (* number of rows to FETCH *)
  end;
```

■ Execute the FETCH command. ALLBASE/SQL packs the data buffer with as many rows from the active set as you specified in SQLDA.SqlNRow. ALLBASE/SQL puts the first select list value into the data buffer, starting at the first byte of the format array and including any VARCHAR prefixes, ALLBASE/SQL null indicators for columns that can contain null values, and any alignment bytes provided by the Pascal compiler. Then ALLBASE/SQL writes the second through last select list values for the first row. If the query result contains another row, the first through last select list values in that row are written to the data buffer. Data values are thus concatenated in the data buffer until the last row has been fetched. When the last row in the active set has been fetched, ALLBASE/SQL sets SQLCA.SQLCODE to 100.

In Figure 10-5, two columns are selected from the vendors table in the sample database. Column VendorNumber is defined in the table as an INTEGER that cannot contain a null value. Column VendorRemarks is defined in the table as a VARCHAR that can contain a null value. Since the VendorRemarks column can contain a null value, a two byte null indicator needs to be provided immediately following this VARCHAR data column. Note the two byte filler that completes the VendorRemarks column definition. It is needed by the compiler for byte alignment purposes; data is aligned on 4 byte boundaries.

300/400   Character array data is aligned on 2 byte boundaries in Series 300 and 400 systems.

The figure illustrates the relationships between column definitions and the layout of data in the data buffer for a Series 800 system.
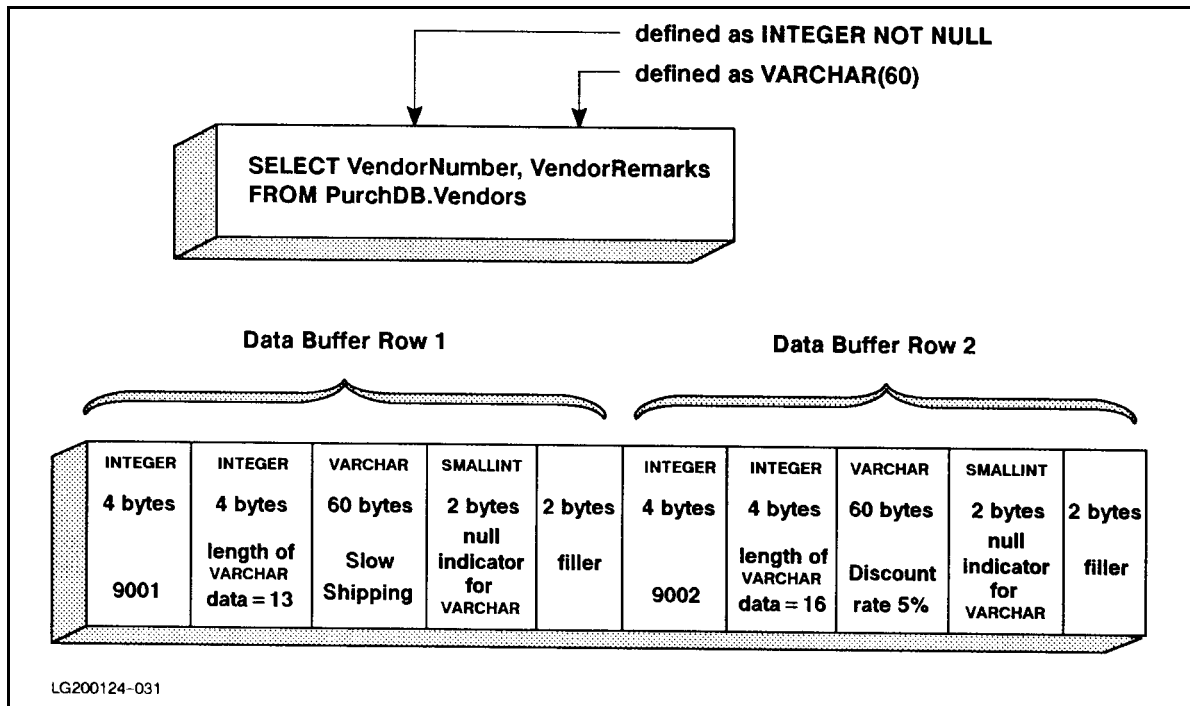
Figure 10-5. Format of the Data Buffer for a Series 800

Note that the number of rows to retrieve with each execution of the FETCH command is specified in SQLDA.SqlNRow. As shown in the above example, you can calculate the number of rows that will fit into the data buffer by dividing the row length (in bytes) into the number of bytes in the data buffer. Sqlrowlen, one of the SQLDA fields set by ALLBASE/SQL when you execute the DESCRIBE command, contains the number of bytes in each row.

```
while SQLCA.SQLCODE = 0 do
begin
  EXEC SQL FETCH Cursor1 USING DESCRIPTOR SQLDA;
  DisplayRow;
end;
```

■ If the query result is of unknown format, parse rows out of the data buffer after each execution of the FETCH command. The technique for parsing is shown in detail in the next section.

Parsing the Data Buffer

The technique for parsing the data buffer and assigning its contents to variables of appropriate types is illustrated in function DisplaySelect of program pasex10a. The listing is found in Figure 10-9 in the following section, "Program pasex10a: Dynamic Commands of Unknown Format." Essentially, you initialize an offset variable for the data buffer, then execute a loop for each row retrieved with the FETCH statement. For each column in the loop, you do the following:

■ Check for null values, taking appropriate action when one is found.

■ Examine the data type and length of the data element itself, assigning it to an appropriate variable of the corresponding size. A dynamically preprocessed PREPARE statement with an output data buffer requires you code Pascal statements yourself to convert Binary Coded Decimal (BCD) representation to character representation. If you use an input buffer with dynamic preprocessing, you must write code that converts the character representation to BCD format before the data is placed in the input buffer.

■ Increment the offset variable by the value of SQLDA.SqlRowLen (the length of a complete row).

The following diagram summarizes the arithmetic used to parse the data buffer in function DisplaySelect in program pasex10a. The data buffer shown is for the first query executed in the dialog in Figure 10-6.
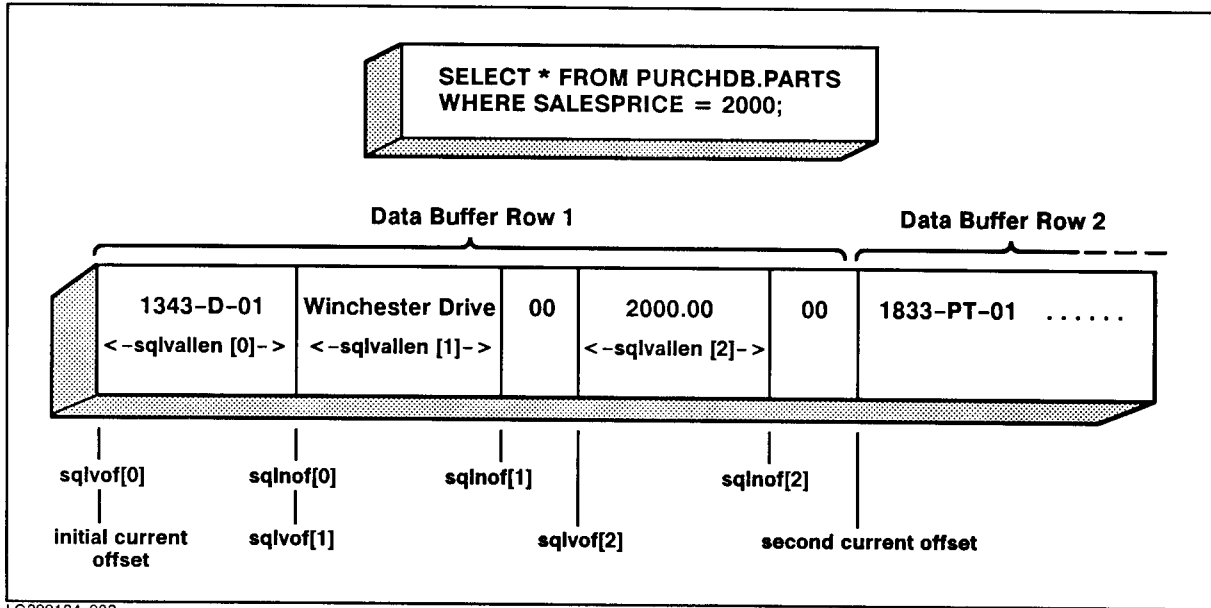
Figure 10-6. Parsing the Data Buffer in Program pasex10a

Program pasex10a uses the following assignment to set the start of a row:

```
CurrentOffset := CurrentOffset + SqlRowLen;
```

To find a null indicator, the program uses the following assignment:

```
NullIndOffset := CurrentOffset + SqlNOf;
```

To move a data value into a variant record, pasex10a uses the following statement:

```
StrMove(SqlValLen, DataBuffer,
CurrentOffset + SqlVOf, OneColumn.CharData, 1);
```

Preprocessing Dynamic Commands That May or May Not Be Queries

You need special techniques to handle dynamic commands which may be either queries or non-queries. In a program that accepts both query and non-query SQL commands, you first PREPARE the command, then use the DESCRIBE command in conjunction with the SQLDA, the data structure that lets you identify whether a command is a query. The PREPARE command must appear physically in your source program *before* the EXECUTE or DECLARE CURSOR command that uses the name you assign to the dynamic command in the PREPARE command.

The sqld field of the SQLDA is set to 0 if the dynamic command is not a query and to a positive integer if it is a query. The SQLDA data structure is used in any program that may host a dynamic query.

In the following example, if the command is not a query, you branch to NonQuery and use the EXECUTE or EXECUTE IMMEDIATE command to execute it. If it is a query, you branch to Query, where you declare a cursor, open it, then use FETCH to retrieve qualifying rows.

```
EXEC SQL PREPARE ThisCommand FROM :DynamicCommand;

EXEC SQL DESCRIBE ThisCommand INTO SQLDA;
```

> *The SQLDA.SQLD field of the SQLDA is set to 0 if the dynamic command*
> *is not a query and to a positive integer if it is a query. The*
> *SQLDA is a special data structure used in any program that may host*
> *a dynamic query. The data structure is fully defined in this*
> *section under "Setting Up the SQLDA."*

```
if SQLDA.SQLD = 0 then NonQuery;
```

> *The command is not a query and the EXECUTE or*
> *EXECUTE IMMEDIATE command is used to execute it.*

```
else if SQLDA.SQLD > 0 then Query;
```

> *The command is a query and a cursor is used*
> *to retrieve qualifying rows.*

To handle a command entirely unknown at programming time, you accept the command into the host variable. In the following example, an SQL command is accepted into a host variable named DynamicCommand, which is declared large enough to accommodate the largest expected dynamic command. User input is accepted into DynamicClause and concatenated in DynamicCommand until the user enters a semicolon.

```
var
  EXEC SQL BEGIN DECLARE SECTION;
  DynamicCommand     := String[1024];
  EXEC SQL END DECLARE SECTION;
  DynamicClause      := String[80];
  Pos                := SmallInt;
.
.
writeln ('Enter an SQL command or clause > ');
writeln;
DynamicCommand := '';
repeat
  prompt('> ');
  readln(DynamicClause);
  if DynamicClause <> '/' then
    begin
      DynamicCommand := DynamicCommand + ' ' + DynamicClause;
      Pos := StrPos(DynamicClause, ';');
      if Pos <> 0 then DynamicClause := '/';
    end
    else
      DynamicCommand := '/';
until DynamicClause = '/';
.
.
EXEC SQL PREPARE SQLCommand FROM :DynamicCommand;
```

Programs Using Dynamic Command Operations

The rest of this chapter contains sample programs that illustrate the use of dynamic preprocessing techniques for commands. There are two complete programs:

- Program pasex10a, which contains statements for executing *any* dynamic command (non-query or query with unknown format).

- Program pasex10b, which contains statements for executing dynamic queries of known format.

For each program, there is a description of the code, a display of the runtime dialog with user input, and a listing.

pasex10a: Program Using Dynamic Commands of Unknown Type

Pasex10a contains the logic to determine whether or not a given dynamic command is a query or non-query, and if it is a query, it uses format array information to parse the data buffer. A flow chart for this program appears in Figure 10-7. The runtime dialog and source code are shown in Figures 10-8 and 10-9, respectively.

Pasex10a starts a DBE session (37) in the sample database in function ConnectDBE (4), then executes the procedure named Describe (23). This procedure:

- Initializes the two SQLDA fields (24) that must be set before executing the DESCRIBE command: SQLDA.Sqln (the number of elements in the format array) and SQLDA.SqlfmtArr (the address of the format array). The number of elements in the format array is defined in the constant NbrFmtRecords, set to 1024 in this program to accommodate the maximum number of columns in any query result.

- Calls procedure GetCommand (25) and processes commands accepted from the user in that procedure until the user enters a slash (/).

Procedure GetCommand (21) accepts SQL commands into the host variable named DynamicCommand. Since the maximum allowable dynamic command is 2048 bytes, including the semicolon, this variable is declared (1) as String[2048]. GetCommand concatenates multiple lines of user input by accepting each line into a local variable, DynamicClause and adding it to the contents of DynamicCommand until the user enters a semicolon; the string function STRPOS is used to detect this character.

After SQL command entry is complete, control returns to procedure Describe (23), which:

- Starts a transaction, in function BeginTransaction (6).

- Executes the PREPARE (26) and DESCRIBE (27) commands.

- Examines the SQLDA.Sqld field (number of columns in query result) to determine whether the dynamic command is a query (28). If this value is 0, the command is not a query and procedure NonQuery (29) is invoked to execute the command. If the SQLDA.Sqld value is not 0, procedure Query (30) is invoked to execute the command.

Note that the FORWARD directive (22) is used for procedures NonQuery and Query, just prior to Describe. These procedures cannot be fully declared until *after* procedure Describe. You must name a dynamic command (in the PREPARE command) before you reference it (in the EXECUTE or DECLARE CURSOR commands). In this program, the PREPARE command is executed in procedure Describe, which calls both NonQuery and Query.

Procedure Query:

- Displays the number of columns in the query result, by using the value ALLBASE/SQL assigned to SQLDA.Sqld when the DESCRIBE command was executed ⌊31⌋.

- Declares and opens a cursor for the dynamic query ⌊32⌋.

- Initializes the three SQLDA fields that must be set before executing the FETCH command ⌊33⌋: SQLDA.SqlBufLen (the size of the data buffer), SQLDA.SqlNRow (the number of rows to put into the data buffer with each FETCH), and SQLDA.SqlRowBuf (the address of the data buffer).

  Note that to set SQLDA.SqlNRow, the program divides the row length into the data buffer size to determine how many rows can fit into the data buffer ⌊34⌋.

- Executes the FETCH command ⌊35⌋ and calls procedure DisplaySelect ⌊36⌋ until the last row in the active set has been fetched. When no more rows are available to fetch, ALLBASE/SQL sets SQLCA.SQLCODE to 100, defined as a constant named EOF in this program.

Procedure DisplaySelect ⌊8⌋ parses the data buffer after each FETCH operation and displays rows:

- The procedure keeps track of the beginning of each row by using a local variable, CurrentOffset, as a pointer. CurrentOffset is initialized to 1 ⌊10⌋ at the beginning of procedure DisplaySelect.

- Column headings are written from the SQLName field of each format array record ⌊11⌋. The loop that displays the headings uses the SQLDA.Sqld value (the number of columns in the query result) as the final value of a format array record counter (x).

- The first through last column values in each row are examined and displayed in a loop. The loop uses the SQLDA.SqlRRow value (the number of rows fetched) as the final value of a row counter ⌊12⌋. The loop also uses the SQLDA.Sqld value (the number of select list items) as the final value of a column counter ⌊13⌋.

- The SqlIndLen field of each column's format array record is examined ⌊14⌋ to determine whether a null value might exist.

- If a column can contain null values, SqlIndLen is greater than zero, and the procedure must examine the indicator variable to determine whether a value is null. A local variable, NullIndOffset, is used to keep track of the first byte of the current indicator variable ⌊15⌋.

- Any null indicator can be located by adding the current value of SqlNOf to the current value of CurrentOffset. SqlNOf is the format array record field that contains the byte offset of a null indicator from the beginning of a row. Recall that CurrentOffset keeps track of the beginning of a row.

- The Pascal ORD function and NullIndOffset are used to determine whether the indicator variable contains zeros ⌊16⌋. If it does, the value is null, and the procedure displays the following message at ⌊17⌋:

      Column is NULL

- If a value is not null, it is moved ⌊18⌋ from the data buffer to OneColumn.CharData. The starting location of a value in the STRMOVE procedure is computed by adding the current value of SqlVOf to the current value of CurrentOffset. SqlVOf is the format array record field that contains the byte offset of a value from the beginning of a row. The number

of bytes to move is the value stored in SqlValLen. OneColumn.CharData is one of the variations of a variant record, GenericColumnType ⑨.

■ GenericColumnType is used to write data values. This variant record has a record definition describing a format for writing data of each of the ALLBASE/SQL data types. The record variation used depends on the value of SqlType ⑲, the format array record field describing the data type of a select list item. In the case of DECIMAL data, a function named BCDToString ② converts the binary coded decimal (BCD) information in the data buffer into ASCII format for display purposes.

■ After each value in a row is displayed, CurrentOffset is incremented by SQLDA.SqlRowLen ⑳ to point to the beginning of the next row.

When the dynamic command has been completely processed, procedure Query calls the EndTransaction procedure ⑦ to process a COMMIT WORK command. Thus each dynamic query hosted by this program is executed in a separate transaction.

To determine whether each SQL command executed successfully, the program examines the value of SQLCA.SQLCODE after SQL commands are executed. Procedure SQLStatusCheck ③ is invoked to display one or more messages from the ALLBASE/SQL message catalog. Any other action taken depends on the SQL command:

■ If the CONNECT command fails, function ConnectDBE ④ sets the ConnectDBE flag to FALSE, then calls procedure SQLStatusCheck. Then the program terminates.

■ If the BEGIN WORK command fails, function BeginTransaction ⑥ calls SQLStatusCheck to display messages, then calls ReleaseDBE ⑤ to end the DBE session. The program then terminate because procedure Describe ㉓ sets DynamicCommand to a slash.

■ If other SQL commands fail, procedure SQLStatusCheck terminates the program whenever the error is serious enough to return an SQLCA.SQLCODE less than -14024.
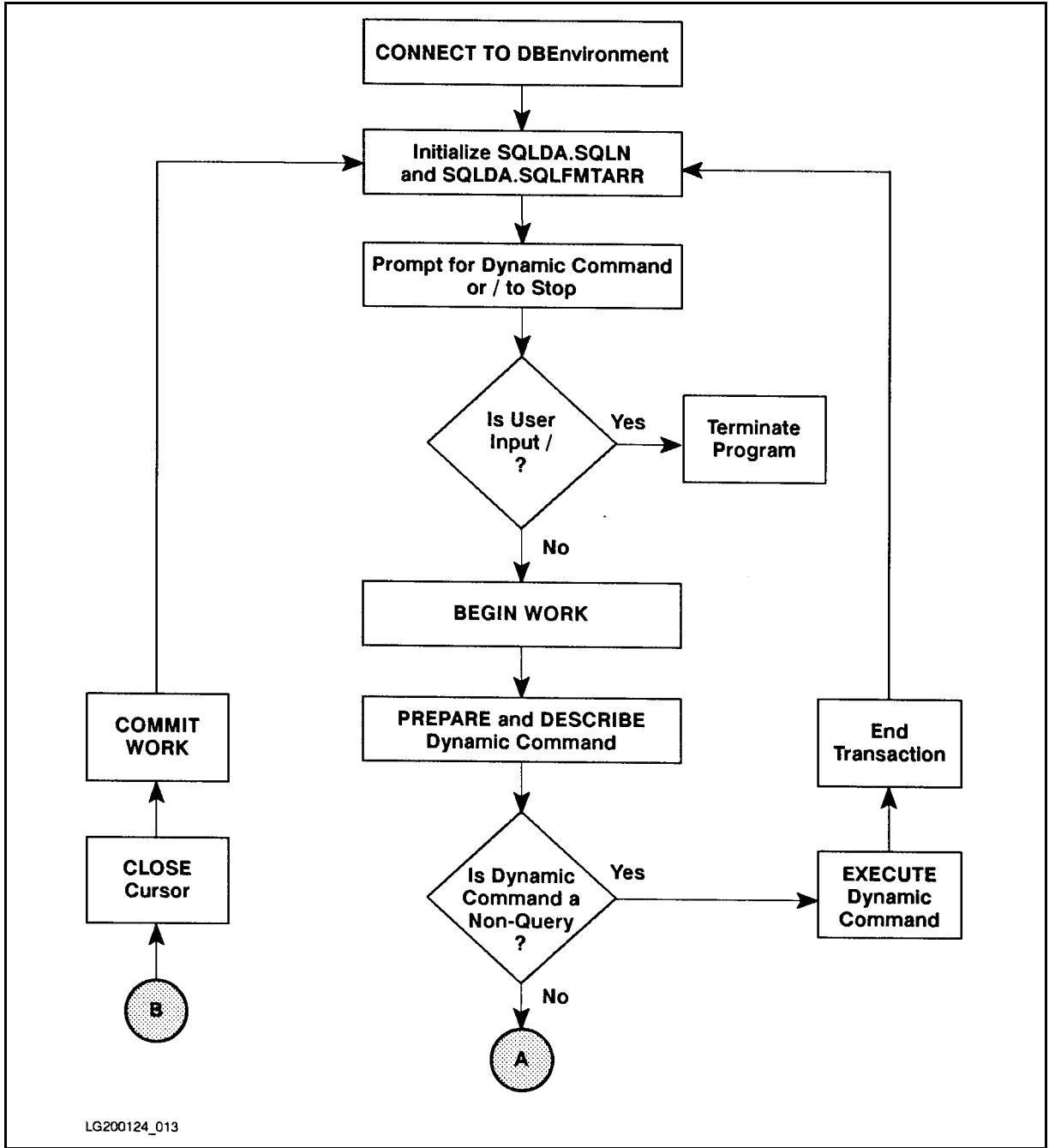
Figure 10-7. Flow Chart of Program pasex10a
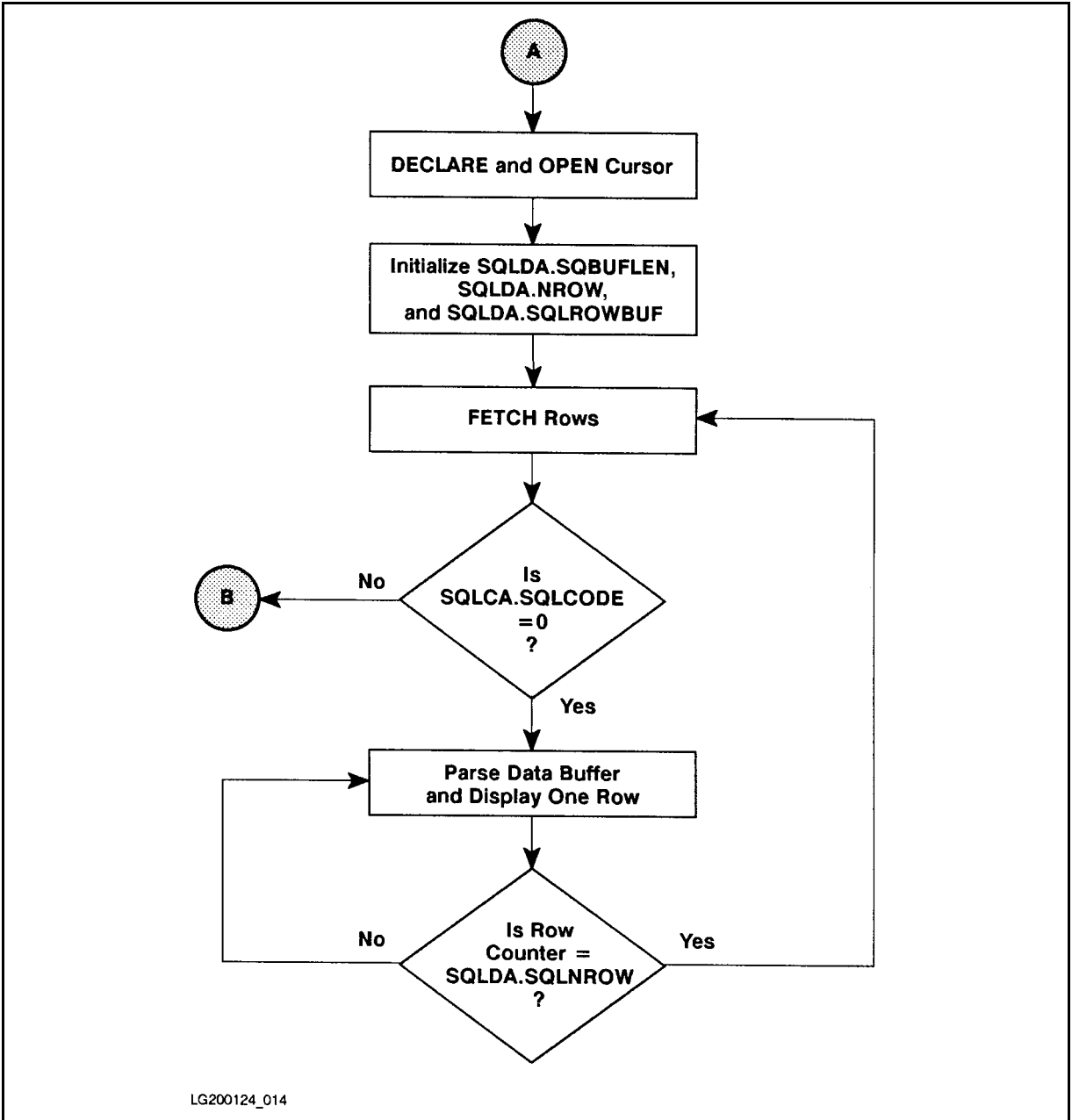
Figure 10-7. Flow Chart of Program pasex10a (page 2 of 2)

```
   Pascal program illustrating dynamic command processing.

   Event List:
     Connect to PartsDBE
     Prompt for any SQL command
     Begin Work
     Prepare
     Describe
     If command is a non-query command, EXECUTE it
     Otherwise execute the following:
     Declare
     Open
     Fetch
     Close
     Commit Work
     Repeat the above ten steps
     Release PartsDBE

   Connect to PartsDBE
     Connect to PartsDBE

   You may enter any SQL command or "/" to STOP the program.
   The command can be continued on the next line.  The command
   must be terminated with a semicolon (;).

   Enter SQL command/clause >

   > SELECT * FROM PURCHDB.PARTS WHERE SALESPRICE = 2000;

   Begin Work
   Prepare
   Describe
   Query SQL command.

   Number of columns:  3

   PARTNUMBER             | PARTNAME              | SALESPRICE          |
   1343-D-01        | Winchester Drive             |          2000.00 |

   Row not found or no more rows

   Commit Work

   You may enter any SQL command or "/" to STOP the program.
   The command can be continued on the next line.  The command
   must be terminated with a semicolon (;).
```

Figure 10-8. Runtime Dialog of Program pasex10a

```
Enter SQL command/clause >

> DELETE FROM PURCHDB.PARTS WHERE PARTNUMBER = '1343-D-01';

Begin Work
Prepare
Describe
Non Query SQL command.
Execute
Non-Query Command Executed Successfully.

Commit Work

You may enter any SQL command or "/" to STOP the program.
The command can be continued on the next line.  The command
must be terminated with a semicolon (;).

Enter SQL command/clause >

> SELECT * FROM PURCHDB.PARTS WHERE SALESPRICE = 2000;

Begin Work
Prepare
Describe
Query SQL command.

Number of columns:  3

Row not found or no more rows

Commit Work

You may enter any SQL command or "/" to STOP the program.
The command can be continued on the next line.  The command
must be terminated with a semicolon (;).

Enter SQL command/clause >

> /
Release PartsDBE
```

Figure 10-8. Runtime Dialog of Program pasex10a (page 2 of 2)

```
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* This program illustrates dynamic preprocessing of SQL commands  *)
(* including SELECT commands using the DESCRIBE command.           *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

Program pasex10a (input, output);
type

(* Nibbles and BCDType are data types needed for decimal type *)
  Nibbles  = 0..15;
  BCDType = packed array [1..20] of Nibbles;

Const
  NotFound     =    100;
  OK           =      0;
  DeadLock     = -14024;

(* NbrFmtRecords is number of columns expected in a dynamic SELECT. *)
  NbrFmtRecords =  1024;
  EOF           =    100;
  MaxDataBuff   =   2500;

Var

          (* Begin Host Variable Declarations *)
  EXEC SQL BEGIN DECLARE SECTION;
  SQLMessage        : packed array[1..132] of char;
(* DynamicCommand is a String that will hold the dynamic command. *)
  DynamicCommand              : String[2048]; (1)
  EXEC SQL END DECLARE SECTION;
          (* End Host Variable Declarations *)

  EXEC SQL INCLUDE SQLCA;

(* SQLDA is the SQL DESCRIBE Area used by the DESCRIBE command. *)
  EXEC SQL INCLUDE SQLDA;

(* Each record in SQLFmts will hold information about each column
 * in a dynamic SELECT.  *)
  SQLFmts : array[1..NbrFmtRecords] of SqlFormat_Type;

(* DataBuffer is the buffer containing retrieved data as a result
 * of a dynamic SELECT.  *)
  DataBuffer : packed array[1..MaxDataBuff] of char;
```

Figure 10-9. Program pasex10a: Dynamic Commands of Unknown Format

```
    Abort              : boolean;
$PAGE $
(* Procedure BCDToString converts a decimal field in the "DataBuffer"
 * buffer to its decimal presentation.  Other input parameters are
 * the Length, precision and Scale.  The input decimal field is passed
 * via "DataBuffer" and the output String is passed via "result".
 *)
procedure BCDToString (DataBuffer : BCDType; Length : SmallInt;(2)
                       Precision : SmallInt; Scale : SmallInt;
                       var Result : String);
const
  hexd        = '0123456789ABCDEF';    (* Hexadecimal digits #001*)
  ASCIIZero = ord('0');
  PlusSign  = 12;
  MinusSign = 13;
  UnSigned  = 14;
var
  i,
  DecimalPlace,
  PutPos,
  DataEnd,
  DataStart : Integer;
  done  : boolean;

begin
  DataEnd := (Length*2) - 1;
  DataStart := (DataEnd - Precision) + 1;
  Result := StrRpt (' ',StrMax(Result));
  DecimalPlace := Precision-Scale;

(* convert decimal to character String *)
  if DecimalPlace = 0 then
    begin

      Result[1] := '.';
      PutPos := 2;

    end
  else
    PutPos := 1;
    for i := DataStart to DataEnd do
    begin
```

Figure 10-9. Program pasex10a: Dynamic Commands of Unknown Format (page 2 of 12)

```
    (* convert each Nibble into a character *)
        Result[PutPos] := chr(ASCIIZero + DataBuffer[i]);
        if PutPos = DecimalPlace then
        begin
          PutPos := succ(PutPos);
          Result[PutPos] := '.';
        end;
        PutPos := succ(PutPos);
    end;
$PAGE $
(* convert leading zeroes to spaces *)
    Result := StrLTrim(StrRTrim(Result));
    i := 1;
    done := False;
    while (i <= StrLen(Result)) AND (not done) do
    if Result[i] <> '0' then
      done := True
    else
      begin
        Result[i] := ' ';
        i := succ(i);
      end;
(* trim spaces from result  *)
    Result := StrLTrim(Result);
    if Result = '' then
      Result := '0'
    else
      begin
        if Result[1] = '.' then
(* place a zero at the left of the decimal point *)
      StrInsert('0', Result, 1);
(* insert sign *)
      case DataBuffer[DataEnd + 1] of
        PlusSign : StrInsert(' ', Result, 1);
        MinusSign: StrInsert('-', Result, 1);
      end; (*case*)
    end; (*else*)
end; (*BCDToString*)
$PAGE $
procedure SQLStatusCheck; (*Procedure to Display Error Messages*)(3)
begin
Abort := FALSE;
if SQLCA.SQLCODE < DeadLock then Abort := TRUE;
repeat
EXEC SQL SQLEXPLAIN :SQLMessage;
writeln(SQLMessage);
until SQLCA.SQLCODE = 0;
```

Figure 10-9. Program pasex10a: Dynamic Commands of Unknown Format (page 3 of 12)

```
if Abort then
  begin
  EXEC SQL COMMIT WORK RELEASE;
  halt;
  end;
end;  (* End SQLStatusCheck Procedure *)

function ConnectDBE:boolean;(*Function to Connect to PartsDBE *)   ④
 begin

writeln('Connect to PartsDBE');
EXEC SQL CONNECT TO '../sampledb/PartsDBE';

ConnectDBE := TRUE;
if SQLCA.SQLCODE & OK then
  begin
  ConnectDBE := FALSE;
  SQLStatusCheck;
  end;  (* End if *)
end;  (* End of ConnectDBE Function *)

procedure ReleaseDBE;   (* Procedure to Release PartsDBE *)      ⑤
begin

writeln('Release PartsDBE');
EXEC SQL RELEASE;

if SQLCA.SQLCODE <> OK then SQLStatusCheck;

end;  (* End ReleaseDBE Function *)
$PAGE $

function BeginTransaction: boolean;  (* Function to Begin Work *)⑥
begin

writeln;
writeln('Begin Work');
EXEC SQL BEGIN WORK;
if SQLCA.SQLCODE & OK then
  begin

  BeginTransaction := FALSE;
  SQLStatusCheck;
  ReleaseDBE;
  end
```

Figure 10-9. Program pasex10a: Dynamic Commands of Unknown Format (page 4 of 12)

```
    else
      BeginTransaction := TRUE;

    end;  (* End BeginTransaction Function *)


procedure EndTransaction;  (* Procedure to Commit Work *)      (7)
begin

    writeln;
    writeln('Commit Work');
    EXEC SQL COMMIT WORK;
    if SQLCA.SQLCODE & OK then SQLStatusCheck;
end;  (* End Transaction Procedure *)
$PAGE $
(* Procedure DisplaySelect deblocks the result of the dynamic
 * SELECT in "DataBuffer".   *)
procedure DisplaySelect;                                        (8)
const
    MaxColSize   = 3996;

type

    GenericColumnType = record                                 (9)
      case SmallInt of
        0 : (CharData      : packed array[1..MaxColSize] of char);
        1 : (VarCharData   : String[MaxColSize]);
        2 : (IntegerData   : Integer);
        3 : (SmallIntData  : SmallInt);
        4 : (FloatData     : LongReal);
        5 : (DecimalData   : BCDType);
      end;
var
    CurrentOffset  : SmallInt;
    NullIndOffset  : SmallInt;
    OneColumn      : GenericColumnType;
    DecString      : string[20];
    IsNull         : Boolean;
    n,i,j,x        : SmallInt;                  (* local loop counters   *)
```

Figure 10-9. Program pasex10a: Dynamic Commands of Unknown Format (page 5 of 12)

```
$PAGE $
begin

CurrentOffset := 1;                                        ⑩

for x := 1 to SQLDA.Sqld do    (* display column names *)
  with SQLFmts[x] do
  begin
    write(SqlName, ' | ');                                 ⑪
  end;
writeln;

for n:= 1 to SQLDA.SqlRRow do     (* for each FETCHed row *) ⑫
begin

 for i:=1 to SQLDA.Sqld do (* for each column in a FETCHed row *)⑬
    with SQLFmts[i] do
    begin

  (* Check to see if this column has the value NULL.  This is done *)
  (* by checking the NULL indicator in the buffer.  This indicator *)
  (* appears after the data value for this column.                 *)

    IsNull := False;
    if SqlIndLen > 0 then                                  ⑭
      begin

        NullIndOffset := CurrentOffset + SqlNOf;           ⑮

        if (ord(DataBuffer[NullIndOffset]) = 0)            ⑯
          AND (ord(DataBuffer[NullIndOffset+1]) = 0) then
          IsNull := False
        else
          IsNull := True;

      end;                              (* end if SQLIndLen > 0 .. *)

        if IsNull then
          write('Column is NULL | ')                       ⑰
        else
          begin
            (* Now bring down the actual value of this column.  *)

              StrMove(SqlValLen, DataBuffer,                ⑱
                    CurrentOffset + SqlVOf, OneColumn.CharData, 1);
```

Figure 10-9. Program pasex10a: Dynamic Commands of Unknown Format (page 6 of 12)

```
$PAGE $
            case SqlType of                                    (19)
              0:                      (* Integer number        *)
                case SqlValLen of
                  2: write(OneColumn.SmallIntData, ' | ');
                  4: write(OneColumn.IntegerData, ' | ');
                end;
              2,8:                    (* fixed-length character *)
                begin
                  for j := 1 to SqlValLen do
                    write(OneColumn.CharData[j]);
                    write(' | ');
                end;
              3,9:                    (* variable-length char   *)
                begin
                  write(OneColumn.VarCharData, ' | ');
                end;
              4:                      (* floating point         *)
                begin
                  write(OneColumn.FloatData, ' | ');
                end;
              5:                      (* Packed decimal         *)
                begin

                  BCDToString(OneColumn.DecimalData, SqlValLen,
                    SqlPrec, SqlScale, DecString);
                  write(DecString:SqlValLen*2, ' | ');

                end;
              otherwise writeln ('SQLType = ',sqltype1);
            end;                              (* case statement        *)

        end;                                  (* if IsNull             *)

      end;                          (* for i/with SQLFmts[i] ...     *)

      CurrentOffset := CurrentOffset + SQLDA.SqlRowLen;   (20)

    writeln;

  end;                                        (* for n := ...          *)

  writeln;

end;                                          (* end of DisplaySelect *)
```

Figure 10-9. Program pasex10a: Dynamic Commands of Unknown Format (page 7 of 12)

```
$PAGE $
procedure GetCommand;                                              (21)

var
  DynamicClause                 : String[80];
  Pos                           : SmallInt;

begin

writeln;
writeln('You may enter any SQL command or "/" to STOP the program.');
writeln('The command can be continued on the next line.  The command');
writeln('must be terminated with a semicolon (;).');
writeln;
writeln('Enter SQL command/clause >');
writeln;
DynamicCommand := '';
repeat
prompt('> ');
readln(DynamicClause);
if DynamicClause <> '/' then
  begin
    DynamicCommand := DynamicCommand + ' ' + DynamicClause;
    Pos := StrPos(DynamicClause, ';');
    if Pos <> 0 then DynamicClause := '/';
  end
  else
    DynamicCommand := '/';
until DynamicClause = '/';  (* end repeat *)
end;  (* end of GetCommand procedure *)
$PAGE $
procedure NonQuery;forward;                                        (22)
procedure Query;forward;
procedure Describe;   (* Describe Procedure *)                     (23)
begin

with SQLDA do   (* set up SQLDA fields *)
  begin
    Sqln      := NbrFmtRecords; (* number of columns expected *) (24)
    SqlFmtArr := waddress(SQLFmts);
  end;
```

Figure 10-9. Program pasex10a: Dynamic Commands of Unknown Format (page 8 of 12)

```
repeat
  GetCommand;                                                      ⌊25⌋

  if DynamicCommand <> '/' then
    begin

      if BeginTransaction then
        begin

          writeln('Prepare');
          EXEC SQL PREPARE CMD1 FROM :DynamicCommand;              ⌊26⌋
          if SQLCA.SQLCODE <> OK then
          begin
            SqlStatusCheck;
            EndTransaction;
          end

          else
          begin

            writeln('Describe');
            EXEC SQL DESCRIBE CMD1 INTO SQLDA;                     ⌊27⌋
            if SQLCA.SQLCODE <> OK then
              begin
                SqlStatusCheck;
                EndTransaction;
              end
              else
              begin
                if SQLDA.Sqld = 0 then NonQuery                   ⌊28⌋
                else Query;
              end; (* end if SQLCA.SQLCODE <> OK after DESCRIBE *)

        end; (* end if SQLDA.SQLCODE <> OK after PREPARE *)

      end  (* end if BeginTransaction *)
      else                        (* BeginTransaction failed; force *)
        DynamicCommand := '/';    (* logical end to *)
                                  (* Describe procedure *)

    end;  (* end if DynamicCommand *)

  until DynamicCommand = '/';   (* end repeat *)

end;  (* end of Describe procedure *)
```

Figure 10-9. Program pasex10a: Dynamic Commands of Unknown Format (page 9 of 12)

```
$PAGE $
procedure NonQuery;                                                    [29]
begin

writeln ('Non Query SQL command.');
writeln ('Execute');
EXEC SQL EXECUTE CMD1;
if SQLCA.SQLCODE <> OK then
begin
  SqlStatusCheck;
  EXEC SQL ROLLBACK WORK;
end
else
  begin
  writeln ('Non-Query Command Executed Successfully.');
  EndTransaction;
  end;

end;  (* end of NonQuery procedure *)

$PAGE $
procedure Query;                                                       [30]

var
  RowLength          : SmallInt;
  i                  : SmallInt;

begin

writeln ('Query SQL command.');
writeln;
writeln('Number of columns: ',SQLDA.Sqld:2);                           [31]
writeln;

EXEC SQL DECLARE CURSOR1 CURSOR FOR CMD1;                              [32]
EXEC SQL OPEN CURSOR1;
if SQLCA.SQLCODE <> OK then SQLStatusCheck
else
begin
with SQLDA do
  begin
    SqlBufLen := sizeof(DataBuffer);                                   [33]

    SqlNRow := SqlBufLen DIV SqlRowLen;                                [34]

    SqlRowBuf := waddress(DataBuffer);
  end;
```

Figure 10-9. Program pasex10a: Dynamic Commands of Unknown Format (page 10 of 12)

```
   while SQLCA.SQLCODE = 0 do
   begin
   EXEC SQL FETCH CURSOR1 USING DESCRIPTOR SQLDA;            (35)
   if SQLCA.SQLCODE <> OK then
     begin
       if SQLCA.SQLCODE = EOF then
         writeln('Row not found or no more rows')
       else
         SQLStatusCheck;
     end
     else
       DisplaySelect;                                       (36)
   end;  (* end of while SQLCA.SQLCODE = 0 *)

   EXEC SQL CLOSE CURSOR1;
   if SQLCA.SQLCODE & 0 then SqlStatusCheck;
   end;  (* end of if OPEN CURSOR is OK *)

   EndTransaction;

end;  (* end of Query procedure *)



$PAGE $
begin  (* Beginning of Program *)

writeln('Pascal program illustrating dynamic command processing.');
writeln;
writeln('Event List:');
writeln('  Connect to PartsDBE');
writeln('  Prompt for any SQL command ');
writeln('  Begin Work');
writeln('  Prepare ');
writeln('  Describe ');
writeln('  If command is a non-query command, EXECUTE it');
writeln('  Otherwise execute the following:');
writeln('  Declare ');
writeln('  Open ');
writeln('  Fetch ');
writeln('  Close ');
writeln('  Commit Work');
writeln('  Repeat the above ten steps');
writeln('  Release PartsDBE');
writeln;
```

Figure 10-9. Program pasex10a: Dynamic Commands of Unknown Format (page 11 of 12)

```
  if ConnectDBE then                                    37
    begin

      Describe;
      ReleaseDBE

    end

    else
    writeln('Error: Cannot Connect to PartsDBE');

end.   (* End of Program *)
```

Figure 10-9. Program pasex10a: Dynamic Commands of Unknown Format (page 12 of 12)

pasex10b: Program Using Dynamic Queries of Known Format

In some applications, you may know the format of a query result in advance, but want to dynamically preprocess the query to create a program that does not have a permanently stored module. Database administration utilities that include system catalog queries often fall into this category of application.

In programs hosting dynamic queries having query results of known format, you do not need to use the format array to parse the data buffer. Because you know in advance the query result format, you can pre-define an array having a complementary format and read information from the array without having to determine where data is and the format it has been returned in.

An example of this type is program pasex10b. A flowchart for pasex10b is shown in Figure 10-10, the runtime dialog is in Figure 10-11, and the source code is in Figure 10-12. Pasex10b executes two dynamic queries with select lists known at programming time. The program reads the SYSTEM.TABLE view and the SYSTEM.COLUMN view in order to re-create the SQL CREATE TABLE commands originally used to define tables in a DBEnvironment. The CREATE TABLE commands are stored in a file you name when you execute the program. Such a file can be used as an ISQL command file in order to re-create the tables in some other DBEnvironment.

The program first prompts ⑥ for the name of the file in which to store the table definitions. It purges ⑦ any file that exists by the same name.

The program then prompts for a DBEnvironment name ⑧. The DBEnvironment name is used to build a CONNECT command in host variable CmdLine ⑨. The CONNECT command is executed by using the EXECUTE IMMEDIATE command ⑩.

The program then prompts for an owner name ⑪. If an owner name is entered, it is upshifted ⑫, then added to the WHERE clause in the first dynamic query ⑭:

```
CmdLine := 'SELECT OWNER, NAME, DBEFILESET, RTYPE FROM SYSTEM.TABLE'
    + ' WHERE TYPE = 0 AND OWNER = ''' + OwnerName + ''';';
```

This query retrieves a row for every table $(TYPE = 0)$ having an owner name as specified in the variable OwnerName. Each row consists of four columns: the owner name, the table name, the name of the DBEFileSet with which the table is associated, and the automatic locking mode.

To obtain a definition of all tables in a DBEnvironment *except* those owned by SYSTEM, the user presses the carriage return in response to the owner name prompt. In this case, the program uses the following form of the dynamic query ⑬:

```
CmdLine := 'SELECT OWNER, NAME, DBEFILESET, RTYPE FROM SYSTEM.TABLE'
    + ' WHERE TYPE = 0 AND OWNER <> ''SYSTEM'';'
```

The PREPARE command ⑮ creates a temporary section named SelectCmd1 for the dynamic query from CmdLine.

Then the program initializes the two SQLDA fields ⑯ needed by the DESCRIBE command ⑰. Because the number of columns in the query result is known to be four at programming time, SqlN is set to 4. Four of the format array records will be needed, one per select list item.

The program then declares and opens a cursor named TableList for the dynamic query ⟨18⟩. Before using the cursor to retrieve rows, the program initializes the SQLDA ⟨19⟩ as follows:

- The SqlBufLen field is set to the size of the data buffer. In this program, the data buffer for the first query is a packed array of records named TableList ⟨4⟩. Note that each record in the array consists of four elements, one for each item in the select list. The elements are declared with types compatible with those in the corresponding SYSTEM.TABLE columns.

- The SqlRowBuf field is set to the address of the data buffer.

- The SqlNRow field is set to 300, defined in the constant MaxNbrTables ⟨1⟩. This number is the maximum number of rows ALLBASE/SQL will return from the active set when the FETCH command is executed.

After initializing the required fields in the SQLDA, the program executes the FETCH command ⟨20⟩. Because the FETCH command is executed only once, this program can re-create table definitions for a maximum of 300 tables.

After the FETCH command is executed, the value in SQLCA.SQLERRD[3] is saved in variable NumOfTables ⟨21⟩. This value indicates the number of rows ALLBASE/SQL returned to the data buffer. NumOfTables is used later as the final value of a counter ⟨23⟩ to control the number of times the second dynamic query is executed; the second query must be executed once for each table qualifying for the first query.

After terminating the transaction that executes the first query ⟨22⟩, the program uses the STRMOVE procedure ⟨24⟩ to move CHAR values to string variables so that other Pascal string procedures can be used when formatting the CREATE TABLE commands and writing them to the output file.

The second query ⟨26⟩ retrieves information about each column in each table qualifying for the first query. This query contains a WHERE clause that identifies an owner and table name:

```
CmdLine := 'SELECT COLNAME, LENGTH, TYPECODE, NULLS, PRECISION,'
        + ' SCALE FROM SYSTEM.COLUMN WHERE OWNER = '''
        + ' OwnerName + ''' AND TABLENAME = ''' + TableName + ''';';
```

These names are obtained from the Owner and Table values in the TableList array ⟨4⟩ after trailing blanks are trimmed by using the STRRTRIM function ⟨25⟩. Note that trailing blanks are also trimmed off the current TableList.FileSet value. Trailing blanks are removed from these three values so excess blanks do not appear when the values are written to the file containing the table definition.

After each version of the second query is dynamically preprocessed ⟨27⟩, the program initializes two SQLDA fields ⟨28⟩ before executing the DESCRIBE command ⟨29⟩. Then a cursor named ColumnList is declared and opened ⟨30⟩ to operate on the active set. Before fetching rows, the program initializes ⟨31⟩ the necessary SQLDA values:

- The SqlBufLen field is set to the size of the data buffer. The data buffer for the second query is a packed array of records named ColumnList ⟨5⟩.

- The SqlRowBuf field is set to the address of the data buffer.

- The SqlNRow field is set to 255, defined in the constant MaxNbrColumns ⟨2⟩. This number is the maximum number of rows ALLBASE/SQL will return from the active set when the FETCH command is executed.

The FETCH command (32) is executed only once for each table that qualified for the first query, since no more than 255 rows would *ever* qualify for the query. The maximum number of columns any table can have is 255.

After the active set has been fetched into data buffer ColumnList, a CREATE TABLE command for the table is written to the schema file (33):

```
CREATE LockMode TABLE OwnerName.TableName,
  (ColumnList.ColName[1]    TypeInfo    NullInfo,
   ColumnList.ColName[2]    TypeInfo    NullInfo,
   .
   .
   .
   ColumnList.ColName[j]    TypeInfo    NullInfo) IN TableList.FileSet[i];
```

Most of the information needed to reconstruct the CREATE TABLE commands is written directly from program variables. In three cases, however, data returned from the system views must be translated:

- LockMode is generated in a CASE statement (34) based on the value ALLBASE/SQL put in TableList.LockMode. The SYSTEM.TABLE view stores the automatic locking mode for tables as an integer from 1 through 3. The CASE statement equates these codes with the expressions that must appear in the CREATE TABLE command.

- TypeInfo is generated in a CASE statement (35) based on the value ALLBASE/SQL put in ColumnList.TypeCode. The SYSTEM.COLUMN view stores the data type of each column as an integer from 0 through 5. The CASE statement equates these codes with the expressions that must appear in the CREATE TABLE command.

- NullInfo is generated from the null indicator ALLBASE/SQL returned to ColumnList.Nulls (36). A value of 0 indicates the column cannot contain null values, and the program inserts NOT NULL into the table definition.

After a CREATE TABLE command has been written for each qualifying table, a COMMIT WORK command is executed (37) to release locks on SYSTEM.COLUMN before the PREPARE command is re-executed and before the DBE session terminates (38).
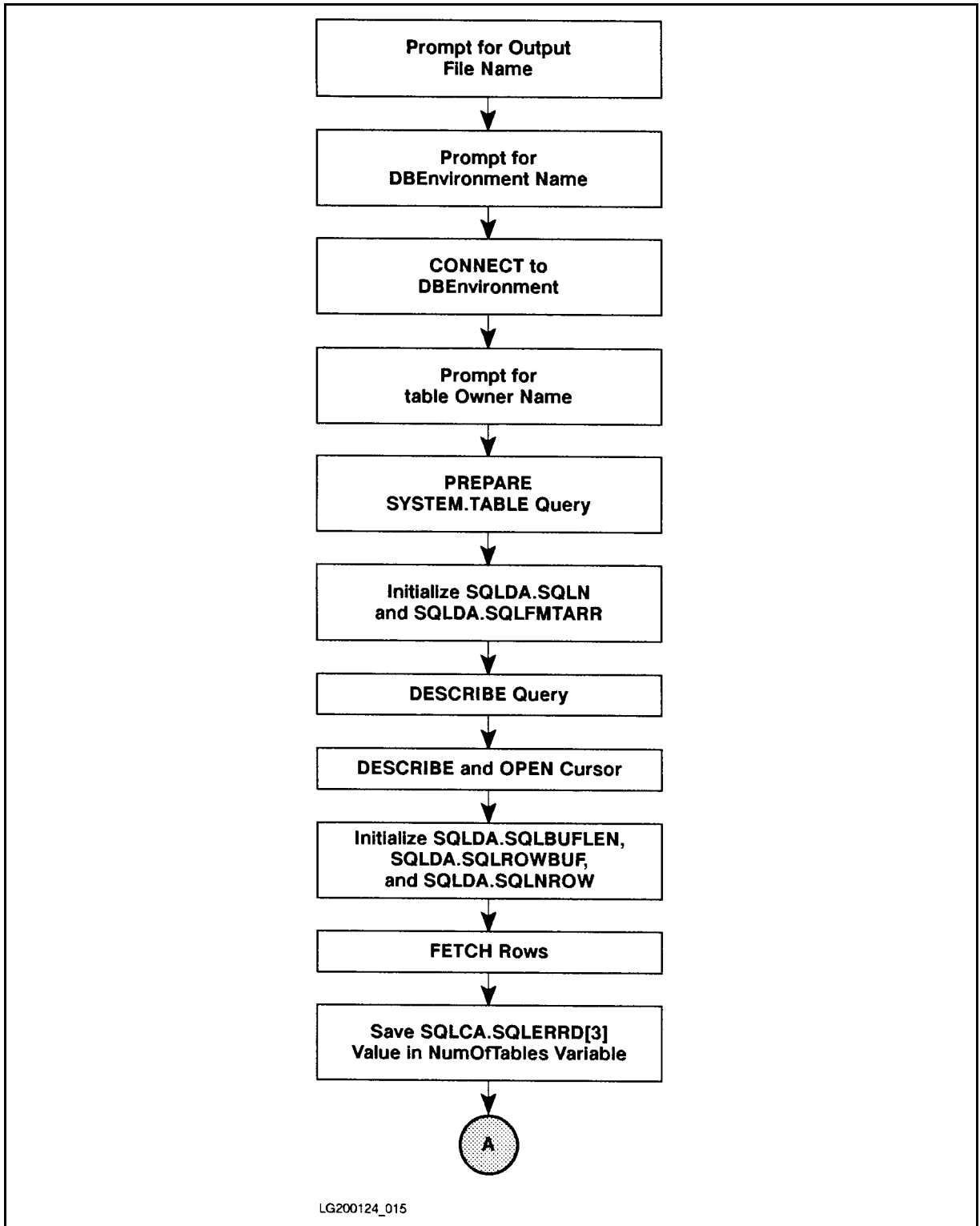
Figure 10-10. Flow Chart of Program pasex10b
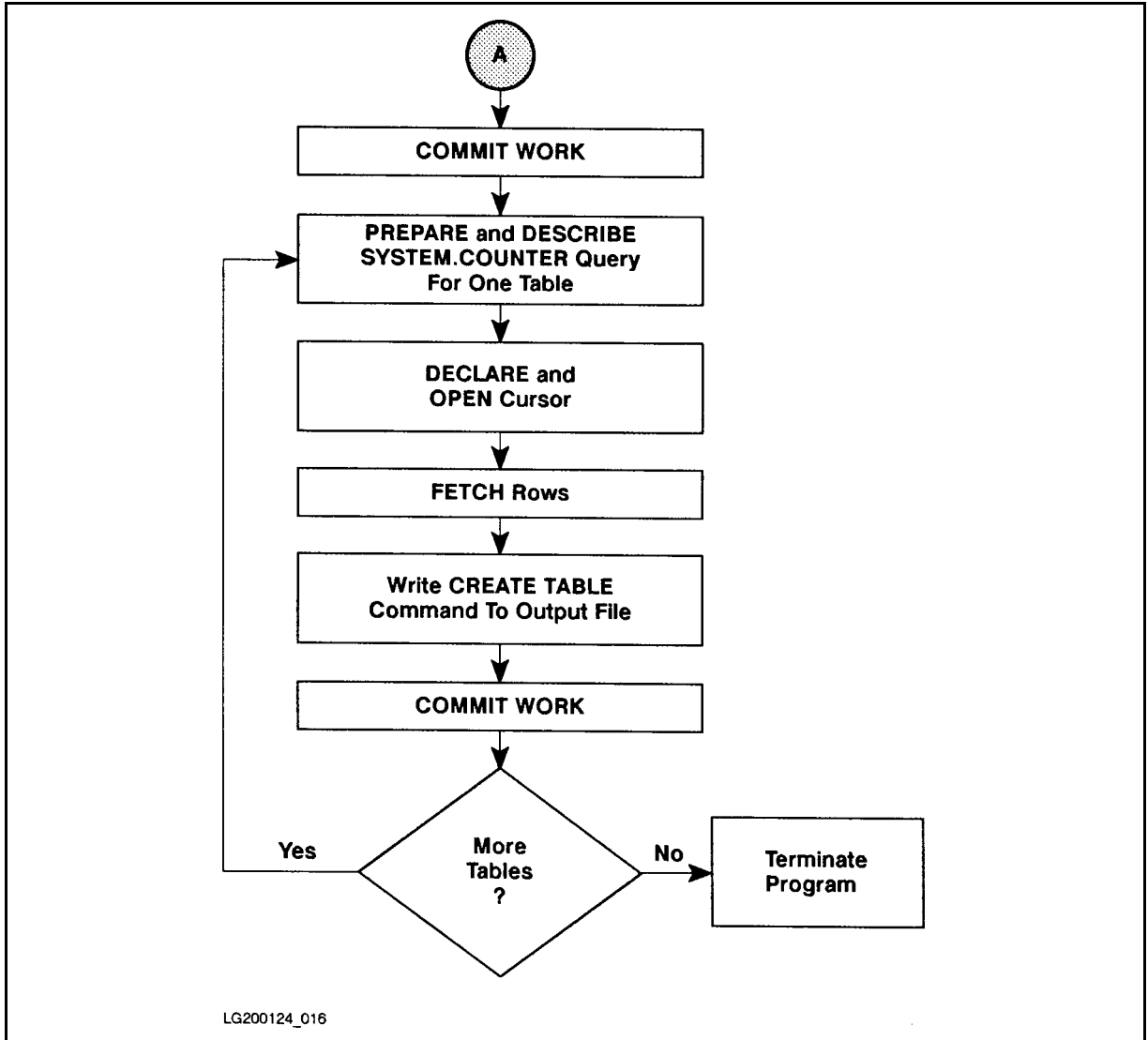
Figure 10-10. Flow Chart of Program pasex10b (page 2 of 2)

In the runtime dialog shown in Figure 10-11, the name of the DBEnvironment must be entered with upper and lower case as shown. The name of the schema file and the name of the owner can be entered with either upper or lower case.

```
ALLBASE/SQL SCHEMA Generator for Tables             X.00.00


Enter name of schema file to be generated > schm1
Enter name of DBEnvironment > ../sampledb/PartsDBE
Enter owner name or RETURN for all owners > PURCHDB


Generating SQL command to CREATE TABLE PURCHDB.INVENTORY
Generating SQL command to CREATE TABLE PURCHDB.ORDERITEMS
Generating SQL command to CREATE TABLE PURCHDB.ORDERS
Generating SQL command to CREATE TABLE PURCHDB.PARTS
Generating SQL command to CREATE TABLE PURCHDB.REPORTS
Generating SQL command to CREATE TABLE PURCHDB.SUPPLYPRICE
Generating SQL command to CREATE TABLE PURCHDB.VENDORS
$ more schm1

CREATE PUBLIC TABLE PURCHDB.INVENTORY
   (PARTNUMBER             CHAR( 16)     NOT NULL,
    BINNUMBER              SMALLINT      NOT NULL,
    QTYONHAND              SMALLINT,
    LASTCOUNTDATE          CHAR(  8),
    COUNTCYCLE             SMALLINT,
    ADJUSTMENTQTY          SMALLINT,
    REORDERQTY             SMALLINT,
    REORDERPOINT           SMALLINT) IN WAREHFS;

CREATE PUBLIC TABLE PURCHDB.ORDERITEMS
   (ORDERNUMBER            INTEGER       NOT NULL,
    ITEMNUMBER             INTEGER       NOT NULL,
    VENDPARTNUMBER         CHAR( 16),
    PURCHASEPRICE          DECIMAL(10, 2) NOT NULL,
    ORDERQTY               SMALLINT,
    ITEMDUEDATE            CHAR(  8),
    RECEIVEDQTY            SMALLINT) IN ORDERFS;

CREATE PUBLIC TABLE PURCHDB.ORDERS
   (ORDERNUMBER            INTEGER       NOT NULL,
    VENDORNUMBER           INTEGER,
    ORDERDATE              CHAR(  8)) IN ORDERFS;

CREATE PUBLIC TABLE PURCHDB.PARTS
   (PARTNUMBER             CHAR( 16)     NOT NULL,
    PARTNAME               CHAR( 30),
    SALESPRICE             DECIMAL(10, 2)) IN WAREHFS;

CREATE PUBLIC TABLE PURCHDB.REPORTS
   (REPORTNAME             CHAR( 20)     NOT NULL,
    REPORTOWNER            CHAR( 20)     NOT NULL,
 .
 .
 .
```

Figure 10-11. Runtime Dialog of Program pasex10b

```
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* This program generates an ISQL Command File that contains CREATE  *)
(* TABLE commands for a particular DBEnvironment.  This program must *)
(* be preprocessed; however, it does not need to be INSTALLed.       *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

Program pasex10b(input, output);

label
  9999;

type
  UXFile         = packed array [1..100] of char;
  FilePtr        = UXFile;

const
  MaxNbrTables   = 300;
  MaxNbrColumns  = 255;
  OK             =   0;
  mode           = 0700;

var

  EXEC SQL BEGIN DECLARE SECTION;
  MsgLine    : Packed Array [ 1..132] of char;
  CmdLine    : String[200];
  EXEC SQL END DECLARE SECTION;

  EXEC SQL INCLUDE SQLCA;
  EXEC SQL INCLUDE SQLDA;

  SchemaFile   : Text;
  FileName     : FilePntr;
  OwnerName    : String[20];
  TableName    : String[20];
  DBEFileSet   : String[20];
  ColumnName   : String[20];
  DBEName      : String[50];
  OneLine      : String[80];
  i,j          : SmallInt;
  Pos          : Integer;
  NumOfTables  : SmallInt;
```

1

2

Figure 10-12. Program pasex10b: Dynamic Queries of Known Format

```
$PAGE $
  SqlFmts     : Array[1..6] of SQLFormat_Type;                    ( 3 )

  TableList   : Packed Array[1..MaxNbrTables] of Record           ( 4 )
                  Owner     : Packed Array[1..20] of char;
                  Table     : Packed Array[1..20] of char;
                  FileSet   : Packed Array[1..20] of char;
                  LockMode  : SmallInt;
                end;

  ColumnList  : Packed Array[1..MaxNbrColumns] of Record          ( 5 )
                  ColName   : Packed Array[1..20] of char;
                  Length    : Integer;
                  TypeCode  : SmallInt;
                  Nulls     : SmallInt;
                  Precision : SmallInt;
                  Scale     : SmallInt;
                end;


$PAGE $

begin


writeln;
writeln('ALLBASE/SQL SCHEMA Generator for Tables          X.00.00');
writeln;

prompt('Enter name of schema file to be generated > ');          ( 6 )
readln(FileName);


rewrite(SchemaFile, FileName);
```

Figure 10-12. Program pasex10b: Dynamic Queries of Known Format (page 2 of 7)

```
prompt('Enter name of DBEnvironment > ');                          8
readln(DBEName);


CmdLine := 'CONNECT TO''' + DBEName + ''';';                       9


EXEC SQL EXECUTE IMMEDIATE : CmdLine;                             10
if SQLCA.SQLCODE <> OK then
  begin
    writeln('Could not CONNECT to DBEnvironment.');
    EXEC SQL SQLEXPLAIN : SQLMessage;
    writeln(SQLMessage);
    goto 9999;
  end;
$PAGE $


prompt('Enter owner name or RETURN for all owners > ');           11
readln(OwnerName);
OwnerName := StrLTrim(StrRTrim(OwnerName));

  (* Upshift OwnerName *)
  for i := 1 to StrLen(OwnerName) do                              12
    if OwnerName[i] in ['a'..'z'] then
      OwnerName[i] := chr(ord(OwnerName[i]) - ord('a') + ord('A'));


writeln;
if OwnerName = '' then
 CmdLine:= 'SELECT OWNER,NAME,DBEFILESET,RTYPE FROM SYSTEM.TABLE'  13
      + ' WHERE TYPE = O AND OWNER <> ''SYSTEM'';'
else
CmdLine:= 'SELECT OWNER,NAME,DBEFILESET,RTYPE FROM SYSTEM.TABLE'   14
      + ' WHERE TYPE = O AND OWNER = ''' + OwnerName + ''';';


EXEC SQL PREPARE SelectCmd1 FROM :CmdLine;                        15
if SQLCA.SQLCODE <> OK then
  begin
    writeln('Problem PREPARING the SELECT command.');
    EXEC SQL SQLEXPLAIN :SQLMessage;
    writeln(SQLMessage);
    goto 9999;
  end;
```

Figure 10-12. Program pasex10b: Dynamic Queries of Known Format (page 3 of 7)

```
with SQLDA do    (* set up SQLDA fields *)                        16
  begin
    Sqln       := 4;  (* number of columns expected *)
    SqlFmtArr := waddress(SQLFmts);
  end;


EXEC SQL DESCRIBE SelectCmd1 INTO SQLDA;                          17


$PAGE $
EXEC SQL DECLARE TableList CURSOR for SelectCmd1;                 18
EXEC SQL OPEN TableList;
if SQLCA.SQLCODE <> OK then
  begin
    writeln('Problem opening TableList cursor.');
    EXEC SQL SQLEXPLAIN :SQLMessage;
    writeln(SQLMessage);
    goto 9999;
  end;


with SQLDA do                                                    19
  begin
    SqlBufLen := SizeOf(TableList);
    SqlRowBuf := Waddress(TableList);
    SqlNRow   := MaxNbrTables;
  end;


  (* Get Table List from SYSTEM.TABLE *)
EXEC SQL FETCH TableList USING DESCRIPTOR SQLDA;                  20
if SQLCA.SQLCODE = 100 then
  begin
    writeln('No tables qualified.');
    goto 9999;
  end
else if SQLCA.SQLCODE <> OK then
  begin
    writeln('Problem encountered when reading SYSTEM.TABLE');
    EXEC SQL SQLEXPLAIN :SQLMessage;
    writeln(SQLMessage);
    goto 9999;
  end;
```

Figure 10-12. Program pasex10b: Dynamic Queries of Known Format (page 4 of 7)

```
  NumOfTables := SQLCA.SQLERRD[3];                           (21)

  EXEC SQL COMMIT WORK;                                       (22)
  $PAGE $
  for i := 1 to NumOfTables do                               (23)
    with TableList[i] do
      begin

      OwnerName := '';
      StrMove(20, Owner, 1, OwnerName,1);                    (24)
      OwnerName  := StrRTrim(OwnerName);
      TableName := '';
      StrMove(20, Table, 1, TableName, 1);
      TableName  := StrRTrim(TableName);
      DBEFileSet := '';
      StrMove(20, FileSet, 1, DBEFileSet, 1);
      DBEFileSet := StrRTrim(DBEFileSet);                    (25)

      write('Generating SQL command to CREATE TABLE ');
      writeln(OwnerName, '.', TableName);

    CmdLine:='SELECT COLNAME, LENGTH, TYPECODE, NULLS, PRECISION,' (26)
             + ' SCALE FROM SYSTEM.COLUMN WHERE OWNER = '''
             + OWNERNAME + ''' AND TABLENAME = ''' + TableName + ''';';

      EXEC SQL PREPARE SelectCmd2 FROM :CmdLine;             (27)
      if SQLCA.SQLCODE <> OK then
        begin
          writeln('Problem PREPARING the SELECT #2 command.');
          EXEC SQL SQLEXPLAIN :SQLMessage;
          writeln(SQLMessage);
          goto 9999;
        end;
      with SQLDA do   (* set up SQLDA fields *)              (28)
        begin
          Sqln       := 6;  (* number of columns expected *)
          SqlFmtArr := waddress(SQLFmts);
        end;

      EXEC SQL DESCRIBE SelectCmd2 INTO SQLDA;               (29)
```

Figure 10-12. Program pasex10b: Dynamic Queries of Known Format (page 5 of 7)

```
$PAGE $
    EXEC SQL DECLARE ColumnList CURSOR for SelectCmd2;              (30)
    EXEC SQL OPEN ColumnList;
    if SQLCA.SQLCODE <> OK then

      begin
        writeln('Problem opening cursor #2.');
        EXEC SQL SQLEXPLAIN :SQLMessage;
        writeln(SQLMessage);
        goto 9999;
      end;

    with SQLDA do                                                  (31)
      begin
        SqlBufLen := SizeOf(ColumnList);
        SqlRowBuf := Waddress(ColumnList);
        SqlNRow   := MaxNbrColumns;
      end;

  (* Get Column List from SYSTEM.COLUMN *)
    EXEC SQL FETCH ColumnList USING DESCRIPTOR SQLDA;              (32)
    if SQLCA.SQLCODE <> OK then
      begin
        writeln('Problem encountered when reading SYSTEM.COLUMN');
        EXEC SQL SQLEXPLAIN :SQLMessage;
        writeln(SQLMessage);
        goto 9999;
      end;
$PAGE $
    writeln(SchemaFile);
    OneLine := 'CREATE ';                                         (33)
    Pos := 8;
    case LockMode of                                              (34)
        1 : StrWrite(OneLine, Pos, Pos, 'PUBLICREAD ');
        2 : StrWrite(OneLine, Pos, Pos, 'PRIVATE ');
        3 : StrWrite(OneLine, Pos, Pos, 'PUBLIC ');
     end;  (* end case  *)

    StrWrite(OneLine, Pos, Pos, 'TABLE ', OwnerName, '.', TableName);
    writeln(SchemaFile, OneLine);
    OneLine := ' (';
    Pos := 4;
```

Figure 10-12. Program pasex10b: Dynamic Queries of Known Format (page 6 of 7)

```
      for j := 1 to SQLCA.SQLERRD[3] do
        with ColumnList[j] do
          begin
          ColumnName := '';
          StrMove(20, ColName, 1, ColumnName, 1);
          StrWrite(OneLine, Pos, Pos, ColumnName, '  ');
          case TypeCode of                                            ┌────┐
            0 : if Length = 4 then                                    │ 35 │
                  StrWrite(OneLine, Pos, Pos,                         └────┘
                            'INTEGER        ')
              else
                StrWrite(OneLine, Pos, Pos,
                          'SMALLINT       ');
            2,8 : StrWrite(OneLine, Pos, Pos,
                      'CHAR(', Length:4, ')       ');
            3,9 : StrWrite(OneLine, Pos, Pos,
                      'VARCHAR(', Length:4, ')  ');
            4 : StrWrite(OneLine, Pos, Pos,
                      'FLOAT          ');
            5 : StrWrite(OneLine, Pos, Pos,
                      'DECIMAL(', Precision:2, ',', Scale:2, ') ');
          otherwise StrWrite(OneLine, Pos, Pos, '****');
          end;  (* case *)
          if Nulls = 0 then                                           ┌────┐
            OneLine := OneLine + 'NOT NULL'                           │ 36 │
          else                                                        └────┘
            OneLine := StrRTrim(OneLine);
          if j <> SQLCA.SQLERRD[3] then
            OneLine := OneLine + ','
          else
            OneLine := OneLine + ') IN ' + DBEFileSet + ';';
          writeln(SchemaFile, OneLine);
          OneLine := '   ';
          Pos := 4;
          end;    (* for j := 1 to SQLCA.SQLERRD[3] *)
    EXEC SQL COMMIT WORK;                                             ┌────┐
    end;          (* for i := 1 to NumOfTables  *)                   │ 37 │
                                                                     └────┘

9999:
EXEC SQL COMMIT WORK RELEASE;                                         ┌────┐
close (SchemaFile,'SAVE');                                           │ 38 │
end.                                                                 └────┘
```

Figure 10-12. Program pasex10b: Dynamic Queries of Known Format (page 7 of 7)

11

Programming With Constraints

This chapter explains the use of statement level integrity versus row level integrity. Also, methods of implementing schema level unique and referential integrity constraints in your database are highlighted.

Integrity constraints allow you to have ALLBASE/SQL verify data integrity at the schema level. Thus you can avoid coding complex verification routines in application programs and avoid the increased execution time of additional queries. Your coding tasks are simplified, and performance is improved.

The following sections are presented in the chapter:

- Comparing Statement Level and Row Level Integrity.
- Using Unique and Referential Integrity Constraints.
- Designing an Application Using Statement Level Integrity Checks.

Comparing Statement Level and Row Level Integrity

The following discussion applies to the use of BULK INSERT, Type 2 INSERT, UPDATE, and DELETE commands.

In ALLBASE/SQL release E.1, enforcement of defined constraints is performed at statement level rather than at the row level of previous releases. This is called **statement level integrity**. Even though a constraint may be violated on a particular row, the check for that constraint is not made until the statement has completed processing. At that time, if there are one or more constraint errors, an error message is issued and the entire statement is rolled back with no rows being processed. You do not need to detect constraint errors yourself and code your program to respond to partially processed tables.

When a statement is rolled back, the appropriate sqlerrd field will be 0, reflecting that no rows were processed. If a constraint error is the cause of the rollback, this field will not be greater than zero indicating a partially processed table. Thus, applications written for ALLBASE/SQL may need to check for a different value in the sqlerrd field.

For information on status checking, see the chapter, "Runtime Status Checking and the SQLCA." For information on deferring constraint error checking to the transaction level and other error checking enhancements related to releases after E.1, see the *ALLBASE/SQL Release F.0 Application Programming Bulletin for HP-UX*.

Using Unique and Referential Integrity Constraints

Any database containing tables with interdependent data is a good candidate for the use of integrity constraints. You can profit from their use whether your data is volatile or stable. For instance, your database might contain a table of employee and department data that is constantly changing, or it could contain a table of part number data that rarely changes even though it is frequently accessed. (Note that integrity constraints cannot be assigned to LONG columns. LONG columns are described in the chapter, "Programming with LONG Columns.")

To implement unique and referential constraints, use the CREATE TABLE command and optionally the GRANT REFERENCES command in your schema file. The following table lists the commands you might use in dealing with integrity constraints.

Table 11-1. Commands Used with Integrity Constraints

DDL Operations	DCL Operations	DML Operations
CREATE TABLE	GRANT REFERENCES	[BULK] INSERT
DROP TABLE	GRANT DBA	UPDATE [WHERE CURRENT]
REMOVE FROM GROUP	REVOKE REFERENCES	DELETE [WHERE CURRENT]
DROP GROUP	REVOKE DBA	

The concepts and syntax of integrity constraints are fully discussed in the *ALLBASE/SQL Reference Manual* , and database administration considerations are found in the *ALLBASE/SQL Database Administration Guide* . This chapter contains techniques to use when coding applications that manipulate data upon which integrity constraints have been defined.

When executing the [BULK] INSERT, UPDATE [WHERE CURRENT], or DELETE [WHERE CURRENT] commands, ALLBASE/SQL considers applicable integrity constraints depending on what the overall effect of a statement would be once it completes execution. The syntax for UNIQUE or PRIMARY KEY requires unique constraint enforcement. The syntax for REFERENCES requires referential constraint enforcement on the referencing and referenced tables involved. For example, consider the following table showing what tests must be passed for a DML command to successfully complete. Refer to the *ALLBASE/SQL Reference Manual* for more information on enforcing constraints.

Table 11-2. Constraint Test Matrix

DML Operations	UNIQUE or PRIMARY KEY	Referenced Table	Referencing Table
[BULK] INSERT or Type 2 INSERT	Must be unique in the table.		Must match a unique key in the referenced table.
UPDATE [WHERE CURRENT]	Must be unique in the table.	No foreign key can reference the unique key being updated.	Must match a unique key in the referenced table.
DELETE [WHERE CURRENT]		No foreign key can reference the unique key being deleted.	

Designing an Application Using Statement Level Integrity Checks

This section contains examples based on the recreation database, RecDB, which is supplied as part of the ALLBASE/SQL software package. The schema files used to create the database are found in appendix C of the *ALLBASE/SQL Reference Manual* .

The recreation database is made up of three tables (Clubs, Members, and Events). Two primary key constraints and two referential constraints were specified when the tables were created to secure the data integrity of these tables.

Figure 11-1 illustrates these constraint relationships by showing the name of each constraint and its referencing or referenced columns. Referencing columns are shaded. Referenced columns are clear white.
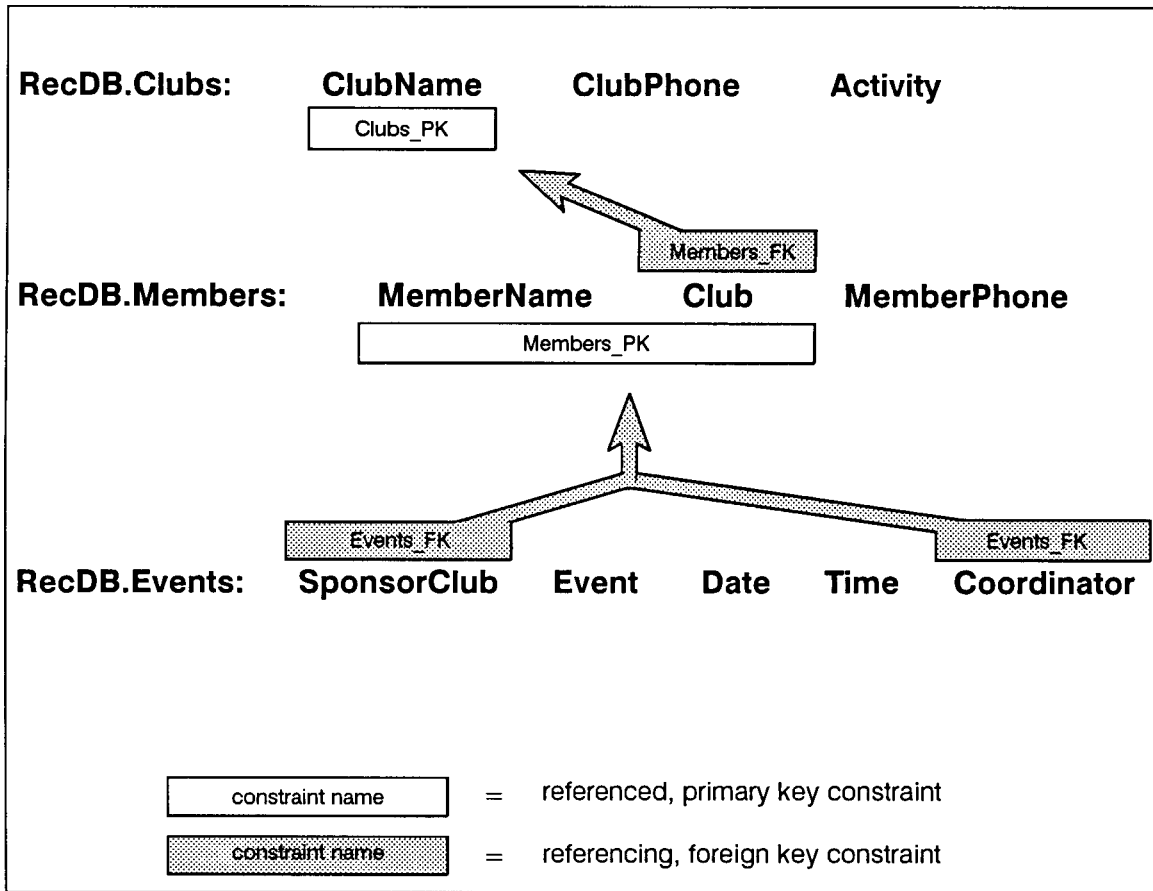
LG200145_007

Figure 11-1. Constraints Enforced on the Recreation Database

Suppose you designed an application program providing a user interface to the recreation database. The interface gives choices for inserting, updating, and deleting data in any of the three tables. Your application is user friendly and guides the user with error messages when their request is denied because it would violate data integrity. The main interface menu might look like this:

```
          Main Menu for Recreation Database Maintenance
          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


   1. INSERT a Club         4. INSERT a Member         7. INSERT an Event
   2. UPDATE a Club         5. UPDATE Member Info.     8. UPDATE Event Info.
   3. DELETE a Club         6. DELETE a Member         9. DELETE an Event
```

When users make a selection (by number or by tabbing to a field), a screen displaying all the appropriate information allows them to insert, update, or delete.

The next sections provide generic examples of how you can code such an application. The error checking in these examples deals with constraint enforcement errors only. (For complete explanation of these errors, see the *ALLBASE/SQL Message Manual* .) Your error checking routine should also include a method of handling multiple errors per command and errors not related to constraint enforcement. (For more information on error coding techniques, see the chapter, "Runtime Status Checking and the SQLCA.")

11-4 Programming With Constraints

Insert a Member in the Recreation Database

The user chooses to insert a new member in the database. For this activity to complete, the foreign key (Club) which is being inserted into the Members table must exist in the primary key (ClubName) of the Clubs table.

Execute routines to display and prompt for information needed in the Members table.

Place user entered information in appropriate host variables.

```
INSERT INTO RecDB.Members
    VALUES (:MemberName,
            :Club,
            :MemberPhone :MemberPhoneInd)
```

Check the sqlcode field of the sqlca.

If sqlcode equals −2293, indicating no primary key match, display the error message and prompt the user to indicate whether or not to insert a new ClubName in the Clubs table, to reenter the Club for the new member, or to exit to the main menu. Execute the appropriate subroutine.

If sqlcode equals −2295, indicating that the user tried to insert a non-unique primary key, display the error message and prompt the user to enter a unique MemberName/Club combination or to exit to the main menu. Execute the appropriate subroutine.

Else, if sqlcode = 0, tell the user the member was inserted successfully, and prompt for another new member or a return to the main menu display.

Update an Event in the Recreation Database

The user now wants to update information in the Events table. For this activity to complete, the SponsorClub and Coordinator being updated in the Events table must exist in the primary key composed of MemberName and Club in the Members table.

Execute subroutines to display and prompt for information needed in the Events table.

Place user entered information in appropriate host variables.

```
UPDATE RecDB.Events
    SET SponsorClub = :SponsorClub :SponsorClubInd,
        Event = :Event :EventInd,
        Date = :Date DateInd,
        Time = :Time TimeInd,
        Coordinator = :Coordinator CoordinatorInd
    WHERE Event = :Event
```

Check the sqlcode field of the sqlca.

If sqlcode equals −2293, indicating no primary key match, display the error message and prompt the user to indicate whether or not to insert a new MemberName/Club primary key in the Members table, to reenter update information for the Events table, or to exit to the main menu. Execute the appropriate subroutine.

Else, if sqlcode = 0, tell the user the event was updated successfully, and prompt for another event or a return to the main menu display.

Delete a Club in the Recreation Database

The user chooses to delete a club. For this activity to complete, no foreign key must reference the primary key (ClubName) that is being deleted.

Execute subroutines to display and prompt for a ClubName in the Clubs table.

Place user entered information in appropriate host variables.

```
DELETE FROM RecDB.Clubs
      WHERE ClubName = :ClubName
```

Check the sqlcode field of the sqlca.

If sqlcode equals −2293, indicating that referencing data exists for ClubName,
display the error message and prompt the user to indicate whether or not
to delete the Members table row or rows that reference the ClubName,
to reenter the ClubName to be deleted, or to exit to the main menu.
Execute the appropriate subroutine.

(If you execute the subroutine to delete those rows in the Members table
which reference the Clubs table, be sure to test sqlcode.
Depending on the result, you can prompt the user to delete referencing
Events table rows, to reenter the Members table information, or to exit
to the main menu. Execute the appropriate subroutine.)

Else, if sqlcode = 0, tell the user the club was deleted successfully,
and prompt for another club or a return to the main menu display.

Delete an Event in the Recreation Database

The user chooses to delete an event. Because no primary key or unique constraints are defined in the Events table, no constraint enforcement is necessary.

Execute subroutines to display and prompt for an Event in the Events table.

Place user entered information in appropriate host variables.

```
DELETE FROM RecDB.Clubs
      WHERE Event = :Event
```

Check the sqlcode field of the sqlca.

If sqlcode = 0, tell the user the event was deleted successfully, and
prompt for another event or a return to the main menu display.

12

Programming with LONG Columns

LONG columns in ALLBASE/SQL enable you to store a very large amount of binary data in your database, referencing that data via a table column name. You might use LONG columns to store text files, software application code, voice data, graphics data, facsimile data, or test vectors. You can easily SELECT or FETCH this data, and you have the advantages of ALLBASE/SQL's recoverability, concurrency control, locking strategies, and indexes on related columns.

You can use LONG columns in an application program to be preprocessed or with ISQL. This discussion focuses on application programming concerns. As you will see, great flexibility is provided so that you can custom design your application.

The chapter highlights methods of implementing LONG columns in your database as follows:

- General Concepts.
- Restrictions.
- Defining LONG Columns with the CREATE TABLE or ALTER TABLE command.
- Defining Input and Output with the LONG Column I/O String.
- Putting Data into a LONG Column with INSERT.
- Retrieving LONG Column Data with SELECT, FETCH, or REFETCH.
- Changing a LONG Column with UPDATE [WHERE CURRENT].
- Using the LONG Column Descriptor.
- Removing LONG Column Data with DELETE or DELETE WHERE CURRENT.
- Coding Considerations.

For every DDL and DML command that can be used with LONG columns, examples are included with discussion of related considerations. These examples pertain to the same logical table (PartsTable) and set of columns. In contrast to other examples in this chapter, PartsTable is a hypothetical table created and altered in this chapter. Refer to the *ALLBASE/SQL Reference Manual* which contains complete syntax specifications for using long columns.

Table 12-1. Commands You Can Use with LONG Columns

DDL Operations	DML Operations
ALTER TABLE	INSERT
CREATE TABLE	UPDATE [WHERE CURRENT]
	SELECT
	FETCH
	REFETCH
	DELETE [WHERE CURRENT]

General Concepts

ALLBASE/SQL stores LONG column data in a database for later retrieval. LONG column data is not manipulated by ALLBASE/SQL when it is modified or retrieved. Any formatting, viewing, or other processing must be accomplished by means of your program. For example, you might use a graphics application to create an intricate graphic display (or set of graphic displays). You could then write a program in which you embed ALLBASE/SQL commands to store each graphics file in your database along with related data in a given row. Your graphics application could be called from another program, this time to select a row and display the graphic. The graphic could be displayed on the upper portion of a screen, with related data from the same row displayed on the lower portion of a screen. The related data in standard columns or LONG columns could be a graphics explanation or an entire chapter.

LONG column data can occupy a practically unlimited amount of space in the database, the maximum number of bytes being $2^{31}-1$ (or 2,147,483,647) per LONG column per row. Standard column data is restricted to 3996 bytes maximum.

The LONG specification is used with a given ALLBASE/SQL data type when you create the LONG column. Currently, LONG BINARY and LONG VARBINARY are available. Refer to the chapter on "Host Variables" for the details of BINARY and VARBINARY data types.

The concept of how LONG column data is stored in a row and retrieved differs from that of standard columns. Although LONG column data is associated with a particular row, it can be stored separately from the row. Thus you can specify a DBEFileSet in which to store data for a LONG column.

During an INSERT or UPDATE operation, you specify a **LONG column I/O string** to indicate where LONG column input data is located and where that data is to be placed when it is later selected or fetched. You indicate either an operating system file or shared memory.

A **LONG column descriptor** (rather than the data itself) is selected or fetched into a host variable. Figure 12-1 and Figure 12-2 illustrate these concepts.
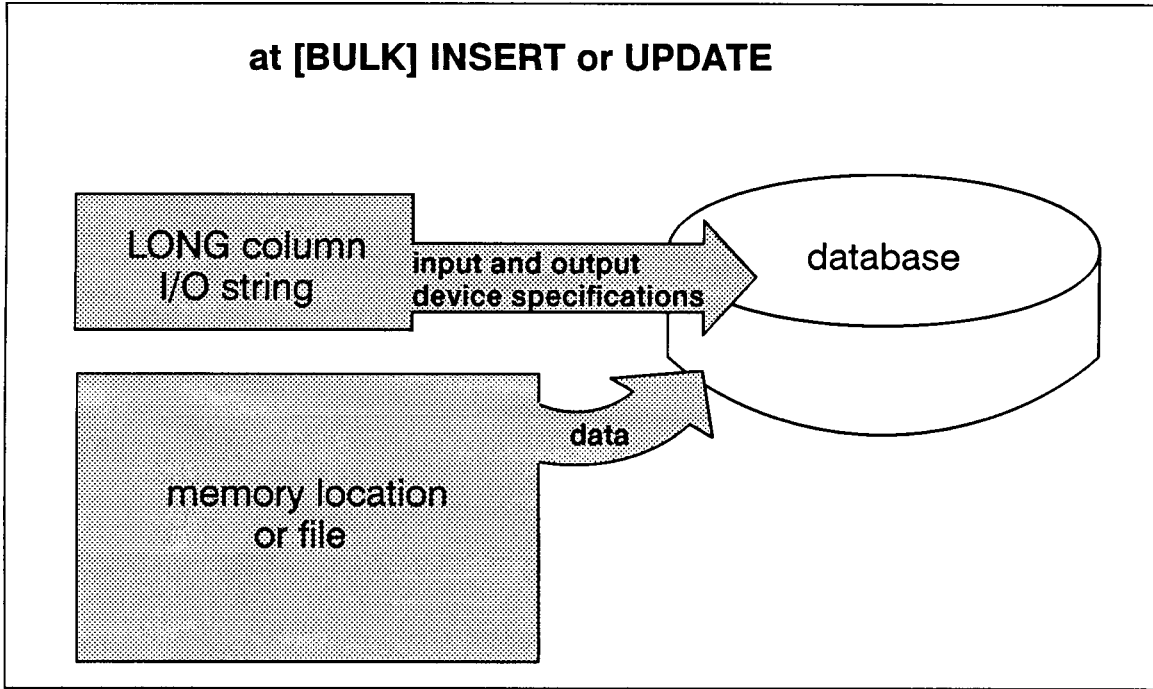
Figure 12-1. Flow of LONG Column Data and Related Information to the Database
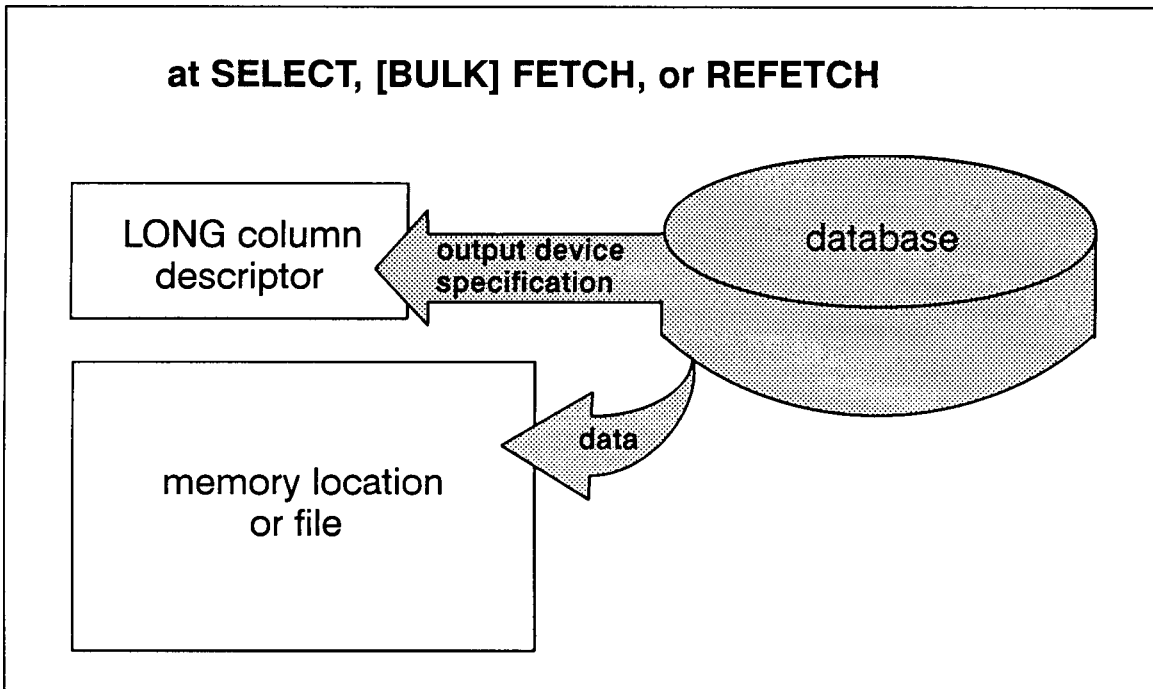
Figure 12-2. Flow of LONG Column Data and Related Information from the Database

Restrictions

A LONG column can be referenced in a select list and/or a host variable declaration. Some restrictions do apply to LONG columns. However, related standard columns are not affected by these restrictions.

LONG columns cannot be used as follows:

- In a WHERE clause.
- In a Type 2 INSERT command.
- Remotely through ALLBASE/NET.
- As hash or B-tree index key columns.
- In a GROUP BY, ORDER BY, DISTINCT, or UNION clause.
- In an expression.
- In a subquery.
- In aggregate functions (AVG, SUM, MIN, MAX).
- As columns to which integrity constraints are assigned.
- With the DEFAULT option of the CREATE or ALTER TABLE commands.

Defining LONG Columns with a CREATE TABLE or ALTER TABLE Command

Following is the portion of the CREATE TABLE or ALTER TABLE command syntax for specifying a LONG column *column definition*. A maximum of 40 such LONG columns may be defined for a single table.

$$(ColumnName \; \texttt{LONG} \; \left\{ \begin{matrix} \texttt{BINARY} \\ \texttt{VARBINARY} \end{matrix} \right\} (ByteSize) \; \big[\texttt{IN} \; DBEFileSet \big] \; \big[\texttt{NOT NULL} \big]) \; \big[\, , \, \dots \, \big]$$

When you create or add a LONG column to a table you have the option of specifying the DBEFileSet in which it is to be stored. Because LONG column data may take up a large chunk of a given DBEFile's data pages, placing LONG column data in a separate DBEFileSet is strongly advantageous from the standpoint of storage as well as performance.

If the IN DBEFileSetName clause is not specified for a LONG column, this column's data is by default stored in the same DBEFileSet as its related table.

Note It is recommended that you do *not* use the SYSTEM DBEFileSet in which to store your data, as this could severely impact database performance.

In the following example, LONG column data for PartPicture will be stored in PartPictureSet while data for columns PartName and PartNumber will be stored in PartsTableSet.

```
CREATE TABLE PartsTable (
          PartName CHAR(10),
          PartNumber INTEGER,
          PartPicture LONG VARBINARY(1000000) IN PartPictureSet)
       IN PartsTableSet
```

The next command specifies that data for new LONG column, PartModule, be stored in PartPictureSet.

```
ALTER TABLE PartsTable
        ADD PartModule LONG VARBINARY(70000) IN PartPictureSet
```

See the "BINARY Data" section of the "Host Variables" chapter for more information on using BINARY and VARBINARY data types in long columns.

Now that we have defined our table, let's see how to put data into it and to specify where data goes when it is retrieved.

Defining Input and Output with the LONG Column I/O String

Both the INSERT and the UPDATE commands allow you to define various input and output parameters for any LONG column. Parameters are specified with a **LONG column I/O string**. You'll need to understand this string in order to input, change, or retrieve LONG column data. This section offers an overview. See the *ALLBASE/SQL Reference Manual* for complete syntax.

Using the INSERT or UPDATE command, you pass the string to ALLBASE/SQL as either a host variable or a literal. Host variables are covered in detail in the "Host Variables" chapter.

Note   The input and output portions of the I/O string are not positional. In the following examples, < indicates input, and > indicates output. See the *ALLBASE/SQL Reference Manual* for a full description of I/O operations with LONG columns.

The input portion of the LONG column I/O string specifies the location of data that you want written to the database. It is also referred to as an **input device specification**. You can indicate a file name or a shared memory ID.

Use the output portion of the I/O string (**output device specification**) to indicate where you want LONG column data to be placed when you use the SELECT or FETCH command. You have the option of specifying a file name, part of a file name, or having ALLBASE/SQL specify a file name. You also can specify a shared memory location or have ALLBASE/SQL specify a memory location. Additional output parameters allow you to append to or overwrite an existing file. Information in the output device specification is stored in the database table and is available to you when a LONG column is selected or fetched (via a **LONG column descriptor**, discussed later in the section, "Using the LONG Column Descriptor").

It is important to note that files used for LONG column input and output are opened and closed by ALLBASE/SQL for its purposes. You need not open or close such files in your program unless you use them for additional purposes. ALLBASE/SQL does not control input or output device files once they are on the operating system. So, any operation on the file is valid, whether by your application or another application or user of the system. Such files are your responsibility, even before the transaction is complete.

The syntax for the INSERT and UPDATE commands is identical except that the input device is required for the INSERT command.

Putting Data into a LONG Column with a INSERT Command

As with any column, use the INSERT command to initially put data into a LONG column. At the time of the insert, all input devices must be on the system in the locations you have specified. Should your insert operation fail, nothing is inserted, a relevant error message is returned to the program, and the transaction continues. Depending on your application, you might want to write a verification routine that reads a portion of each specified input device to make certain valid data exists prior to using the INSERT command.

The next examples are based on the PartsTable created and altered in the previous section, "Defining LONG Columns with CREATE TABLE or ALTER TABLE." Additional examples of LONG column I/O string usage are found in the *ALLBASE/SQL Reference Manual* .

Insert Using Host Variables for LONG Column I/O Strings

When inserting a single row, use a version of the LONG Column I/O String for each LONG column following the VALUES clause, as below.

```
INSERT INTO PartsTable
     VALUES ('bracket',
             200,
             :PartPictureIO,
             :PartModuleIO)
```

An example of the values that might be stored in the host variables, :PartPictureIO and :Part ModuleIO, are shown in the last two fields of a hypothetical record below. In the above example, the values, bracket and 200, are coded as constants, rather than coming from the data file. Your data file might look like this (note that each item is limited to 80 characters per record to facilitate documentation):

```
bracket    200 0'<../tools/bracket >bracket'   0'<../module/mod88 > mod88'    0
hammer     011 0'<../tools/hammer >hammer'      0'<../module/mod11 > mod11'    0
file       022 0'<../tools/file >file'          0'<../module/mod22 > mod22'    0
saw        033 0'<../tools/saw > saw'           0'<../module/mod33 > mod33'    0
wrench     044 0'<../tools/wrench >wrench'       0'<../module/mod44 > mod44'    0
lathe      055 0'<../tools/lathe >lathe'         0'<../module/mod55 > mod55'    0
drill      066 0'<../tools/drill >drill'         0'<../module/mod66 > mod66'    0
pliers     077 0'<../tools/pliers >pliers'       0'<../module/mod77 > mod77'    0
                .
                .
                .
```

Retrieving LONG Column Data with a SELECT, FETCH, or REFETCH Command

The following syntax represents the available subset when your select list includes one or more LONG columns. Remember, a LONG column can be referenced only in a select list and/or a host variable declaration.

$$
\texttt{SELECT} \; [\,\texttt{ALL}\,] \; \left\{ \begin{array}{l} * \\ [\,Owner.\,]\,Table.* \\ CorrelationName.* \\ CorrelationName.ColumnName \end{array} \right\} [\,,\,\cdots\,]
$$

$$
[\,\texttt{INTO}\; HostVariableDeclaration\,] \; \texttt{FROM} \; \{\,[\,Owner.\,]\,FromTableName\;[\,CorrelationName\,]\,\} \\ [\,,\,\cdots\,]
$$

As we noted earlier, the concept of how LONG column data is retrieved differs from that of standard columns. The LONG column descriptor (rather than the data itself) is selected or fetched into a host variable. In the case of a dynamic FETCH command, the LONG column descriptor information goes to the data buffer. In any case, the LONG column data is written to a file or shared memory as specified.

When the following SELECT command is executed, :HostPartPic will contain the LONG column descriptor information for column PartPicture. LONG column data will go to the output device specified when column PartPicture was last inserted or updated.

```
SELECT PartNumber, PartPicture
  INTO :HostPartNum, :HostPartPic
  FROM PartsTable
 WHERE PartNumber = 200
```

Using the LONG Column Descriptor

ALLBASE/SQL does not swap LONG column data into or out of a host variable. Instead a 96-byte descriptor is available to your program at select or fetch time. It contains LONG column information for your program for which you must declare an appropriate host variable.

For example, if you do not know the output device type and its name or address, you obtain this information from the descriptor. Then open the appropriate file or call the operating system to access shared memory. Table 12-2 shows the format of the LONG column descriptor.

Note   The LONG column descriptor must be declared whether or not you access its contents in your code.

Table 12-2. LONG Column Descriptor

Description	Possible Binary Values	Byte Range
Name or Address of Output Device	File name or shared memory ID	1 through 44
Output Device Options	0 = no output specified 1 = overwrite 2 = append 3 = wildcard 4 = overwrite and wildcard 5 = append and wildcard	45
Output Device Type	0 = no device specified 1 = file 2 = shared memory	46
Input Device Type	0 = no device specified 1 = file 2 = shared memory	47
Reserved for Internal Use		48
Size in Bytes of LONG Column Data	1 to $2^{31}-1$	49 through 52
Reserved for Internal Use		53 through 96

Parsing LONG Column Descriptors

Record structures cannot be declared as host variables unless they are to be used in BULK operations. To parse the LONG column descriptor returned by a SELECT or FETCH statement, copy it to a record like the one shown below.

```
(* Use case 0 when you don't need to break down the descriptor.    *)
(* Use case 1 when you want to access a portion of the descriptior. *)

type
  desc_type = packed record
        case integer of
        0:(lfhv   : packed array [1..96] of char);
        1:(name   : packed array [1..44] of char;
                opt    : 0..255;
                outdev : 0..255;
                indev  : 0..255;
                xxx    : 0..255;
                size   : integer;
                intern : packed array [1..44] of char);
             end;
```

Using LONG Columns with a SELECT Command

When you use the BULK SELECT command with long columns, should an error occur before completion of the BULK SELECT command, any operating system files written before the error occurred remain on the system, and LONG column descriptors written to a host variable array remain. It is your responsibility to remove such files as appropriate.

Using LONG Columns with a Dynamic FETCH Command

If you have the need to dynamically retrieve LONG column data, the sqlrowbuf column of the SQLDA, as always, contains the address of the data buffer. However, the data buffer, rather than containing LONG column data, holds the 96-byte LONG column descriptor.

The sqltype field of the format array holds a data type ID number of 15 for a LONG BINARY column and 16 for a LONG VARBINARY column. And the sqltotallen and sqlvallen columns will always contain a value of 96 (indicating the length of the descriptor).

When a NULL is fetched as the LONG column value, no external files are created, and the associated indicator variable for the LONG column descriptor is set to −1.

Changing a LONG Column with an UPDATE [WHERE CURRENT] Command

When you issue an UPDATE command on a LONG column, you have the following options:

- Change the stored data as well as the output device name and/or options.
- Change the stored data only.
- Change the output device name and/or options only.

Specify a LONG column I/O string (discussed earlier in this chapter) following the SET clause, for each LONG column to be updated. You must specify either the input device, the output device, or both. Complete syntax with examples is found in the *ALLBASE/SQL Reference Manual* .

In the following example, the LONG column I/O string is contained in host variable PartPictureIO.

```
UPDATE PartsTable
   SET PartPicture = :PartPictureIO
 WHERE PartName = 'saw'
```

Removing LONG Column Data with a DELETE [WHERE CURRENT] Command

Syntax for the DELETE and DELETE WHERE CURRENT commands is unchanged for use with LONG columns. It is limited for the DELETE command in that a LONG column cannot be used in the WHERE clause.

In the following example, any rows in PartsTable with the PartName of hammer are deleted.

```
DELETE FROM PartsTable WHERE PartName = 'hammer'
```

When LONG column data is deleted, the space it occupied in the DBEnvironment is released when your transaction ends. But any data file selected earlier still exists on the operating system. You may want to design a "cleanup" strategy for such files that are no longer needed.

Coding Considerations

File versus Shared Memory

Depending on your application, you might want to use a file or shared memory as your input or output device. Whereas shared memory may provide faster data access, a file may provide greater security. Consider how memory will be used by other processes and how much memory will be available. What about using a file as an I/O device? You might ask yourself the following questions. Whom do you want to access the file during and after the application transaction is complete? How will it be "cleaned up" when it is no longer being used; perhaps the overwrite option would be helpful, or you could create a maintenance procedure.

File Naming Conventions

When a LONG column is selected or fetched, data goes to the output device you have specified at insert or update time. In the case of a file, because this output device name can be completely defined by you, partially defined by you, or assigned by ALLBASE/SQL, you may want to consider whether or not naming conventions are necessary. For instance, if your application is such that you can always give the same name to your LONG column output device as you give to the standard column you use in the WHERE clause, no need exists to extract the device name from the LONG column descriptor when you select or fetch it. For example, assuming your WHERE clause uses the PartsTable PartName column, the data_file example in the previous section, "Example Data File," uses this strategy. (Your application might still require information other than a file name from the descriptor area.)

Considering Multiple Users

With multiple users reading the same LONG column data, it is preferable for each user to run the application in a local area. This can prevent file access problems.

If several users must access the same data from the same directory, you might want to use the wildcard option ($) and avoid using the overwrite option (!).

Deciding How Much Space to Allocate and Where

Remember to consider the space requirements of any DBEFileSet used for LONG column data. For example, suppose you execute an INSERT or UPDATE command for a LONG column defined as VARBINARY. If inadequate space is available in the database for the new data, an error message is returned to your program, and the transaction is rolled back. In this case, you can create another DBEFile and add it to the appropriate DBEFileSet.

You will also want to consider the amount of shared memory available for your use in relation to the size and number of LONG columns to be selected or fetched.

To place data in shared memory use the wildcard option ($) because Pascal cannot specify random heap space address. The actual shared memory address will be placed in the LONG column descriptor. Refer to this address when using SELECT and UPDATE commands.

13

Programming with ALLBASE/SQL Functions

This chapter highlights some functions available in ALLBASE/SQL. The functions return values that can be used to access, search, update, and delete data. Refer to the "Expressions" chapter of the *ALLBASE/SQL Reference Manual* for a discussion of other available ALLBASE/SQL functions. The ALLBASE/SQL functions discussed in this chapter are as follows:

- Date/Time functions.
- Tuple Identifier (TID) function.

Programming with Date/Time Functions

Seven functions can be used with date/time data types. These functions provide flexibility for inputting and retrieving date/time data from the database.

These functions can be used with a preprocessed application or with ISQL. This chapter outlines basic principles for using date/time functions in an application program. The following sections are included:

- Where Date/Time Functions Can Be Used.
- Defining and Using Host Variables with Date/Time Functions.
- Using Date/Time Input Functions.
- Using Date/Time Output Functions.
- Using the Date/Time ADD_MONTHS Function.
- Coding Considerations.
- Program Examples for Date/Time Data.

Date/time functions are used as you would use an expression. And when used in a select list, all date/time functions produce data output. Refer to the section in this chapter, "Defining and Using Host Variables with Date/Time Functions."

Refer to the "Host Variables" chapter for more information on date/time data types. Complete syntax and format specifications for date/time functions are found in the *ALLBASE/SQL Reference Manual* in the chapters on "Data Types" and "Expressions."

Suppose for example that you are programming for an international corporation. Your database tables contain various date/time columns and the data is used by employees in several countries. You write a generic program on which you base a set of customized programs, one for each geographical location. Each customized program allows the employees at a given location to input and retrieve date/time information in the formats with which they are most comfortable.

Note   For all date/time functions, character input and output values are in
n-computer format.

Where Date/Time Functions Can Be Used

Use date/time functions, as you would an expression, in the DML operations listed below:

Table 13-1. Where to Use Date/Time Functions

DML Operation	Clause
INSERT [1]	VALUES
	WHERE
UPDATE or UPDATE WHERE CURRENT	SET
	WHERE
DELETE or DELETE WHERE CURRENT	WHERE
SELECT	Select list [2]
	WHERE
DECLARE	Select list [2]
	WHERE

[1] The output functions, TO_CHAR and TO_INTEGER, and the ADD_MONTHS function, are limited to use in the select list and the WHERE clause of a Type 2 INSERT.

[2] Input functions, TO_DATE, TO_TIME, TO_DATETIME, and TO_INTERVAL, are generally not appropriate in a select list.

Defining and Using Host Variables with Date/Time Functions

Date/time functions can be used in the way an expression is used; that is, in a select list to indicate the columns you want in the query result, in a search condition to define the set of rows to be operated on, and to define the value of a column when using the UPDATE command. (See the *ALLBASE/SQL Reference Manual* for in-depth information regarding expressions.)

Whether you use host variables or literal strings to specify the parameters of the date/time functions depends on the elements of your application and on how you are using the functions. This section focuses on the use of host variables.

You can use host variables to specify input or output format specifications. Use them as well to hold data input to and any resulting data output from the date/time functions. (Host variables cannot be used to indicate column names.)

Host variables for format specifications must be defined in your application to be compatible with ALLBASE/SQL CHAR or VARCHAR data types. The exception is the ADD_MONTHS function which requires an INTEGER compatible host variable.

As for host variables containing input and output data, define them to be CHAR or VARCHAR compatible with one exception. The TO_INTEGER function requires an INTEGER compatible host variable for its output.

Reference the chapter on defining host variables for additional information on host variable ALLBASE/SQL data type compatibility. Note that the declarations relate to the default format specification for each date/time data type. Your declaration must reflect the length of the format you are using.

Table 13-2 shows host variable data type compatibility for date/time functions.

Table 13-2. Host Variable Data Type Compatibility for Date/Time Functions

Date/Time Function	Input Format Specification	Output Format Specification	Input Data	Output Data
TO_DATE TO_TIME TO_DATETIME TO_INTERVAL	(VAR)CHAR		(VAR)CHAR	(VAR)CHAR [1]
TO_CHAR		(VAR)CHAR		(VAR)CHAR
TO_INTEGER		(VAR)CHAR		INTEGER
ADD_MONTHS	INTEGER			(VAR)CHAR [1]
[1] Applies only when used in a select list.				

Using Date/Time Input Functions

The input functions are designed so that you can easily input data for a given date/time data type in either the default format or a format of your choice. (When you do not include a format specification, the default is used.)

You have the option of choosing a literal string or a host variable to indicate a desired data value and/or optional format specification. See the *ALLBASE/SQL Reference Manual* for detailed syntax.

Following is the general syntax for date/time input functions:

$$
\left\{
\begin{array}{l}
\text{TO\_DATETIME } (DataValue\ [\ ,FormatSpecification]) \\
\text{TO\_DATE } (DataValue\ [\ ,FormatSpecification]) \\
\text{TO\_TIME } (DataValue\ [\ ,FormatSpecification]) \\
\text{TO\_INTERVAL } (DataValue\ [\ ,FormatSpecification])
\end{array}
\right\}
$$

Input functions can be used in DML operations as shown in Table 13-1. It is most appropriate to use date/time input functions in a WHERE, VALUES, or SET clause. Although they can be used in a select list, it is generally not appropriate to do so. The data value returned to the function in this instance is not a column value but is identical to the value you specify as input to the function.

Examples of TO_DATETIME, TO_DATE, TO_TIME, and TO_INTERVAL Functions

Imagine a situation in which users will be inputting and retrieving date/time data in formats other than the default formats. (Refer to the *ALLBASE/SQL Reference Manual* for default format specifications.)

The data is located in the TestData table in the manufacturing database. (Reference appendix C in the *ALLBASE/SQL Reference Manual* .)

You are to provide them with the capability of keying and retrieving data in the formats shown in Table 13-3.

Table 13-3. Sample of User Requested Formats for Date/Time Data

Date/Time Data Type	Desired Format Specification	Length of Format Specification in ASCII Characters
DATETIME	MM-DD-YYYY HH:MM:SS.FFF	23
DATE	MM-DD-YYYY	10
TIME	HH:MM:SS [1]	8
INTERVAL	DDDDDDD HH:MM:SS	16
[1] This is the default time data format.		

You might use the following generic code examples to meet their needs.

Example Using the INSERT Command.

Your application allows users to enter data in their desired formats with a minimum of effort on your part.

```
BEGIN DECLARE SECTION
```

Declare input host variables (:BatchStamp, :BatchStamp-Format, :TestDate, :TestDate-Format, :TestStart, :LabTime, and :LabTime-Format) to be compatible with data type CHAR or VARCHAR.

Declare input indicator variables (:TestDateInd, TestStartInd, and :LabTimeInd).

```
END DECLARE SECTION
⋮

INSERT
  INTO MANUFDB.TESTDATA
      (BatchStamp,
       TestDate,
       TestStart,
       TestEnd,
```

```
                 LabTime,
                 PassQty,
                 TestQty)
     VALUES ( TO_DATETIME (:BatchStamp, :BatchStamp-Format),
              TO_DATE (:TestDate :TestDateInd, :TestDate-Format),
              TO_TIME (:TestStart :TestStartInd),
              :TestEnd :TestEndInd,
              TO_INTERVAL (:LabTime :LabTimeInd, :LabTime-Format),
              :PassQty :PassQtyInd,
              :TestQty :TestQtyInd)
```

Note that the user requested time data format is the default format. Using the two time data columns in the TestData table (TestStart and TestEnd), the above example illustrates two ways of specifying a default format. Specify a date/time function without a format, or simply do not use a date/time function.

Example Using the UPDATE Command.

These users want the capability of updating data based on the BatchStamp column.

```
     BEGIN DECLARE SECTION
```

*Declare input host variables (:TestDate, :TestDate-Format, :BatchStamp,
and :BatchStamp-Format) to be compatible with data type CHAR or VARCHAR.*

Declare input indicator variable (:TestDateInd).

```
     END DECLARE SECTION
     ⋮
     UPDATE MANUFDB.TESTDATA
        SET TESTDATE = TO_DATE
                      (:TestDate :TestDateInd, :TestDate-Format),
                   TestStart = :TestStart :TestStartInd,,
                   TestEnd = :TestEnd :TestEndInd,,
                   LabTime = :LabTime :LabTimeInd,
                   PassQty = :PassQty :PassQtyInd,
                   TestQty = :TestQty :TestQtyInd
        WHERE BatchStamp = TO_DATETIME
                      (:BatchStamp, :BatchStamp-Format)
```

Example Using the SELECT Command.

The users are planning to select data from the TestData table based on the lab time interval between the start and end of a given set of tests.

```
BEGIN DECLARE SECTION

Declare input host variables (:BatchStamp, :BatchStamp-Format,
:LabTime, and :LabTime-Format) to be compatible with data type
CHAR or VARCHAR.

END DECLARE SECTION
.
.
.
SELECT BatchStamp
       TestDate
       TestStart,
       TestEnd,
       LabTime
       PassQty,
       TestQty
  INTO :BatchStamp,
       :TestDate :TestDateInd,
       :TestStart :TestStartInd,
       :TestEnd :TestEndInd,
       :LabTime :LabTimeInd,
       :PassQty : PassQtyInd,
       :TestQty :TestQtyInd
  FROM MANUFDB.TESTDATA
 WHERE LabTime > TO_INTERVAL (:LabTime, :LabTime-Format)
   AND TO_DATETIME (:BatchStamp, :BatchStamp-Format),
BETWEEN :StampOne AND :StampTwo
```

Example Using the DELETE Command.

The users want to delete data from the TestData table by entering a value for the BatchStamp column.

```
BEGIN DECLARE SECTION

Declare input host variables (:BatchStamp and :BatchStamp-Format)
to be compatible with data type CHAR or VARCHAR.

END DECLARE SECTION
⋮
DELETE FROM MANUFDB.TESTDATA
 WHERE BatchStamp = TO_DATETIME (:BatchStamp, :BatchStamp-Format)
```

Using Date/Time Output Functions

Specify the output format of any type of date/time column by using a date/time output function. Use an output function with any DML operation listed in Table 13-2 with one exception. In the case of a INSERT command, output functions are limited to use in the select list and the WHERE clause of a Type 2 INSERT command.

As with date/time input functions, use a host variable or a literal string to indicate a format specification. See the *ALLBASE/SQL Reference Manual* for detailed syntax.

Following is the general syntax for date/time output functions:

$$\left\{ \begin{array}{l} \texttt{TO\_CHAR}\ (\mathit{ColumnName}\ [\ ,\mathit{FormatSpecification}\,]) \\ \texttt{TO\_INTEGER}\ (\mathit{ColumnName},\ \mathit{FormatSpecification}) \end{array} \right\}$$

Example TO_CHAR Function

The default format for the DATETIME data type specifies the year followed by the month followed by the day. The default format for the TIME data type specifies a 24-hour clock. (Refer to the *ALLBASE/SQL Reference Manual* .)

Suppose users located in Italy want to input a specified batch stamp to obtain the start and end times of the related test in 12-hour format. They will key the batch stamp in this format, "DD-MM-YYYY HH12:MM:SS:FFF AM or PM." The times returned will be in this format, "HH12:MM:SS.FFF AM or PM."

Data is located in the TestData table in the manufacturing database. (Refer to appendix C in the *ALLBASE/SQL Reference Manual* .) The following code could be used:

```
BEGIN DECLARE SECTION

Declare input host variables (:TwelveHourClockFormat, :BatchStamp,
:ItalianFormat, and :SpecifiedInput) to be compatible with data type
CHAR or VARCHAR.

Declare output host variables (:TestStart and :TestEnd) to be compatible
with data type CHAR or VARCHAR .

Declare output indicator variables (:TestStartInd and :TestEndInd).

END DECLARE SECTION
    ⋮
SELECT TO_CHAR(TestStart, :TwelveHourClock),
       TO_CHAR(TestEnd, :TwelveHourClock)
  INTO :TestStart :TestStartInd,
       :TestEnd :TestEndInd,
  FROM ManufDB.TestData
  WHERE TO_DATETIME(:BatchStamp, :ItalianFormat) = :SpecifiedInput
```

Note the use of indicator variables in the above example. Because the TO_CHAR function is used in the select list, no need exists to specify an indicator variable as part of the function.

Example TO_INTEGER Function

The TO_INTEGER format specification is mandatory and differs from that of other date/time functions in that it must consist of a single element only. See the *ALLBASE/SQL Reference Manual* for detailed format specifications.

Perhaps you are writing a management report that indicates the quarter of the year in which tests were performed. (As in the previous example, data is located in the TestData table in the manufacturing database.) You could use the following code:

```
BEGIN DECLARE SECTION
```

Use the ALLBASE/SQL Reference Manual to determine your desired format specification. (In this case it is Q.)

Declare the input host variable, :QuarterlyFormat, to be compatible with data types CHAR or VARCHAR.

Declare an output host variable (:TestDateQuarter) to be compatible with data type INTEGER. Declare other output host variables (:BatchStamp, :LabTime, :PassQty, and :TestQty) to be compatible with data type CHAR or VARCHAR.

Remember to declare output indicator variables (:TestDateQuarterInd, :LabTimeInd, :PassQtyInd, and :TestQtyInd).

```
END DECLARE SECTION
.
.
.
DECLARE ReportInfo CURSOR FOR
                   SELECT BatchStamp,
                          TO_INTEGER(TestDate, :QuarterlyFormat),
                          LabTime,
                          PassQty,
                          TestQty
                     FROM ManufDB.TestData
.
.
.
    FETCH ReportInfo
      INTO :BatchStamp,
           :TestDateQuarter   :TestDateQuarterInd,
           :LabTime           :LabTimeInd,
           :PassQty           :PassQtyInd,
           :TestQty           :TestQtyInd
```

Using the Date/Time ADD_MONTHS Function

This function allows you to add an integer number of months to a DATE or DATETIME column. Do so by indicating the number of months as a positive, negative, or unsigned integer value. (An unsigned value is assumed positive.) Also, you can specify the integer in a host variable of type INTEGER.

The ADD_MONTHS function can be used in both input and output operations as shown in Table 13-2.

Following is the general syntax for the ADD_MONTHS function:

{ ADD_MONTHS (*ColumnName*, *IntegerValue*) }

As with date/time output functions, use the ADD_MONTHS function with any DML operation listed in Table 13-1 with one exception. In the case of a [BULK] INSERT command, the ADD_MONTHS function is limited to use in the select list and the WHERE clause of a Type 2 INSERT command.

Example ADD_MONTHS Function

Perhaps you want to increment each date in the TestDate column by one month in the ManufDB.TestData table of the manufacturing database. The following command could be used:

```
UPDATE ManufDB.TestData
    SET TestDate = ADD_MONTHS (TestDate, 1);
```

Coding Considerations

The following list provides helpful reminders when you are using date/time functions:

- Input functions require leading zeros to match the fixed format of an element. (Z is not supported.)

- For all date/time functions, when you provide only some elements of the complete format in your format specification, any unspecified elements are filled with default values.

- Arithmetic operations are possible with functions of type INTEGER.

- The length of the data cannot exceed the length of the format specification for that data. The maximum size of a format specification is 72 bytes.

- Because LIKE works only with CHAR and VARCHAR values, if you want to use LIKE with date/time data, you must first convert it to CHAR or VARCHAR. For this you can use the TO_CHAR conversion function.

- MIN, MAX, COUNT can be used with any DATE/TIME column type. SUM, AVG can be used with INTERVAL data only.

- Do not specify an indicator variable as a parameter of a date/time function used in the select list of a query.

- When using the ADD_MONTHS function, if the addition of a number of months (positive or negative) would result in an invalid day, the day field is set to the last day of the month for the appropriate year, and a warning is generated indicating the adjustment.

Program Example for Date/Time Data

The next data conversion program is intended as a guide should you decide to convert any character (CHAR) columns in an existing table to a date/time data type.

Before running this program, you must create a new table, PurchDB.NewOrders, in PartsDBE. This table is similar to the PurchDB.Orders table already existing in PartsDBE, except that the OrderDate column is of the DATE data type. You can create the table by issuing the following command from ISQL:

```
CREATE PUBLIC TABLE PurchDB.NewOrders
                  (OrderNumber   INTEGER NOT NULL,
                   VendorNumber  INTEGER,
                   OrderDate     DATE)
              IN OrderFS;
```

```
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
(* This program uses BULK FETCH and BULK INSERT commands to select all *)
(* rows from the Orders table (part of the sample DBEnvironment,       *)
(*PartsDBE), convert the order date column from the CHAR data type to  *)
(*the DATE data type default format, and write all Orders table        *)
(*information to another table called NewOrders table (created         *)
(*previously by you as described in this chapter).                     *)
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

Program pasex9a(input, output);

const
    OK          =        0;
    NotFound    =      100;
    DeadLock    =  -14024;
    NoMemory    =   -4008;

var

         (* Begin Host Variable Declarations *)
    EXEC SQL BEGIN DECLARE SECTION;

(*********************************************************************************)
(* Arrays are NOT packed, although elements within the arrays can be.    *)
(* This prevents possible data allignment problems on Series 300 systems.*)
(*********************************************************************************)
    Orders                  : array[1..25]
      of record
        OrderNumber      : integer;
        VendorNumber     : integer;
        VendorNumberInd  : sqlind;
        OrderDate        : packed array[1..8] of char;
        OrderDateInd     : sqlind;
      end;
    StartIndex             : SmallInt;
    NumberOfRows           : SmallInt;

    NewOrders              : array[1..25]
      of record
        NewOrderNumber   : integer;
        NewVendorNumber  : integer;
        NewVendorNumberInd : sqlind;
        NewOrderDate     : packed array[1..10] of char;
        NewOrderDateInd  : sqlind;
      end;
    StartIndex2            : SmallInt;
    NumberOfRows2         : SmallInt;
```

Figure 13-1. Sample Program Converting Column from CHAR to DATE

```
    SQLMessage                : packed array[1..132] of char;
    EXEC SQL END DECLARE SECTION;
           (* End Host Variable Declarations *)


    SQLCA : SQLCA_type;    (* SQL Communication Area *)

    DoneConvert              : boolean;
    OrdersOK                 : boolean;
    Abort                    : boolean;

    counter1                 : integer;


(***********************************************************************)
(*                    Procedure to release PartsDBE.                 *)
(***********************************************************************)

procedure TerminateProgram;   (* Procedure to Release PartsDBE *)
begin

EXEC SQL RELEASE;

end;  (* End TerminateProgram Procedure *)


(***********************************************************************)
(* Procedure to display error messages and terminate the program when the *)
(* transaction has been rolled back by ALLBASE/SQL.                  *)
(***********************************************************************)

procedure SQLStatusCheck;  (* Procedure to Display Error Messages *)
begin

Abort := FALSE;

if SQLCA.SQLCODE <= DeadLock then Abort := TRUE;
if SQLCA.SQLCODE = NoMemory then Abort := TRUE;

repeat
   EXEC SQL SQLEXPLAIN :SQLMessage;
   writeln(SQLMessage);
until SQLCA.SQLCODE = 0;
if Abort then TerminateProgram;

end;  (* End SQLStatusCheck Procedure *)
```

Figure 13-1. Sample Program Converting Column from CHAR to DATE (page 2 of 8)

```
(*****************************************************************)
(* The cursor for the BULK FETCH is declared in a function that is never *)
(* executed at run time.  The section for this cursor is created and     *)
(* stored in the program module at preprocess time.                      *)
(*****************************************************************)

procedure DeclareCursor;

begin
  EXEC SQL DECLARE OrdersCursor
               CURSOR FOR
      SELECT *
              FROM PurchDB.Orders;
end;


(*****************************************************************)
(* Function to connect to the sample database environment, PartsDBE.    *)
(*****************************************************************)

function ConnectDBE: boolean;
begin

writeln('Connect to PartsDBE');
EXEC SQL CONNECT TO '../sampledb/PartsDBE';

ConnectDBE := TRUE;
if SQLCA.SQLCODE  OK then
  begin

  ConnectDBE := FALSE;
  SQLStatusCheck;

  end;  (* End if *)
end;  (* End of ConnectDBE Function *)


(*****************************************************************)
(* Procedure to begin the transaction with cursor stability specified.   *)
(*****************************************************************)

procedure BeginTransaction;
begin

EXEC SQL BEGIN WORK CS;
```

Figure 13-1. Sample Program Converting Column from CHAR to DATE (page 3 of 8)

```
if SQLCA.SQLCODE  OK then
  begin

  SQLStatusCheck;
  TerminateProgram;

  end;

end;  (* End BeginTransaction Procedure *)

(**************************************************************************)
(* Procedure to commit work to the database OR save the cursor position.  *)
(**************************************************************************)

procedure CommitWork;
begin

writeln('Commit Work');
EXEC SQL COMMIT WORK;
if SQLCA.SQLCODE  OK then
  begin
  SqlStatusCheck;
  TerminateProgram;
  end;

end;  (* End CommitWork Procedure *)

(**************************************************************************)
(* Procedure to rollback the transaction.                                 *)
(**************************************************************************)

procedure RollBackWork;
begin

writeln('Rollback Work');
EXEC SQL ROLLBACK WORK;
if SQLCA.SQLCODE  OK then
  begin
  SqlStatusCheck;
  TerminateProgram;
  end;

end;  (* End RollBackWork Procedure *)
```

Figure 13-1. Sample Program Converting Column from CHAR to DATE (page 4 of 8)

```
(**************************************************************************)
(* Procedure to BULK INSERT into PurchDB.NewOrders table.               *)
(**************************************************************************)

procedure InsertNew;
begin

NumberOfRows2 := counter1;
StartIndex2   := 1;

writeln('BULK INSERT INTO  PurchDB.NewOrders');

EXEC SQL BULK INSERT INTO  PurchDB.NewOrders
                    VALUES (:NewOrders,
                            :StartIndex2,
                            :NumberOfRows2);
case SQLCA.SQLCODE of
OK        : ;

Otherwise   begin
        SQLStatusCheck;
        RollBackWork;
        OrdersOK := FALSE;
                DoneConvert := TRUE;
            end;

end;     (* case *)
end;                 (* End of Procedure InsertNew *)
(**************************************************************************)
(* Procedure to convert OrderDate from CHAR to DATE data type and transfer*)
(* data to an array in preparation for BULK INSERT into a new table.    *)
(**************************************************************************)

   procedure TransferData;

   var

    i,j:integer;

   begin

   NumberOfRows := counter1;

   for i := 1 to NumberOfRows do
   begin
     NewOrders[i].NewOrderNumber := Orders[i].OrderNumber;
     NewOrders[i].NewVendorNumber := Orders[i].VendorNumber;
   end;
```

Figure 13-1. Sample Program Converting Column from CHAR to DATE (page 5 of 8)

```
                        (* Convert Date *)
   for i := 1 to NumberOfRows do
   begin
      for j := 1 to 4 do
         begin
            NewOrders[i].NewOrderDate[j] := Orders[i].OrderDate[j];
         end;
      NewOrders[i].NewOrderDate[5] := '-';
      for j := 6 to 7 do
        NewOrders[i].NewOrderDate[j] := Orders[i].OrderDate[j-1];
        NewOrders[i].NewOrderDate[8] := '-';
      for j := 9 to 10 do
        NewOrders[i].NewOrderDate[j] := Orders[i].OrderDate[j-2];
   end;

end;   (* End of Procedure TransferData *)

(****************************************************************************)
(* Procedure to BULK FETCH Orders table data 25 rows at a time             *)
(* into an array.                                                          *)
(****************************************************************************)

procedure FetchOld;

begin;

NumberOfRows := 25;
StartIndex   := 1;

writeln('BULK FETCH PurchDB.Orders');

EXEC SQL BULK FETCH OrdersCursor
        INTO :Orders, :StartIndex, :NumberOfRows;

counter1 := SQLCA.SQLERRD[3];

case SQLCA.SQLCODE of
OK       :  begin
     CommitWork;                    (* SAVE THE CURSOR POSITION *)
            end;                            (* Used in conjunction with *)
                                            (* cursor stability.        *)
NotFound :  begin
              CommitWork;
      writeln;
      writeln('There are no Orders Table rows to FETCH.');
              DoneConvert := TRUE;
            end;
```

Figure 13-1. Sample Program Converting Column from CHAR to DATE (page 6 of 8)

```
Otherwise   begin
      SQLStatusCheck;
      RollBackWork;
      OrdersOK := FALSE;
             DoneConvert := TRUE;
           end;

end;     (* case *)

if not DoneConvert then
TransferData;

if not DoneConvert then
InsertNew;

end;     (* End of procedure FetchOld *)

(**************************************************************************)
(*                          Beginning of program.                       *)
(**************************************************************************)

begin

writeln('Program to convert date from CHAR to DATE data type.');
writeln('Event List:');
writeln('  Connect to PartsDBE');
writeln('  BULK FETCH all rows from Orders Table.');
writeln('  Convert the date.');
writeln('  BULK INSERT all fetched rows into NewOrders Table' );
writeln('  with converted date.');
writeln('  Release PartsDBE');
writeln;


if ConnectDBE then

   begin

   DoneConvert := FALSE;
   OrdersOK := TRUE;

   BeginTransaction;
   EXEC SQL OPEN OrdersCursor KEEP CURSOR WITH LOCKS;
```

Figure 13-1. Sample Program Converting Column from CHAR to DATE (page 7 of 8)

```
    if SQLCA.SQLCODE  OK then
       begin
       SQLStatusCheck;
       RollBackWork;
       OrdersOK := FALSE;
       DoneConvert := TRUE;
       end;

    repeat
      FetchOld
    until DoneConvert;     (* DoneConvert is TRUE when all data has been   *)
                           (* converted and inserted or when an error       *)
                           (* condition not serious enough for ALLBASE/SQL *)
                           (* to rollback work was encountered.            *)

 if OrdersOK then          (* If there were no errors in processing, data  *)
    CommitWork;            (* is committed to the database.                *)

    end;

TerminateProgram;

end.  (* End of Program *)
```

Figure 13-1. Sample Program Converting Column from CHAR to DATE (page 8 of 8)

Programming with TID Data Access

Each row (tuple) in an ALLBASE/SQL table is stored at a database address on disk. This unique address is called the **tuple identifier** or **TID**. When using a SELECT statement, you can obtain the TID of any row. In turn, you can use this TID to specify the target row for a SELECT, UPDATE, or DELETE statement. TID functionality provides the fastest possible data access to a single row at a time (TID access) in conjunction with maximum coding flexibility. The following options are available:

- Rapid read and write access to a specific row without the use of a cursor (less overhead).
- Rapid update and delete capability based on TIDs returned by a nested query, a union query, a join query, or a query specifying sorted data.

Other ALLBASE/SQL functionality provides a method of processing a multiple row query result sequentially, one row at a time. This involves the use of a cursor with the UPDATE WHERE CURRENT, DELETE WHERE CURRENT, and REFETCH commands which internally utilize TID access. See the *ALLBASE/SQL Reference Manual* for more details.

The nature of your applications will determine how valuable TID functionality can be to you. It could be most useful for applications designed for interactive users and applications that must update a set of related rows as a group. See the programming examples at the end of this chapter.

A TID function and host variable data type are provided. The TID function is used in the select list and/or the WHERE clause of a SELECT statement and in the WHERE clause of an UPDATE or DELETE statement. The new host variable data type is used in an application program to hold data input to and output from the TID function.

Understanding TID Function Input and Output

The next sections describe how TID output is accessed via a select list and how you provide TID input via a WHERE clause. Topics discussed are as follows:

- Using the TID Function in a Select List.
- Using the TID Function in a WHERE Clause.
- Declaring TID Host Variables.
- Understanding the SQLTID Data Format.

Using the TID Function in a Select List

When using the TID function in a select list, specify it as you would a column name. In an application, you could use a statement like the following:

```
SELECT TID(), VendorNumber, VendorName, PhoneNumber
       INTO   :TidHostVar, :VendorNumber,
              :VendorName, :PhoneNumber;
       FROM   Purchdb.Vendors
       WHERE  VendorName = :VendorName
```

The resulting TID and column data is placed in the host variables, TidHostVar, VendorNumber, VendorName, PhoneNumber.

The next example illustrates how to obtain TID values for qualifying rows of a two table join. Correlation names are used.

```
      SELECT TID(sp), TID(o)
        INTO :SupplyPriceTID, :OrdersTID,
        FROM PurchDB.SupplyPrice sp,
             PurchDB.Orders o
      WHERE sp.VendorNumber = :VendorNumber
        AND  o.VendorNumber = :VendorNumber
```

Using the TID Function in a WHERE Clause

When using the TID function in a WHERE clause, you provide an input parameter. For application programs, this parameter can be specified as a host variable, a constant, or a question mark (?) representing a dynamic parameter. The input parameter is a constant. For example:

```
   DELETE FROM PurchDB.Parts WHERE TID() = 3:3:30;
```

In an application, you could use a statement like the following to verify the data integrity of a previously accessed row:

```
      SELECT  PartNumber, PartName, SalesPrice
        INTO  :PartNumber, :PartName, :SalesPrice
        FROM  PurchDB.Parts
       WHERE  TID() = :PartsTID
```

You might use the following statement in an application to update a row:

```
   UPDATE PurchDB.Parts
      SET PartNumber = :PartNumber,
          PartName = :PartName,
          SalesPrice = :SalesPrice
    WHERE TID() = :PartsTID
```

Declaring TID Host Variables

Host variables for TID function input and output must be declared in your application as SQLTID host variables. You would declare an SQLTID host variable as follows:

```
   TIDVarName : SQLTID;
```

Understanding the SQLTID Data Format

The data in SQLTID host variables has its own unique format which is not compatible with any other ALLBASE/SQL data type. It is *not* necessary to know the internal format of SQLTID data to use the TID function. The information in this section is provided in case you require the TID value to be broken into its components.

For instance, you might want to know the page numbers of all TID's in a table in order to analyze data distribution. To do this, you must parse the SQLTID host variable.

ALLBASE/SQL does allow you to unload SQLTID data. However, you cannot use the LOAD command to load TID data back into a table. The TID is a unique identifier generated internally by ALLBASE/SQL, and cannot be assigned by users.

An SQLTID host variable consists of eight bytes of binary data and has the following format:

Table 13-4. SQLTID Data Internal Format

Content	Byte Range
Version Number	1 through 2
File Number	3 through 4
Page Number	5 through 7
Slot Number	8

The SQLTID version number is an optional input parameter. If not specified, the version number defaults to 0. If you do specify the version, it must always be 0. If a version other than 0 is specified, no rows will qualify for the operation.

TID function application output always contains a version number of 0.

Transaction Management with TID Access

TID data access is fast, and it must be used with care. A great deal of flexibility of use is possible, and exactly how it should be used depends on your application programming needs.

The next sections look at performance, concurrency and data integrity issues involved in designing database transactions that use TID access. Although a possible usage scenario is given, you must decide how to combine the elements of transaction management to best suit your purposes. The following concepts are highlighted:

- Comparing TID Access to Other Types of Data Access.
- Ensuring that the Optimizer Chooses TID Access.
- Verifying Data that is Accessed by TID.
- Stable versus Volatile Data.
- Using Isolation Levels with the TID Function.
- Considering Interactive User Applications.
- Coding Strategies.

TID access requires an initial SELECT, BULK SELECT, FETCH or BULK FETCH to obtain TID values. You can then select, update or delete data by TID.

Comparing TID Access to Other Types of Data Access

When using TID functionality, data access speed is always improved compared to the speed of other ALLBASE/SQL access methods, for the following reasons:

- Index access must lock more pages (i.e. index pages).
- Relation access locks more pages to find the TID of any qualifying row.
- Hash access employs more search overhead.

Note that use of the TID function in a WHERE clause does *not* guarantee that TID access will be chosen by the optimizer. For example, the following statement would utilize TID access.

```
DELETE FROM PurchDB.Parts
     WHERE TID() = :PartsTID AND PartName = 'Winchester Drive'
```

However, in the next statement TID access would not be used because it uses an OR:

```
DELETE FROM PurchDB.Parts
     WHERE TID() = :PartsTID1 OR TID() = :PartsTID2
```

See the "Expressions" chapter of the *ALLBASE/SQL Reference Manual* for an explanation of the above restriction on the OR and additional optimization criteria.

Verifying Data that is Accessed by TID

It is important to note that a TID in ALLBASE/SQL is unique and is valid until its related data is deleted. You must take precautions to assure that the data you are selecting or changing exists when your statement executes. (Note that a TID can be reassigned after its data has been deleted.)

You can rely on the existence of a given TID, if you *know* its data won't be deleted. That is, you know the nature of the data is non-volatile. In this case, you can select the TID and update by TID with the assurance that data integrity will be maintained. An example might be a table that has been created as private. Another example might be a table that you know is currently being accessed only by your application. (You have begun the transaction with the RR isolation level, or you have used the LOCK TABLE command.)

By contrast, you may be dealing with data that changes frequently. In cases where you are using the CS, RC, or RU isolation levels, you must verify that your data has not changed between the time you select it and the time you update or delete it. A method is to end the transaction in which you selected the data, and begin an RR transaction in which you use a SELECT statement with the TID function in the WHERE clause. See the following section titled "Coding Strategies" for an example.

When you attempt to access a row for update or delete, the status checking procedure is the same as for a statement that does not contain the TID function. An application must check the sqlcode field of the sqlca for a value of 100.

Considering Interactive User Applications

Some transaction management basics that apply to TID functionality when used in interactive applications are listed below:

- Be sure to avoid holding locks against the database within a transaction driven by interactive user input. This is sometimes termed "holding locks around terminal reads." It means that the speed at which the user enters required data determines the execution time of your transaction and thus the time span of transaction locks.
- Does your transaction use the RR isolation level? If so, there is no need to verify your data prior to updating or deleting within the same transaction.
- Does your transaction use the CS, RC, or RU isolation level? If so, in order to maintain data integrity, you *must* verify that the data has not changed before you attempt to update or delete it. By verifying the data in this way, you insure that it still exists and can determine whether or not it has changed from the time it was last presented to the user.

Coding Strategies

Suppose you are writing an application that will be executed by many simultaneous users in an online transaction processing environment. You want each user to be able to locate and update just a few rows in a table that is frequently accessed by many users.

The following scenario illustrates the use of two transactions with different isolation levels. Figure 13-2 uses the RC isolation level with a BULK SELECT statement to obtain data and RR isolation level with a SELECT statement based on TID access to verify the data before it is updated.

Define two arrays, one (OrdersArray) to hold the qualifying rows of the Orders table and another (NewOrdersArray) to hold the rows that the user wants to change. Be sure to define an element in each array to hold the TID value.

Begin the transaction with RC isolation level. This ensures maximum concurrency for committed data. Locks are released immediately following data access.

```
BEGIN WORK RC

BULK SELECT TID(), OrderNumber, VendorNumber, OrderDate
     INTO :OrdersArray, :StartIndex, :NumberOfRows;
     FROM PurchDB.Orders
     WHERE OrderNumber BETWEEN 30510 AND 30520

COMMIT WORK
```

Once all qualifying rows have been loaded into OrdersArray, end the transaction. Then loop through the array displaying the rows and accepting any user entered changes in NewOrdersArray. Include the appropriate TID values with each NewOrdersArray entry.

Figure 13-2. Using RC and RR Transactions with BULK SELECT, SELECT, and UPDATE

```
BEGIN WORK RR
        For each entry in NewOrdersArray, do the following:

        SELECT TID(), *
          INTO :TIDvalue, :OrderNumber, :VendorNumber, :OrderDate
          FROM PurchDB.Orders
         WHERE TID() = :TIDHostVariable

        Verify the selected data against the corresponding data in OrdersArray.
        If the row is unchanged, update it using TID access.

        UPDATE PurchDB.Orders
           SET   OrderNumber = :NewOrderNumber :NewOrderNumberInd,
                 VendorNumber = :NewVendorNumber :NewVendorNumberInd,
                 OrderDate = :NewOrderDate :NewOrderDateInd
                 WHERE TID() = :TIDHostVariable

        If the row has changed or has been deleted, inform the user and offer
        appropriate options.
   COMMIT WORK
```

Figure 13-2. Using RC and RR Transactions with BULK SELECT, SELECT, and UPDATE (2 of 2)

Reducing Commit Overhead for Multiple Updates with TID Access

Figure 13-3 shows how to reduce commit overhead when performing multiple updates following a BULK FETCH. Two loops are used, each with its own cursor and own set of locks.

In the outer loop, a BULK FETCH is performed with a cursor to load an array. The transaction enveloping the outer loop uses an RC isolation level to allow maximum concurrency while the user is entering data at the terminal. The locks associated with the BULK FETCH cursor are released after each fetch.

The inner loop uses another cursor to FETCH a single row of data based on the TID value. Since the RC isolation level is being used, the data must be refetched to prevent other transactions from modifying it. The data is verified, and an UPDATE is performed.

After the inner loop has finished updating the rows of data, a COMMIT WORK is issued to actually commit the updates to the data base and to release the exclusive locks held by the updates in the inner loop. This use of a single COMMIT WORK for the multiple updates in the inner loop reduces overhead.

Define two arrays, one (PartsArray) to hold the qualifying rows of the Parts table and another (NewPartsArray) to hold the rows that the user wants to change. Be sure to define an element in each array to hold the TID value. Declare the cursor (BulkCursor) used by the BULK FETCH ④ *that loads the PartsArray.*

```
DECLARE BulkCursor CURSOR FOR
 SELECT TID(), PartNumber, PartName, SalesPrice
   FROM PurchDB.Parts
```

Declare the cursor (TidCursor) used to UPDATE ⑪ *an individual row based on the TID value.*

```
DECLARE TidCursor CURSOR FOR                                         [1]
 SELECT PartName, SalesPrice
   FROM PurchDB.Parts
  WHERE TID() = :HostPartTid
    FOR UPDATE OF PartName, SalesPrice
```

Begin the transaction with the RC isolation level. This ensures maximum concurrency while assuring that only commited data is read.

```
BEGIN WORK RC                                                       [2]
```
OPEN the cursor associated with the BULK FETCH ④. *The KEEP CURSOR parameter maintains the cursor position across transactions until the CLOSE* ⑥ *statement. The WITH NOLOCKS parameter releases all locks associated with the cursor when the COMMIT WORK* ⑦ *statement is executed.*
```
OPEN BulkCursor KEEP CURSOR WITH NOLOCKS
```

The following COMMIT WORK ③ *statement preserves the open cursor position and automatically starts a new transaction with an RC isolation level.*

```
COMMIT WORK                                                         [3]
```
Loop until no more rows are fetched.

```
   BULK FETCH BulkCursor INTO :PartsArray                           [4]
```

   Display the rows in PartsArray and move any changes entered by the user to NewPartsArrray. Include the appropriate TID value with each NewPartsArray entry.

   For each row in the NewPartsArray
       VerifyAndUpdate ⑧
   End For

Figure 13-3. Using TID Access to Reduce Commit Overhead

The following COMMIT WORK ⑤ *statement commits the updates* ⑪ *in VerifyAndUpdate and releases the locks held.*

```
    COMMIT WORK                                              ⑤
```

End Loop

```
CLOSE BulkCursor                                            ⑥
```

The final COMMIT WORK ⑦ *statement ends the transaction started by the BEGIN WORK RC* ②. *Any locks still held are released.*

```
COMMIT WORK                                                ⑦
```
Begin the VerifyAndUpdate routine.   ⑧

Assign to HostPartTid the TID value in NewPartsArray.

```
    OPEN TidCursor
```

Using the cursor declared above ① *as TidCursor, perform a FETCH* ⑨ *and REFETCH* ⑩ *to verify the data. The REFETCH* ⑩ *places a lock on the data page, to prevent another transaction from modifying the data. The lock is held until all the rows in the NewPartsArray have been updated and the COMMIT WORK* ⑤ *is performed.*

```
FETCH TidCursor INTO :PartName, :SalesPrice                ⑨
```

```
REFETCH TidCursor INTO :PartName, :SalesPrice             ⑩
```

Verify the fetched data against the corresponding row in PartsArray. If the row is unchanged, update it using the TID cursor.

```
    UPDATE PurchDB.Parts                                   ⑪
          SET PartName   = :NewPartName,
              SalesPrice = :NewSalesPrice
          WHERE CURRENT OF TidCursor
```

If the row has changed or has been deleted, inform the user and offer appropriate options.

```
    CLOSE TidCursor
```

End the VerifyAndUpdate routine.

Figure 13-3. Using TID Access to Reduce Commit Overhead (page 2 of 2)

Index