

HP 9000 Computer Systems  
**ALLBASE/SQL C**  
**Application Programming Guide**



HP Part No. 36217-90014  
Printed in U.S.A. 1992

Fourth Edition  
E0992

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for direct, indirect, special, incidental or consequential damages in connection with the furnishing or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

**Copyright © 1987, 1988, 1989, 1990, 1991, 1992 by Hewlett-Packard Company**

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013. Rights for non-DoD U.S. Government Departments and agencies are as set forth in FAR 52.227-19 (c) (1,2).

Hewlett-Packard Company  
3000 Hanover Street  
Palo Alto, CA 94304 U.S.A.

**Restricted Rights Legend**

---

## 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.

<b>Edition</b>	<b>Date</b>	<b>Product Number</b>	<b>Hardware Series</b>	<b>HP-UX Release</b>
First Edition	July 1987	36217-02A.01.01	800	1.2
First Edition	February 1988	79725-1.01	300	1.2
Second Edition	February 1988	36217-02A.B2.00.03	800	2.0
Second Edition	February 1989	79725-2.00	300	2.0
Third Edition	September 1989	36217-02A.07.00	800	7.0
		HP79725 Rev.7.0	300	7.0
Fourth Edition	September 1992	36217-02A.E1.00	800	8.0
		36217-02A.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)



---

## 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 C 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 400 system information appears here.*

---

This manual is intended as a learning tool and a reference guide for C programmers. It presumes the reader has a working knowledge of C, 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 Programming in C,” 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 Preprocessor,” explains the ALLBASE/SQL preprocessor and how to invoke it.
- Chapter 3, “Host Variables,” describes how to define and use variables to transfer data between your C program and an ALLBASE/SQL DBEnvironment.
- Chapter 4, “Runtime Status Checking and the sqlca,” defines ways to monitor and handle successful and unsuccessful SQL command execution.

Chapters 5 through 11 address the various ways to manipulate data in an ALLBASE/SQL C program.

- Chapter 5, “Simple Data Manipulation,” explains how to operate on one row at a time.
- Chapter 6, “Processing with Cursors,” shows how to process a multiple row query result one row at a time.
- Chapter 7, “BULK Table Processing,” examines the processing of multiple rows at a time.
- Chapter 8, “Using Dynamic Operations,” describes the use of ALLBASE/SQL commands that are preprocessed at runtime.
- Chapter 9, “Programming with Constraints,” compares the use of statement level integrity and row level integrity and discusses the use of integrity constraints.
- Chapter 10, “Programming with LONG Columns,” discusses the LONG BINARY and LONG VARBINARY data types.
- Chapter 11, “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? >> yes**

{ } 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 { ON }  
          { OFF }**

[ ] 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* [OPTION  
                  *parameter*]**



---

## 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.




$\left\{ \begin{array}{l} \mathbf{A} \\ \mathbf{B} \end{array} \right\} | \dots |$

... 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,  represents the carriage return key or  represents the shift key.

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



# Contents

---

<b>1. Getting Started with ALLBASE/SQL Programming in C</b>	
Understanding ALLBASE/SQL Operations . . . . .	1-1
Using DML to Manipulate Data . . . . .	1-2
Using DDL to Define Database Objects . . . . .	1-2
Using DCL to Manage Security . . . . .	1-3
Handling Transactions . . . . .	1-3
Handling Errors . . . . .	1-3
Dynamic and Non-Dynamic Operations . . . . .	1-4
Understanding the Program Life Cycle . . . . .	1-5
Developing ALLBASE/SQL Applications . . . . .	1-5
Creating Source Files . . . . .	1-6
Using Embedded SQL . . . . .	1-7
General Rules for Embedding SQL . . . . .	1-7
Declaring Special Data Structures . . . . .	1-7
Declaring Host Variables . . . . .	1-8
Skeleton Program . . . . .	1-8
Rules of Syntax for Embedded SQL Statements . . . . .	1-10
Preprocessing the Source File . . . . .	1-11
Creating the Modified Source File . . . . .	1-11
Creating Stored Sections . . . . .	1-11
Stored Form of the SQL Command . . . . .	1-12
Optimized Access Instructions . . . . .	1-12
Validity Flag . . . . .	1-12
Runtime Revalidation of Sections . . . . .	1-13
Generating the Message File . . . . .	1-13
Compiling and Linking the Program . . . . .	1-13
Running the Program . . . . .	1-15
Authorizations . . . . .	1-15
Debugging and Testing . . . . .	1-16
Moving into the Production Phase . . . . .	1-16
Installing Program Modules . . . . .	1-16
Granting Module Owner Authorizations . . . . .	1-18
Granting Program User Authorization . . . . .	1-18
Maintaining ALLBASE/SQL Applications . . . . .	1-19
Tuning Performance . . . . .	1-19
Managing Source Code . . . . .	1-20
Updating Application Programs . . . . .	1-20
Changing Program-Related Authorization . . . . .	1-21
Dropping Obsolete Modules . . . . .	1-21
Programming Under the HP-UX Operating System . . . . .	1-22
Access to Directories . . . . .	1-22
Pathname Conventions in Referencing Files . . . . .	1-22

Use of Shared Memory . . . . .	1-23
Series 300 Shared Memory . . . . .	1-24
Choosing an Interprocess Communication Option . . . . .	1-25
Using Fast IPC . . . . .	1-25
Using Signal IPC . . . . .	1-26
Setting the HPSQLfast_ipc Environment Variable . . . . .	1-26
Setting Signal Handling for Debugging . . . . .	1-26
Using Semaphores . . . . .	1-27
Native Language Support . . . . .	1-27
Looking at an Embedded SQL Source Program . . . . .	1-28

## 2. Using the Preprocessor

Invoking the C Preprocessor . . . . .	2-1
Full Preprocessing Mode . . . . .	2-1
Parameters . . . . .	2-2
Description . . . . .	2-3
Authorization . . . . .	2-4
Example . . . . .	2-5
Syntax Checking Mode . . . . .	2-6
Parameters . . . . .	2-6
Description . . . . .	2-6
Authorization . . . . .	2-6
Example . . . . .	2-7
DBEnvironment Access . . . . .	2-8
Running the Preprocessor in the Background . . . . .	2-8
Compiling and Linking . . . . .	2-8
Running the Program . . . . .	2-10
Accessing Multiple DBEnvironments . . . . .	2-11
Identifying Preprocessor Input . . . . .	2-12
Source File . . . . .	2-12
ALLBASE/SQL Message Catalog . . . . .	2-12
Identifying Preprocessor Output . . . . .	2-13
Modified Source File . . . . .	2-14
Include Files . . . . .	2-15
ALLBASE/SQL Message File . . . . .	2-15
Stored Module Containing Sections . . . . .	2-19
Installable Module File . . . . .	2-23
Handling Preprocessor Errors . . . . .	2-23
Preprocessor or DBEnvironment Termination . . . . .	2-23
Preprocessor Invocation Errors . . . . .	2-24
Source File Errors . . . . .	2-24
DBEnvironment Errors . . . . .	2-24
Sample Modified Source File . . . . .	2-25
Sample Preprocessor Generated Include Files . . . . .	2-31

<b>3. Host Variables</b>	
Using Host Variables . . . . .	3-1
Host Variable Names . . . . .	3-2
Input and Output Host Variables . . . . .	3-3
Indicator Variables . . . . .	3-3
Bulk Processing Variables . . . . .	3-5
Declaring Host Variables . . . . .	3-6
Creating Declaration Sections . . . . .	3-6
Declaring Variables for Data Types . . . . .	3-8
CHAR Data . . . . .	3-8
VARCHAR Data . . . . .	3-8
SMALLINT Data . . . . .	3-10
INTEGER Data . . . . .	3-10
FLOAT Data . . . . .	3-10
Floating Point Data Compatibility . . . . .	3-10
BINARY Data . . . . .	3-11
Binary Data Compatibility . . . . .	3-11
Using the LONG Phrase with Binary Data Types . . . . .	3-11
Declaring Host Variables for BINARY Data . . . . .	3-11
Declaring Host Variables for VARBINARY Data . . . . .	3-12
Inserting and Updating VARBINARY Data . . . . .	3-13
Selecting and Fetching VARBINARY Data . . . . .	3-15
DECIMAL Data . . . . .	3-16
DATE, TIME, DATETIME, and INTERVAL Data . . . . .	3-16
Using Default Data Values . . . . .	3-17
Coding Considerations . . . . .	3-18
When the DEFAULT Clause Cannot be Used . . . . .	3-18
Declaring Variables for Compatibility . . . . .	3-19
String Data Conversion . . . . .	3-23
String Data Truncation . . . . .	3-23
Numeric Data Conversion . . . . .	3-24
Declaring Variables for Program Elements . . . . .	3-25
sqlca Array . . . . .	3-25
Dynamic Processing Arrays . . . . .	3-26
Bulk Processing Arrays . . . . .	3-27
Indicator Variables . . . . .	3-27
Dynamic Commands . . . . .	3-27
Savepoint Numbers . . . . .	3-28
Messages from the Message Catalog . . . . .	3-28
DBEnvironment Name . . . . .	3-29
<b>4. Runtime Status Checking and the sqlca</b>	
Purposes of Status Checking . . . . .	4-2
Handling Runtime Errors and Warnings . . . . .	4-2
Maintaining Data Consistency . . . . .	4-2
Checking the Most Recently Executed Command . . . . .	4-3
Using the sqlca . . . . .	4-4
sqlcode . . . . .	4-6
sqlerrd[2] . . . . .	4-8
sqlwarn[0] . . . . .	4-9
sqlwarn[1] . . . . .	4-10

sqlwarn[2]	4-11
sqlwarn[3]	4-11
sqlwarn[6]	4-12
Approaches to Status Checking	4-13
Implicit Status Checking Techniques	4-13
Program Illustrating Implicit and Explicit Status Checking	4-17
Explicit Status Checking Techniques	4-24
Handling Deadlock and Shared Memory Problems	4-30
Determining Number of Rows Processed	4-30
INSERT, UPDATE, and DELETE Operations	4-31
BULK Operations	4-32
Detecting End of Scan	4-35
Determining When More Than One Row Qualifies	4-36
Detecting Log Full Condition	4-37
Handling Out of Space Conditions	4-37
Checking for Authorizations	4-37
<b>5. Simple Data Manipulation</b>	
SQL Commands	5-1
SELECT	5-1
INSERT	5-5
UPDATE	5-6
DELETE	5-7
Transaction Management for Simple Operations	5-7
Sample Program Using Simple DML Commands	5-10
<b>6. Processing with Cursors</b>	
SQL Cursor Commands	6-1
DECLARE CURSOR	6-2
OPEN	6-3
FETCH	6-3
UPDATE WHERE CURRENT	6-5
DELETE WHERE CURRENT	6-7
CLOSE	6-8
Transaction Management for Cursor Operations	6-9
Using KEEP CURSOR	6-10
KEEP CURSOR and Isolation Levels	6-10
OPEN Command Without KEEP CURSOR	6-10
OPEN Command Using KEEP CURSOR WITH LOCKS and CS Isolation Level	6-11
OPEN Command Using KEEP CURSOR WITH NOLOCKS	6-12
KEEP CURSOR and BEGIN WORK	6-13
KEEP CURSOR and COMMIT WORK	6-14
KEEP CURSOR and ROLLBACK WORK	6-14
KEEP CURSOR and Aborted Transactions	6-14
Writing Keep Cursor Applications	6-15
Examples	6-16
Common StatusCheck Procedure	6-17
Single Cursor WITH LOCKS	6-18
Multiple Cursors and Cursor Stability	6-20
Avoiding Locks on Terminal Reads	6-23

Sample Program Using Cursor Operations . . . . .	6-25
<b>7. BULK Table Processing</b>	
Variables Used in BULK Processing . . . . .	7-1
SQL Bulk Commands . . . . .	7-4
BULK SELECT . . . . .	7-4
BULK FETCH . . . . .	7-8
BULK INSERT . . . . .	7-10
Transaction Management for BULK Operations . . . . .	7-12
Sample Program Using BULK Processing . . . . .	7-13
<b>8. Using Dynamic Operations</b>	
Review of Preprocessing Events . . . . .	8-1
Differences between Dynamic and Non-Dynamic Preprocessing . . . . .	8-2
Permanently Stored vs. Temporary Sections . . . . .	8-2
Examples of Non-Dynamic and Dynamic SQL Statements . . . . .	8-4
Why Use Dynamic Preprocessing? . . . . .	8-5
Passing Dynamic Commands to ALLBASE/SQL . . . . .	8-5
Understanding the Types of Dynamic Operations . . . . .	8-6
Preprocessing of Dynamic Non-Queries . . . . .	8-6
Using EXECUTE IMMEDIATE . . . . .	8-6
Using PREPARE and EXECUTE . . . . .	8-8
Preprocessing of Dynamic Queries . . . . .	8-8
Dynamically Updating and Deleting Data . . . . .	8-10
Setting Up the SQLDA . . . . .	8-11
Setting Up the Format Array . . . . .	8-13
Setting Up the Data Buffer . . . . .	8-15
Setting up a Buffer for Query Results of Unknown Format . . . . .	8-15
Setting up a Buffer for Query Results of Known Format . . . . .	8-15
Using the Dynamic Query Data Structures . . . . .	8-16
Parsing the Data Buffer . . . . .	8-19
Preprocessing of Commands That May or May Not Be Queries . . . . .	8-21
Sample Programs Using Dynamic Command Operations . . . . .	8-23
cex10a: Dynamic Commands of Unknown Format . . . . .	8-23
cex10b: Program Using Dynamic Queries of Known Format . . . . .	8-41
<b>9. Programming with Constraints</b>	
Comparing Statement Level and Row Level Integrity . . . . .	9-1
Using Unique and Referential Integrity Constraints . . . . .	9-2
Designing an Application Using Statement Level Integrity Checks . . . . .	9-3
Insert a Member in the Recreation Database . . . . .	9-5
Update an Event in the Recreation Database . . . . .	9-6
Delete a Club in the Recreation Database . . . . .	9-7
Delete an Event in the Recreation Database . . . . .	9-7

<b>10. Programming with LONG Columns</b>	
General Concepts . . . . .	10-2
Restrictions . . . . .	10-4
Defining LONG Columns with a CREATE TABLE or ALTER TABLE Command . . . . .	10-4
Defining Input and Output with the LONG Column I/O String . . . . .	10-5
Putting Data into a LONG Column with a [BULK] INSERT Command . . . . .	10-6
Insert Using Host Variables for LONG Column I/O Strings . . . . .	10-6
Bulk Insert Using Host Variables for LONG Column I/O Strings . . . . .	10-6
Example . . . . .	10-7
Example Data File . . . . .	10-9
Retrieving LONG Column Data with a [BULK] SELECT, FETCH, or REFETCH Command . . . . .	10-10
Using the LONG Column Descriptor . . . . .	10-10
Example LONG Column Descriptor Declaration . . . . .	10-11
Using LONG Columns with a BULK SELECT Command . . . . .	10-12
Using LONG Columns with a Dynamic FETCH Command . . . . .	10-12
Changing a LONG Column with an UPDATE [WHERE CURRENT] Command . . . . .	10-12
Removing LONG Column Data with a DELETE [WHERE CURRENT] Command . . . . .	10-13
Coding Considerations . . . . .	10-13
File versus Shared Memory . . . . .	10-13
File Naming Conventions . . . . .	10-13
Considering Multiple Users . . . . .	10-14
Deciding How Much Space to Allocate and Where . . . . .	10-14
<b>11. Programming with ALLBASE/SQL Functions</b>	
Where Date/Time Functions Can Be Used . . . . .	11-2
Defining and Using Host Variables with Date/Time Functions . . . . .	11-2
Using Date/Time Input Functions . . . . .	11-3
Examples of TO_DATETIME, TO_DATE, TO_TIME, and TO_INTERVAL Functions . . . . .	11-4
Example Using the INSERT Command . . . . .	11-4
Example Using the UPDATE Command . . . . .	11-5
Example Using the SELECT Command . . . . .	11-6
Example Using the DELETE Command . . . . .	11-6
Using Date/Time Output Functions . . . . .	11-7
Example TO_CHAR Function . . . . .	11-7
Example TO_INTEGER Function . . . . .	11-8
Using the Date/Time ADD_MONTHS Function . . . . .	11-9
Example ADD_MONTHS Function . . . . .	11-9
Coding Considerations . . . . .	11-9
Program Example for Date/Time Data . . . . .	11-10
Example Program cex9a . . . . .	11-11
Programming with TID Data Access . . . . .	11-18
Understanding TID Function Input and Output . . . . .	11-18
Using the TID Function in a Select List . . . . .	11-18
Using the TID Function in a WHERE Clause . . . . .	11-19
Declaring TID Host Variables . . . . .	11-19
Understanding the SQLTID Data Format . . . . .	11-19
Transaction Management with TID Access . . . . .	11-20



Comparing TID Access to Other Types of Data Access . . . . .	11-20
Verifying Data that is Accessed by TID . . . . .	11-21
Considering Interactive User Applications . . . . .	11-21
Coding Strategies . . . . .	11-22
Reducing Commit Overhead for Multiple Updates with TID Access . . . . .	11-23

**Index**

## Figures

---

1-1. Creating an ALLBASE/SQL C Application Program . . . . .	1-6
1-2. Skeleton ALLBASE/SQL C Program . . . . .	1-9
1-3. Components of a Stored Section . . . . .	1-12
1-4. Ways of Compiling and Linking an ALLBASE/SQL C Program . . . . .	1-14
1-5. Moving an Application to a Production System . . . . .	1-17
1-6. Use of DBCore Shared Memory in a DBEnvironment . . . . .	1-24
1-7. Runtime Dialog of Program cex2 . . . . .	1-30
1-8. Program cex2: Using Simple SELECT . . . . .	1-31
2-1. Compiling and Linking . . . . .	2-9
2-2. Full Preprocessing Mode Input and Output . . . . .	2-13
2-3. Sample sqlmsg Showing Errors . . . . .	2-17
2-4. Sample sqlmsg Showing Warning . . . . .	2-18
2-5. Information in SYSTEM.SECTION on Stored Sections . . . . .	2-21
2-6. Sample Modified Source File . . . . .	2-25
2-7. Sample Type Include File . . . . .	2-31
2-8. Sample Variable Include File . . . . .	2-33
2-9. Sample Externals Include File . . . . .	2-33
3-1. Host Variable Declarations . . . . .	3-7
3-2. Declaring Host Variables for Single-Row Query Results . . . . .	3-21
3-3. Declaring Host Variables for Multiple-Row Query Results . . . . .	3-22
3-4. Declaring Host Variables for Dynamic Commands . . . . .	3-27
3-5. Declaring Host Variables for Savepoint Numbers . . . . .	3-28
3-6. Declaring Host Variables for Message Catalog Messages . . . . .	3-28
3-7. Declaring Host Variables for DBEnvironment Names . . . . .	3-29
4-1. Program cex5: Implicit and Explicit Status Checking . . . . .	4-18
4-2. Explicit Status Checking Procedures . . . . .	4-25
4-3. Determining Number of Rows Processed After a BULK SELECT . . . . .	4-33
5-1. Flow Chart of Program cex7 . . . . .	5-13
5-2. Runtime Dialog of Program cex7 . . . . .	5-15
5-3. Program cex7: Using INSERT, UPDATE, SELECT and DELETE . . . . .	5-18
6-1. Cursor Operation without the KEEP CURSOR Feature . . . . .	6-11
6-2. Cursor Operation Using KEEP CURSOR WITH LOCKS . . . . .	6-12
6-3. Cursor Operation Using KEEP CURSOR WITH NOLOCKS . . . . .	6-13
6-4. Keep Cursor Application Program . . . . .	6-16
6-5. Flow Chart of Program cex8 . . . . .	6-27
6-6. Runtime Dialog of Program cex8 . . . . .	6-28
6-7. Program cex8: Using UPDATE WHERE CURRENT . . . . .	6-30
7-1. Flow Chart of Program cex9 . . . . .	7-16
7-2. Runtime Dialog of Program cex9 . . . . .	7-17
7-3. Program cex9: Using BULK INSERT . . . . .	7-20
8-1. Creation and Use of a Program that has a Stored Module . . . . .	8-3
8-2. Creation and Use of a Program that has no Stored Module . . . . .	8-4

8-3. Procedure Hosting Dynamic Non-Query Commands . . . . .	8-7
8-4. Dynamic Query Data Structures and Data Assignment . . . . .	8-9
8-5. Format of the Data Buffer for a Series 700 and 800 . . . . .	8-18
8-6. Parsing the Data Buffer in cex10a . . . . .	8-20
8-7. Flow Chart of Program cex10a . . . . .	8-26
8-8. Run Time Dialog of Program cex10a . . . . .	8-28
8-9. Program cex10a: Dynamic Commands of Unknown Format . . . . .	8-30
8-10. Flow Chart of Program cex10b . . . . .	8-44
8-11. Run Time Dialog of Program cex10b . . . . .	8-46
8-12. Program cex10b: Dynamic Queries of Known Format . . . . .	8-47
9-1. Constraints Enforced on the Recreation Database . . . . .	9-4
10-1. Flow of LONG Column Data and Related Information to the Database . .	10-3
10-2. Flow of LONG Column Data and Related Information from the Database .	10-3
11-1. Program cex9a: Using Date/Time Functions . . . . .	11-11
11-2. Using RC and RR Transactions with BULK SELECT, SELECT, and UPDATE . . . . .	11-22
11-3. Using TID Access to Reduce Commit Overhead . . . . .	11-24

## Tables

---

3-1. Data Type Declarations . . . . .	3-9
3-2. ALLBASE/SQL Floating Point Column Specifications . . . . .	3-10
3-3. C Data Type Equivalency and Compatibility . . . . .	3-19
3-4. Program Element Declarations . . . . .	3-25
4-1. sqlca Status Checking Fields . . . . .	4-5
8-1. SQLDA Fields . . . . .	8-12
8-2. Fields in a Format Array Record . . . . .	8-14
9-1. Commands Used with Integrity Constraints . . . . .	9-2
9-2. Constraint Test Matrix . . . . .	9-3
10-1. Commands You Can Use with LONG Columns . . . . .	10-1
10-2. LONG Column Descriptor . . . . .	10-11
11-1. Where to Use Date/Time Functions . . . . .	11-2
11-2. Host Variable Data Type Compatibility for Date/Time Functions . . . . .	11-3
11-3. Sample of User Requested Formats for Date/Time Data . . . . .	11-4
11-4. SQLTID Data Internal Format . . . . .	11-20

## Getting Started with ALLBASE/SQL Programming in C

---

ALLBASE/SQL is a relational database management system that uses SQL statements to access data within an ALLBASE/SQL DBEnvironment. Embedding SQL statements in your C program and preprocessing with the SQL preprocessor allows programmatic access to this data. Embedded statements are ALLBASE/SQL commands in the flow of what would otherwise be C source code. When embedded statements are present, the source code must be passed through an ALLBASE/SQL preprocessor to convert them to a form understood by the C compiler. This chapter examines the steps to follow as you begin to work with embedded SQL. It focuses on the programmer's tasks, which are presented in the following sections:

- Understanding ALLBASE/SQL Operations.
- Understanding the Program Life Cycle.
- Developing ALLBASE/SQL Applications.
- Moving Programs into Production.
- Maintaining ALLBASE/SQL Applications.
- Programming Under the HP-UX Operating System.
- Looking at an Embedded SQL Source Program

Embedded SQL programming is a simple process. You include SQL statements in a C source program, then you preprocess the code using the ALLBASE/SQL C preprocessor before compiling and linking. The preprocessor is fully described in Chapter 2, "Using the Preprocessor," and the specific techniques of ALLBASE/SQL programming are presented in more detail in the succeeding chapters. Readers who are already familiar with the general process of embedded SQL programming should skip ahead to chapter 2.

---

### Understanding ALLBASE/SQL Operations

You can incorporate almost any ALLBASE/SQL command in an application program. Some commands can only be used in applications; others require special varieties of syntax that are only used in applications. All the ALLBASE/SQL commands and the places where you can use them are described in the *ALLBASE/SQL Reference Manual*.

---

**Note**            You *cannot* use ISQL, SQLUtil, SQLGEN, or SQLMIGRATE commands programmatically. ISQL, SQLUtil, SQLGEN, and SQLMIGRATE are themselves independent applications.

---

There are several kinds of SQL operations you can perform from application programs:

- Accessing data using data manipulation language (DML).
- Creating ALLBASE/SQL objects using data definition language (DDL).
- Managing security with data control language (DCL).
- Handling transactions.
- Handling errors.
- Performing dynamic processing.

## Using DML to Manipulate Data

You use DML to examine or modify the rows in the tables in a database. The chief DML command for creating queries is `SELECT`. For other kinds of data manipulation you use `DELETE`, `INSERT`, or `UPDATE`. Whether for OLTP (online transaction processing) or report-writer applications, DML commands are the heart of database access. Therefore, much of the programmer's time is spent using DML to create the most efficient queries and updates. Most of this manual is devoted to the task of coding effective DML statements.

The following example of a simple data manipulation statement reads a row of data into some host variables (host variables are described below, under "Creating Source Files").

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

## Using DDL to Define Database Objects

You use DDL to create database objects, including tables, views, indexes, and authorization groups. For these tasks, you use the SQL `CREATE` commands, including `CREATE TABLE`, `CREATE VIEW`, `CREATE INDEX`, and `CREATE GROUP`. DDL is not always used in application programs, since database objects frequently exist before coding starts. However, it is sometimes useful to define objects at run time. Or you might wish to create special applications to tailor the creation of objects for the specific needs of your system. For detailed information about DDL functions, refer to the *ALLBASE/SQL Database Administration Guide*.

The following example of a data definition statement shows the creation of an index on one column of the `Parts` table:

```
EXEC SQL CREATE UNIQUE INDEX PartNumIndex
          ON PurchDB.Parts (PartNumber);
```

Data definition also includes the `DROP` commands for database objects such as the index created in the previous example, and the `ALTER TABLE` command for making changes to these objects. It also includes SQL statements for creating files and filesets for storage of database information.

## Using DCL to Manage Security

You use DCL to create a security scheme for your databases. Control statements are used to initiate access to a DBEnvironment and to provide security on specific database objects.

You may find it useful to assign permissions for newly created objects when the default permissions are not sufficient for the needs of your application. You can control the *ownership* of objects in ALLBASE/SQL DBEnvironments at creation time or through the TRANSFER OWNERSHIP command. Similarly, you can assign *authorization* for specific objects as you create them or through the GRANT and REVOKE commands.

Data control statements let you create groups, assign members to them, then grant all the authorities as needed. You can also grant and revoke authorities to individual users.

The following example of a data control statement assigns CONNECT authority to everyone for the current DBEnvironment:

```
EXEC SQL GRANT CONNECT TO ALL;
```

## Handling Transactions

Transactions are the units of work in an ALLBASE/SQL application. Transactions are usually delimited by BEGIN WORK and COMMIT WORK or ROLLBACK WORK statements. During a transaction, ALLBASE/SQL obtains locks on data and index pages; this can cause others to wait for data access. ALLBASE/SQL lets you regulate the kinds of locking your transactions use so as to achieve the greatest concurrency possible while protecting data from the actions of other users. The *ALLBASE/SQL Reference Manual* contains information about managing the transactions in your applications.

## Handling Errors

You use error handling to detect error conditions that arise while you are connected to an ALLBASE/SQL DBEnvironment. Error handling can be either implicit or explicit.

In implicit error handling, you use the WHENEVER SQLERROR and WHENEVER SQLWARNING statements to branch to a specific location following any error or warning. In this kind of processing, you do not test for a specific error condition.

In explicit error handling, it the programmer's responsibility to examine the specific error code returned by ALLBASE/SQL. For example, if the error number returned is -14024, ALLBASE/SQL has detected a deadlock and has rolled back a transaction. This means you can try the transaction again. On the other hand, an error of -2206 means that a table or view was not found. In this case, it does not make sense to reattempt the transaction.

Both implicit and explicit error handling techniques usually employ an error routine that incorporates the ALLBASE/SQL message command, SQLEXPLAIN. For example, a loop may be used to display to the user the text of each error or warning message related to a given SQL command.

Refer to the chapter, "Runtime Status and the sqlca," for complete details about error handling, including several examples.

## Dynamic and Non-Dynamic Operations

Whether you are using DML, DDL, or DCL operations, you can structure embedded SQL statements as either dynamic or non-dynamic. Commands are **non-dynamic** when the syntax of the entire command is known and preprocessed prior to run time. For many non-dynamic commands, ALLBASE/SQL can speed up database access by storing runtime instructions in the DBEnvironment at preprocessing time. The following is a non-dynamic query.

```
printf("Enter the Part Number: ");
gets(PartNumber);

EXEC SQL SELECT PartName, SalesPrice
        INTO :PartName, :SalesPrice
        WHERE PartNumber = :PartNumber;
```

Commands are **dynamic** when they are preprocessed at run time. ALLBASE/SQL converts these commands into executable instructions at run time rather than at preprocessing time. For example, you might know which table and which columns you want to query with a SELECT, but you may not know until run time the name of the DBEnvironment itself. In such a case, the preprocessor cannot store instructions in the DBEnvironment prior to run time.

The following is an example of a dynamic CONNECT in which the user enters the name of the DBEnvironment at run time:

```
printf("Enter the DBEnvironment Name: ");
gets(DBENAME);
sprintf(DynamicCommand,"CONNECT TO '%s';",DBENAME);
EXEC SQL EXECUTE IMMEDIATE :DynamicCommand;
```

There are many techniques for dynamic processing of queries and non-queries. The topic is discussed fully in the chapter, "Using Dynamic Operations."



---

## Understanding the Program Life Cycle

Each ALLBASE/SQL application program undergoes a number of phases in its useful life:

- **Development:** In the **development phase**, you create C source code procedures and integrate them into applications. Database access is provided through embedded SQL statements which are translated by a preprocessor into C procedures. After preprocessing the source, you compile, test, and debug, until the application is satisfactory. Then you install the application on a production system.
- **Production:** In the **production phase**, the application runs more or less continuously. The goals in this phase are to provide a high degree of concurrent access to data consistent with data security. The programmer can often enhance the performance of an application during production runs by using specific techniques as the code is developed. During production, the database administrator (DBA) can also help improve performance by monitoring the system's use of resources and by allocating additional disk space or shared memory as needed.
- **Maintenance:** In the **maintenance phase**, you modify the application in response to the changing needs of its users. This may mean updating the code, changing the security structure, or adding code to accommodate changes in the DBEnvironment. By careful coding to anticipate future developments in your databases, you can avoid the need to make extensive changes to production software.

---

## Developing ALLBASE/SQL Applications

The basic steps in developing ALLBASE/SQL applications are as follows:

- Create a source file.
- Preprocess the source file.
- Compile and link the program.

These steps, shown in Figure 1-1, are further described in the following paragraphs.

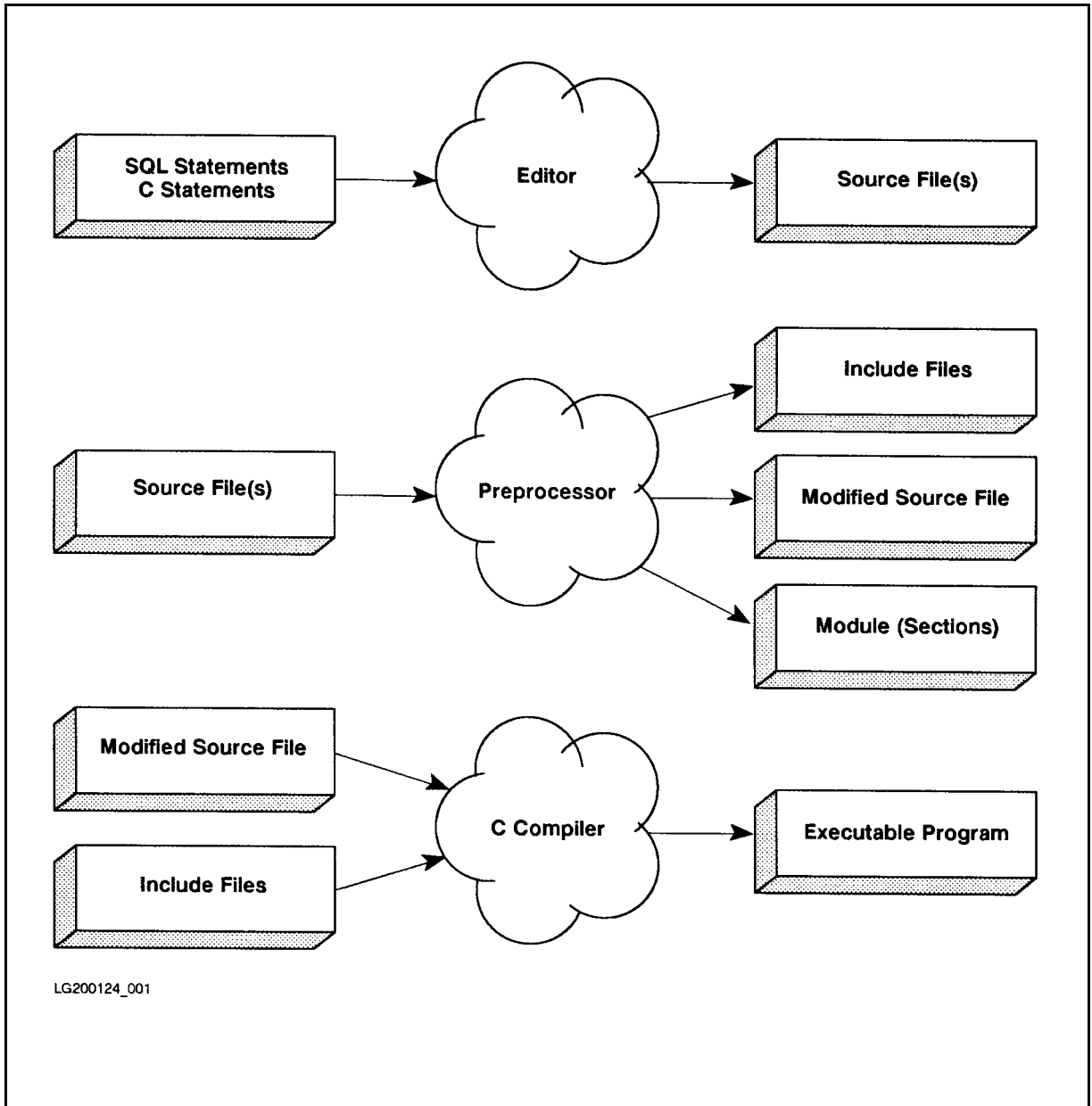


Figure 1-1. Creating an ALLBASE/SQL C Application Program

### Creating Source Files

Using an editor, you create one or more files containing C source code and SQL commands, which are said to be **embedded** in the program. You can use multiple source files, but each file containing embedded SQL statements must be separately preprocessed.

## Using Embedded SQL

The following SQL statements can be embedded in your program:

- Host variable declarations.
- INCLUDE SQLCA statements to let ALLBASE/SQL pass return codes to your program.
- A CONNECT command specifying the name of a DBEnvironment.
- BEGIN WORK and COMMIT WORK commands to delimit transactions (Note that if a transaction is not already started, *any* SQL command will automatically issue an implicit BEGIN WORK.).
- SELECT, INSERT, UPDATE, or DELETE commands; or cursor operations.
- Explicit or implicit error handling.

The *ALLBASE/SQL Reference Manual* describes the complete syntax for each kind of SQL statement.

Some SQL statements can only be used in application programs. These include the cursor commands (DECLARE, OPEN, FETCH, REFETCH, UPDATE WHERE CURRENT, DELETE WHERE CURRENT, and CLOSE); the bulk commands (BULK SELECT and BULK INSERT); and the error handling commands (WHENEVER and SQLEXPLAIN).

Moreover, some of the statements have a different effect when used in an application program than when used interactively. For example, the SELECT command in ISQL may be used when you wish to retrieve multiple rows, but in an application program a simple SELECT that retrieves more than one row results in an error.

## General Rules for Embedding SQL

ALLBASE/SQL C programs often follow a pattern where a main program segment calls functions that include the embedded SQL code.

### Declaring Special Data Structures

You must include the following instruction to the ALLBASE/SQL preprocessor in the global declarations area of each source file:

```
EXEC SQL INCLUDE SQLCA;
```

This incorporates into your program a data structure for handling return codes from ALLBASE/SQL, including numbered error conditions. You can use the information in the sqlca (SQL Communications Area) to test and branch. Refer to Chapter 4, “Runtime Status Checking and the sqlca,” for complete information about using the sqlca.

If your program includes dynamic query processing for the FETCH command with a USING DESCRIPTOR clause, add the INCLUDE SQLDA statement to your global declarations, and define the appropriate data buffer and format array. These elements are described fully in the chapter, “Using Dynamic Operations.”

All other SQL commands may appear in any part of your program.

## Declaring Host Variables

In addition to the normal variable declarations, the source file contains variable declarations for host variables. These can appear wherever declarations are legal. However, it is recommended that you include them at the beginning of the file, since they are translated into global variables by the preprocessor anyway. If more than one source file references the same set of variables, you must declare them separately in each file, and you must preprocess each file separately. See the chapter, “Using the Preprocessor,” for more details. Host variable declarations appear within a pair of SQL statements:

```
EXEC SQL BEGIN DECLARE SECTION;
.
.
EXEC SQL END DECLARE SECTION;
```

Host variables can appear both in the embedded SQL statements in your code and in ordinary C statements. When they appear in embedded SQL statements, you prefix them with a colon, as shown in this example:

```
EXEC SQL SELECT PartName, SalesPrice
        INTO :PartName, :SalesPrice
        FROM PurchDB.Parts;
```

Host variables are treated fully in Chapter 3.

## Skeleton Program

The skeleton program in Figure 1-2 illustrates the relationship between C constructs and embedded SQL commands in an application program. SQL commands may appear in the program at locations indicated by the comments.

```

.
/* STATIC VARIABLE DECLARATION PART */
sqlca Declaration
sqlda Declaration
.
/* Host Variable Declarations */
/* Host Variables can be global, local, or both */
.
/* main DECLARATION PART */
main(argc,argv)
/* Parameter Declaration */
{
.
/* Host Variable Declarations */
/* Host Variables can be local, or for called routines */

/* C statements, some containing SQL Commands */
EXEC SQL ...

int FunctionName();
.
}
/* FUNCTION DECLARATION PART */
FunctionName(parameter list)
/* Parameter Declaration */
{
.
/* Host Variable Declarations */
/* Host Variables can be local */

/* C statements, some containing SQL Commands */
EXEC SQL ...
.
}

```

**Figure 1-2. Skeleton ALLBASE/SQL C Program**

Most SQL commands appear within C functions in which you establish DBEnvironment access and manipulate data in a database. Variable declarations should follow rules for coding C programs without SQL statements. In addition to functions for various kinds of database access, you should code a CONNECT function, a RELEASE function, an error handling function, and transaction management functions containing BEGIN WORK and COMMIT WORK statements.

## Rules of Syntax for Embedded SQL Statements

You must follow some simple rules when embedding SQL statements in C code:

- Commands must be of appropriate size:
  - An embedded SQL command has no maximum length.
  - A dynamic SQL command within a host variable is limited only by the size of the host variable's declaration.
  - A dynamic SQL command not within a host variable can be no longer than 2048 bytes.
- Use EXEC SQL as the prefix to each SQL statement. The entire prefix, EXEC SQL, must appear on one line, as follows:

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

The following is *not* legal:

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

- Use a semicolon at the end of the SQL command, as shown above.
- C comments (those which begin with /\* and end with \*/) may appear within or between embedded SQL commands, as in the following:

```
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 */
```

- 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.;
```

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

- Non-numeric literals in embedded commands may be continued from one line to another.
- When referring to a host variable within an embedded SQL statement, precede it with a colon (:). Do not use the colon outside the embedded SQL statement.

A sample source file appears at the end of this chapter. And more detailed explanation of coding for different types of embedded SQL statements appears in Chapters 4 through 8.

## Preprocessing the Source File

After embedding SQL commands in the source code, *preprocess* it with the ALLBASE/SQL C *preprocessor*. Use the following command, which is described fully in the chapter, “Using the Preprocessor:”

```
$ psqlc DBEnvironmentName -i SourceFileName
```

This command includes ALLBASE/SQL DBEnvironment and source file name parameters. In addition to checking the syntax of your SQL statements, preprocessing also does the following:

- Creates a modified source file.
- Stores sections in the DBEnvironment.
- Generates a file, `sqlmsg`, for preprocessing messages. Creates an installable module file.

### Creating the Modified Source File

The preprocessor translates embedded SQL statements into C language statements and comments out the SQL statements. Non-SQL statements in your code are not translated. As output, the preprocessor creates a **modified source file** which can then be compiled. The original source file is not changed.

The preprocessor also creates three **include files**, which contain variable declarations, type declarations, and external procedure declarations used by the preprocessor generated C code. The modified source file contains include statements which direct the compiler to incorporate these include files at compile time.

### Creating Stored Sections

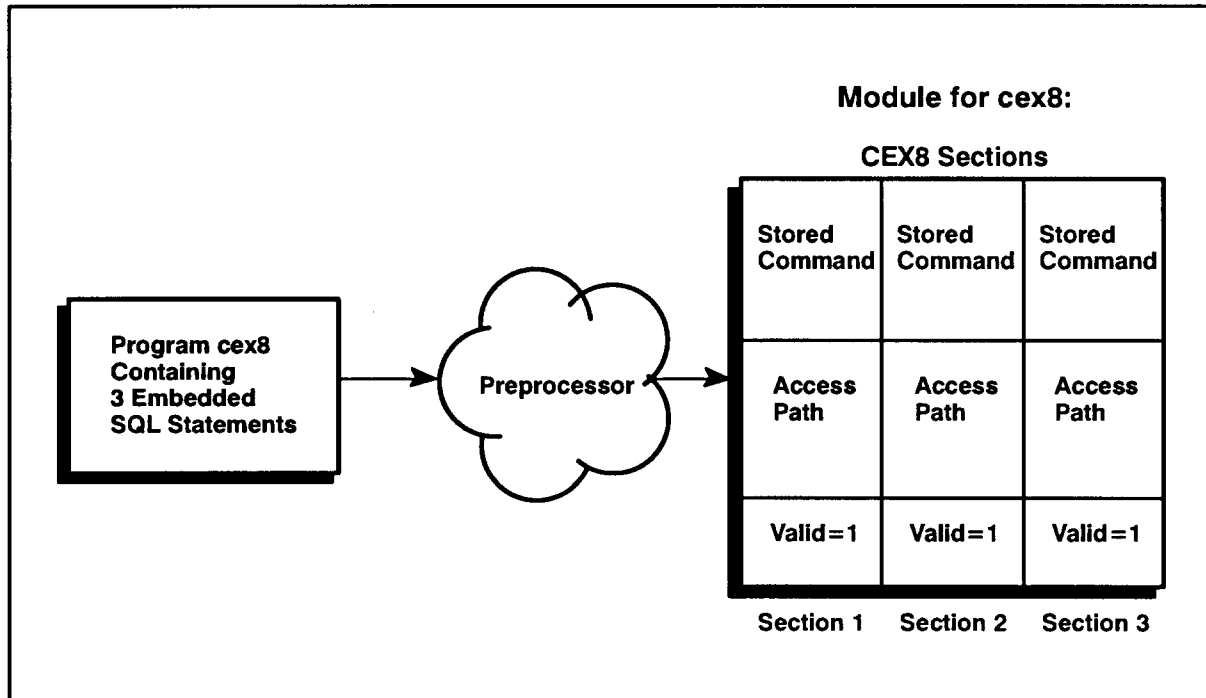
The preprocessor also stores runtime instructions in the system catalog of the DBEnvironment you specify when preprocessing. These instructions are called **sections** and are stored in a **module**, which contains a section for each DML, DDL, or DCL statement in your program that can be completely defined before run time. A module is referenced in the system catalog of the DBEnvironment.

Not all SQL statements cause the preprocessor to store a section. Only statements that access data cause a section to be stored.

A section has three parts:

- A stored form of the SQL command.
- Instructions for executing the command according to the best available access path.
- A flag indicating whether the section is *valid* or *invalid*.

These elements are illustrated in Figure 1-3.



LG200124\_002a

**Figure 1-3. Components of a Stored Section**

**Stored Form of the SQL Command.** In addition to translating each SQL statement into C code, the preprocessor stores a version of the statement in a section. This version of the statement is used at a later time if it becomes necessary to revalidate the section.

**Optimized Access Instructions.** ALLBASE/SQL also chooses the best available path for accessing the data referred to in the statement. This process is called **optimization**. For example, the following query can be optimized:

```
EXEC SQL SELECT * FROM PurchDB.Parts
WHERE PartNumber = :PartNumber;
```

ALLBASE/SQL first determines whether or not indexes exist on the PartNumber column. If there are indexes, ALLBASE/SQL then computes whether the use of an available index is more efficient than doing a serial scan of the entire table. The result of this decision whether or not to use the index (or which index to use, if there is more than one) is stored as part of the section for the query.

**Validity Flag.** The validity flag provides a way to ensure that any objects you reference in an SQL statement still exist and that runtime authorization criteria are satisfied. If the SQL command in a section 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**. 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 activities such 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.

In general, ALLBASE/SQL invalidates a section whenever there is a chance that the existing access path to the data might have changed. Suppose you drop an index on a column that appears in a query. In such a case, the section which contains that query and any other sections which reference that column are marked invalid.

**Runtime Revalidation of Sections.** At run time, ALLBASE/SQL checks each section for validity before executing it. When a section is found to be invalid, ALLBASE/SQL revalidates it to revalidate it (if possible), then executes it. You may notice a slight delay as the revalidation takes place. If you wish, you can re-preprocess the entire program to revalidate all sections at once, avoiding the delay of runtime revalidation. However, this is not required, since revalidation is done automatically and transparently.

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. The new owner can validate the section.

### **Generating the Message File**

During preprocessing, messages are written to the file sqlmsg. If preprocessing is successful, the file contains a statement indicating the name of the module stored in the DBEnvironment and the number of sections stored.

### **Compiling and Linking the Program**

Use the **C compiler** and **system linker** to create the **executable program** from the modified source code file and the include files. You can run both the compiler and the linker with the cc command, specifying the SQL link libraries on the command line. Alternatively, you can create object files by using the `-c` option of the cc command. In this case, you must link object(s) in a separate step. The second technique is used if you maintain your object modules separately. Figure 1-4 shows both techniques.

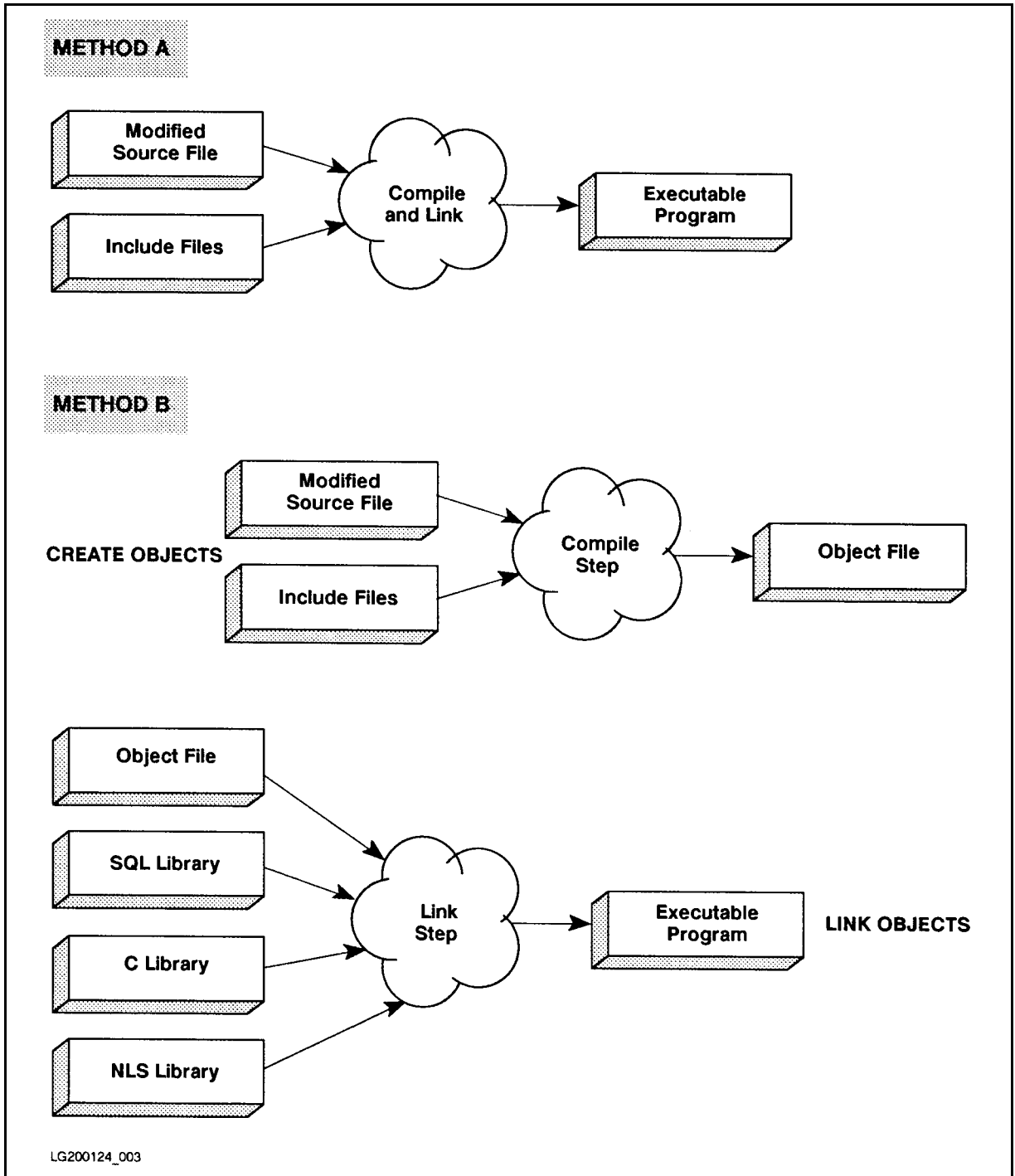


Figure 1-4. Ways of Compiling and Linking an ALLBASE/SQL C Program

**Caution**

When correcting your programs during the compile process, be sure to edit the *original* source file, not the preprocessor output file. Note that when it encounters an error, the compiler returns line numbers for the *modified* source file, so you must extrapolate from these to the actual lines in your original source. Edit the original source, re-preprocess, then re-compile and re-link.

## Running the Program

Once the preprocessing and compile steps have completed without error, you can run the application.

All the C 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. As your program runs, it executes code that calls each section that was previously stored. If the section is valid, it is then executed to carry out the query or other SQL operation.

If the section cannot be executed for any reason, ALLBASE/SQL returns an error code to the sqlca. Your program can examine this data structure and print out the text of messages that correspond to the error codes sent back by ALLBASE/SQL. Using a standard error handling routine makes debugging the embedded SQL a straightforward process. Refer to the chapter, “Runtime Status Checking and the sqlca,” for details and examples of incorporating error routines.

---

**Note** The error codes returned at run time are not the same as the errors shown in sqlmsg at preprocessing time. Runtime errors include such logical mistakes as issuing a BEGIN WORK when a transaction is already underway, OPENING a cursor that is already open, and so on.

---

If your program contains the SQLEXPLAIN command, you can display the text of an ALLBASE/SQL error message as an aid to debugging. SQLEXPLAIN obtains warning and error messages from the ALLBASE/SQL message catalog, which must be available at run time. 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 then uses the default catalog instead.

## Authorizations

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 the DBEnvironment your program accesses. When you preprocess a program, your login name becomes the owner of that module. *Subsequently*, only you or someone else with DBA authority can re-preprocess the program.

To access an ALLBASE/SQL DBEnvironment through a program, you need the authority to execute the command used in the program to start the DBE session:

- 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 original OWNER of the module has the authorization to execute the command at run time. However, any

dynamic command (an SQL command entered by the user at run time) is executed only if the login of the user running the program has the authority to execute the entered command.

Whoever runs the program must have either RUN authorization for the module or else be the OWNER or DBA. Granting authorizations is further described in Chapter 2 of this manual and in the *ALLBASE/SQL Database Administration Guide*.

## Debugging and Testing

As you test the program, use a set of database tables that resembles the production DBEnvironment as closely as possible. This will let you judge the performance of the program as well as exercising each of the segments of code. Remember that in debugging and testing the application, you are also testing the DBEnvironment's parameters. Elements such as log size, buffer size, maximum number of transactions, and other configurable parameters may need to be adjusted for the needs of the application. Use SQLUtil to adjust the parameters of the DBEnvironment. (DBA authority is required.)

After testing in single user mode, run tests with multiple users to observe the level of concurrency and the degree of throughput your application achieves. If relevant, observe the performance of the application while other applications are running. Refer to the *ALLBASE/SQL Reference Manual* for additional guidelines on coding for performance.

---

## Moving into the Production Phase

At the beginning of the production phase, you need to:

- Install the program module in the production DBEnvironment.
- Assign the appropriate security structure on the production system.

## Installing Program Modules

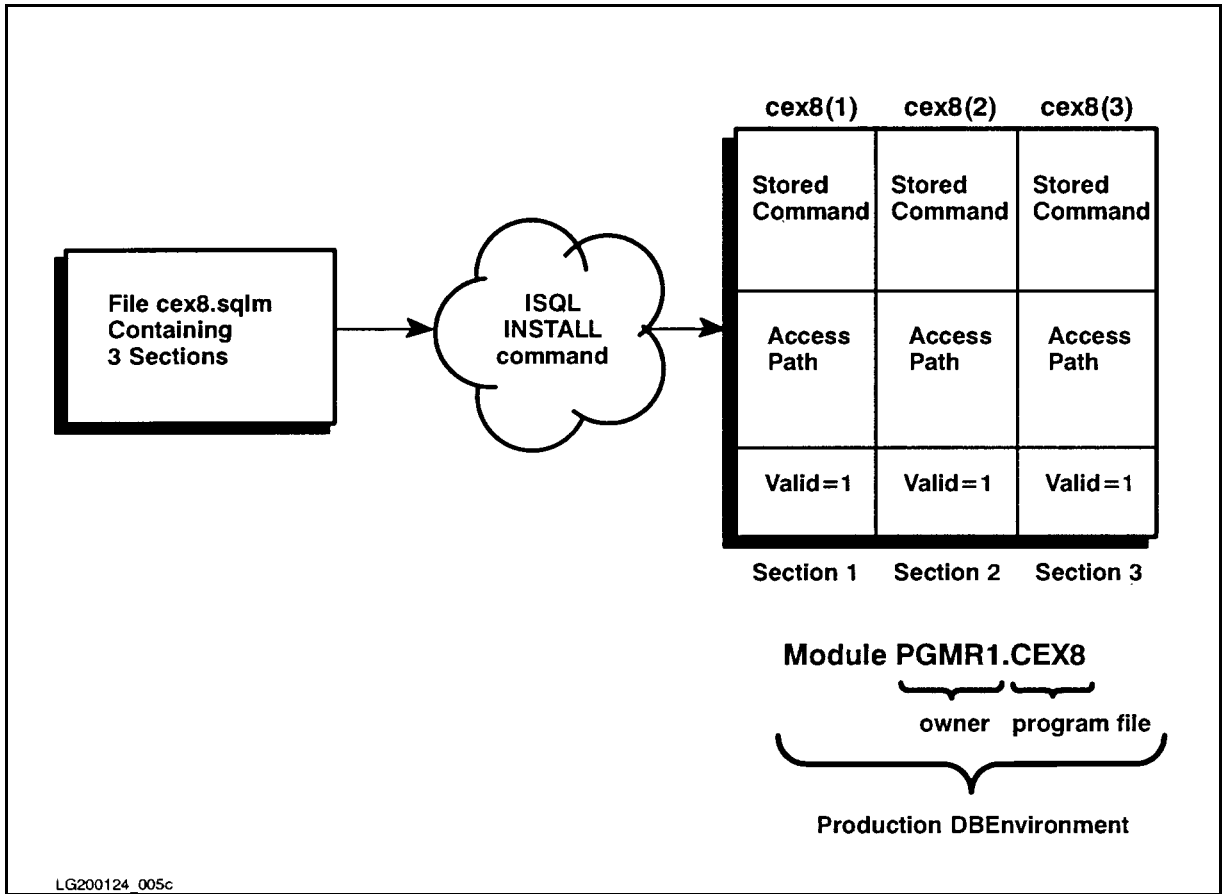
Installation involves using the ISQL INSTALL command to store a module created in one DBEnvironment into a different DBEnvironment on the same or a different system. When the preprocessor stores a module in a DBEnvironment, it also creates a file containing a copy of the module, which can be installed into another DBEnvironment. The installable module file in the following example is *SourceFileName.sqlm*. The module also has an internal, SQL name, in this case CEX8, which is saved as part of the module at preprocessing time. Use the INSTALL command in ISQL as shown in this example:

```
isql=> CONNECT TO 'somedbe';
isql=> INSTALL cex8.sqlm;

Name of module in this file: CEX8
Number of sections installed: 3
COMMIT WORK to save to DBEnvironment.

isql=> COMMIT WORK;
```

This process is illustrated in Figure 1-5.



**Figure 1-5. Moving an Application to a Production System**

ISQL copies the module from the installable module file named cex8.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 to be able to start a DBE session in the DBEnvironment that will 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. Before installing, you should DROP any existing module on the production machine that has the same name:

```
isql=> DROP MODULE CEX8;
```

## Granting Module Owner Authorizations

The original module owner is the DBEUserID of the person who preprocesses the program or, if specified, the DBEUserID used with the OWNER option in the preprocessor command line. 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 authority to a user in the production DBEnvironment with the same name as the original module owner in the development environment.

If module CEX8 contains a SELECT command for table PurchDB.Parts, the following grant would ensure valid owner authorization in the development environment:

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

Being the owner of the module, pgmr1 could have assigned ownership of the module to another owner at preprocessing time by using the -o option:

```
$ psqlc somedbe -o PurchMgrs -m cex8 -i cex8.sql -p cex8.c
```

In this case, ownership belongs to a group, PurchMgrs. Only members of the group or an individual with DBA authority can maintain this program, and runtime authorization would be established as follows:

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

---

### Note

Keep in mind that the original module owner's name is coded into the module itself. Therefore, the original DBEUserId must exist on the production system and possess the appropriate CONNECT and object authorities. As a rule, it is wise to assign ownership of modules to a group name which can be used on both development and production systems. In this fashion, you can assign different membership to the group on the production system than you assigned on the development system.

---

## Granting Program User Authorization

In order to execute an ALLBASE/SQL program, you must have CONNECT authority for any DBEnvironment accessed by 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;

login: somelogin
.
.
$ someprog.r
```

Refer to the *ALLBASE/SQL Reference Manual* for complete information on the GRANT command.

---

## Maintaining ALLBASE/SQL Applications

After ALLBASE/SQL C programs are in production use, changes in procedures, personnel, or databases may necessitate various changes in the application itself. Maintaining an ALLBASE/SQL program includes such activities as these:

- Tuning performance.
- Managing the source code.
- Updating a program in production use.
- Changing runtime authorizations as program users change.
- Obsoleting application programs.

For these activities, you need OWNER authority for the module or DBA authority.

### Tuning Performance

When an application enters production, you can estimate the best allocation of system resources, but only after it has run for a while can you fine-tune the performance. Refer to the *ALLBASE/SQL Performance Guidelines* for additional information.

The best performance also depends on good transaction management. Refer to the *ALLBASE/SQL Reference Manual* for additional discussion of transaction management.

## Managing Source Code

You should carefully maintain the source code for each application program in case it should be necessary to re-preprocess and re-compile at a later time. In addition to C source, it is recommended that you keep copies of the ISQL command files used to create your DBEnvironment. This information is extremely useful when modifying existing code.

Use SQLGEN to create schema files for storage once you have configured the DBEnvironment on the production side. Refer to the *ALLBASE/SQL Database Administration Guide* for information about using SQLGEN.

## Updating Application Programs

Minor modifications to programs in the production environment can be made while the program is not currently in use. Major program modifications, because they are more time-consuming, are usually made on a development machine in a 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 by using the DROP MODULE command in ISQL or can be specified 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:

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

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

You can also drop a module and revoke any existing run authorities for it at preprocessing time:

```
$ psqlc somebbe -m modulename -i mysource.sql -d -r
```

The -d option 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, and revokes any related RUN authorities. To revoke the RUN authorities by specifying the -r (revoke) option in the command string, you must also drop the module by specifying the -d (drop) option. The DROP MODULE command is also useful for 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, 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 change. Revoking CONNECT authority requires DBA authorization:

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

Revoking RUN authority requires either module OWNER or DBA authority:

```
isql=> REVOKE RUN ON PGMR1.SOMEPGM FROM OLDUSER;
```

## Dropping Obsolete Modules

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.

---

## Programming Under the HP-UX Operating System

You must take certain characteristics of the HP-UX operating system into account as you code ALLBASE/SQL C applications. These include:

- Access to Directories.
- Pathname Conventions in Referencing Files.
- Use of Shared Memory.
- Use of Interprocess Communication.
- Native Language Support.

### Access to Directories

It is best to create DBEnvironments in directories separate from the directory in which source files reside. For greatest security, the DBEnvironment should reside in a directory owned by user hpdb to which the programmer does not have write access. Program files should be kept in a separate directory to which the programmer has write access. Then you can code CONNECT statements like the following:

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

where sampledb is the directory containing the DBEnvironment and the two periods (..) are the parent directory of both the program directory and the DBEnvironment directory. Following this procedure assures the safety of DBEnvironment files.

### Pathname Conventions in Referencing Files

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

For example, if a DBEnvironment was created with the following command:

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

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

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

the pathname stored in the DBECon file would be ignored while creating this file. The user would need to fully qualify this pathname each time this file is referenced.

Both absolute and relative pathnames are restricted to a maximum length of 127 bytes (including the filename, e.g., OrdersDF).

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 qualify the pathname of that DBEnvironment.

For example, if the DBEnvironment you want the preprocessor to access resides in your `sampledb` subdirectory, you would invoke the preprocessor as follows:

```
$ psqlc sampledb/SomeDBE -i SourceFileName -d
```

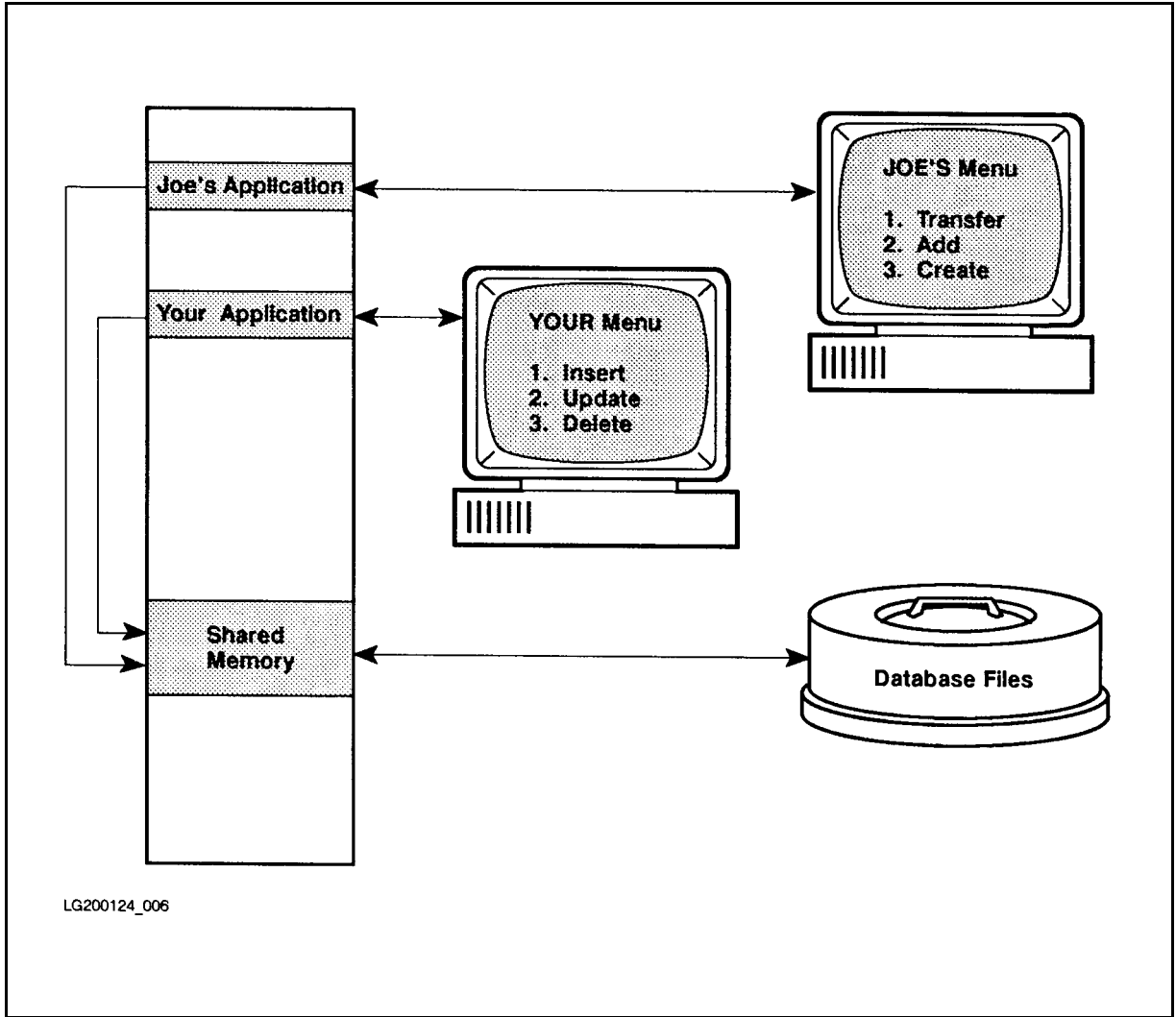
## Use of Shared Memory

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 last user 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 1-6.



**Figure 1-6. 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 ALTDDBE command. Refer to “Estimating 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 mapped to the user’s process.

### Series 300 Shared Memory

**300/400** The following shared memory considerations apply only to Series 300 and 400 systems.

Systems using SQL must be configured with a minimum shmmxaddr of 13MB. The default shmmxaddr is 16MB and is recommended. (See the *HP-UX System Administrator Manual* for information on shmmxaddr, 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 (SIGHUP)
- 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.

## Native Language Support

ALLBASE/SQL lets you manipulate databases in a wide 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 a wide variety of places, including:

- Character literals.
- Host variables for CHAR or VARCHAR data (but not variable names).
- ALLBASE/SQL object 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:

```
setenv LANG LanguageName
```

For the K shell, use the following command (no spaces before or after the equals 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 syntax, LANG = *LanguageName*, in the option of the START DBE NEW command to specify the language when you create a DBEnvironment.
4. If you are using a language with 16 bit character data to name database objects referenced in your program, enable NLS for the C compiler by using the -Y command line option.

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.

---

## Looking at an Embedded SQL Source Program

In every ALLBASE/SQL C program, you embed SQL commands in the declaration part and the procedure part of your program to carry out specific tasks. The program listing shown in Figure 1-8 illustrates where in a program you can embed SQL commands to accomplish the tasks listed below. Figure 1-7 shows the runtime dialog for the program.

- 1 Declare the SQL Communications Area (sqlca).  
The sqlca is an ALLBASE/SQL data structure that contains current information about a program's DBE session. Every ALLBASE/SQL C program must contain an sqlca declaration in the global declaration section. Use the following command:  

```
EXEC SQL INCLUDE SQLCA;
```
- 2 Declare host variables.  
All host variables used in a program must be declared in a declaration part. You can put more than one such declaration section in a program, but all host variables must be declared between the BEGIN and END DECLARE SECTION commands.
- 3 Display error and warning messages from the ALLBASE/SQL message catalog.



You can display messages for any errors encountered in execution as shown in the SQLStatusCheck function. Complete details about error and message handling are presented in the chapter, “Runtime Status Checking and the sqlca.”

4 Start a DBE session.

In most application programs, you embed the CONNECT command to start a DBE session. This command must be executed before you can access the DBEnvironment

5 Check the status of SQL command execution.

Your program should check for the success or failure of execution of each SQL command, as shown in the example program.

6 Terminate the DBE session.

You use the RELEASE command or the RELEASE option of the COMMIT WORK command to end a DBE session.

7 and Define transactions.

8 You define transactions in a program to control concurrency and consistency in your database access. The transaction is bounded by the BEGIN WORK and COMMIT WORK or ROLLBACK WORK commands. When a COMMIT WORK is successfully executed, all operations performed by the transaction it ends are permanently committed to the DBEnvironment

9 Define or manipulate data in the DBEnvironment.

Nearly all programs access data in one or more databases. The SELECT command shown in the example program retrieves the row from PurchDB.Parts that contains a part number matching the value in the host variable named in the WHERE clause. Note, indicator variables such as SalesPriceInd are discussed in the chapter, “Host Variables,” and data manipulation is presented fully in Chapters 5 through 8.

```
Program to SELECT specified rows from the Parts Table - cex2

Event List:
CONNECT TO ../sampledb/PartsDBE
BEGIN WORK
SELECT specified row from Parts Table
  until user enters a '/'
COMMIT WORK
RELEASE from ../sampledb/PartsDBE

Connect to ../sampledb/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 ../sampledb/PartsDBE
```

Figure 1-7. Runtime Dialog of Program cex2

```

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

typedef int boolean;

char    response[2];
boolean Abort;

#include <stdio.h>

#define     OK           0
#define     NotFound     100
#define     MultipleRows -10002
#define     DeadLock     -14024
#define     FALSE       0
#define     TRUE        1

EXEC SQL INCLUDE SQLCA;    /* SQL Communication Area */           1

                /* Begin Host Variable Declarations */
EXEC SQL BEGIN DECLARE SECTION;                                  2
char            PartNumber[17];
char            PartName[31];
double         SalesPrice;
sqlind         SalesPriceInd;
char           SQLMessage[133];
EXEC SQL END DECLARE SECTION;                                    2
                /* End Host Variable Declarations */

```

**Figure 1-8. Program cex2: Using Simple SELECT**

```

int SQLStatusCheck() /* Function to Display Error Messages */
{
Abort = FALSE;
if (sqlca.sqlcode < DeadLock)
    Abort = TRUE;
do {
    EXEC SQL SQLEXPLAIN :SQLMessage;
    printf("\n");
    printf("%s\n",SQLMessage);
    } while (sqlca.sqlcode != 0);

if (Abort) {
    EndTransaction();
    ReleaseDBE();
    }
} /* End SQLStatusCheck Function */

boolean ConnectDBE() /* Function to Connect to ../sampledb/PartsDBE */
{
boolean ConnectDBE;

ConnectDBE = TRUE;

printf("\n Connect to ../sampledb/PartsDBE";

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

if (sqlca.sqlcode != OK) {
    ConnectDBE = FALSE;
    SQLStatusCheck();
    } /* End if */
return (ConnectDBE);
} /* End of ConnectDBE Function */

int ReleaseDBE() /* Function to Release ../sampledb/PartsDBE */
{
printf("\n");
printf("\n Release ../sampledb/PartsDBE");
printf ("\n");
EXEC SQL RELEASE;

if (sqlca.sqlcode != OK) SQLStatusCheck();
} /* End ReleaseDBE Function */

```

Figure 1-8. Program cex2: Using Simple SELECT (page 2 of 5)

```

boolean BeginTransaction()    /* Function to Begin Work */
{
boolean BeginTransaction;
BeginTransaction = TRUE;

printf("\n");
printf("\n Begin Work");
EXEC SQL BEGIN WORK;

if (sqlca.sqlcode != OK) {
    BeginTransaction = FALSE;
    SQLStatusCheck();
    ReleaseDBE();
} /* End if */
return (BeginTransaction);
} /* End BeginTransaction Function */

int EndTransaction() /* Function to Commit Work */
{
printf("\n");
printf("\n Commit Work");
EXEC SQL COMMIT WORK;

if (sqlca.sqlcode != OK) SQLStatusCheck();
} /* End EndTransaction Function */

int DisplayRow() /* Function to Display Parts Table Rows */
{
printf("\n");
printf("\n Part Number:  %s\n",  PartNumber);
printf(" Part Name:      %s\n",  PartName);

if (SalesPriceInd < 0) {
    printf(" Sales Price:  is NULL \n");
}
else
    printf(" Sales Price: %10.2f\n", SalesPrice);
} /* End of DisplayRow */

```

Figure 1-8. Program cex2: Using Simple SELECT (page 3 of 5)

```

int Select()    /* Function to Query Parts Table */
{
do {
    printf("\n");
    printf("\n Enter Part Number within Parts Table or '/' to STOP > ");
    scanf("%s",PartNumber);

    if (PartNumber[0] != '/') {

        BeginTransaction();

        printf("\n SELECT PartNumber, PartName, SalesPrice");
        EXEC SQL SELECT PartNumber, PartName, SalesPrice          9
            INTO :PartNumber,
                :PartName,
                :SalesPrice :SalesPriceInd
            FROM  PurchDB.Parts
            WHERE PartNumber = :PartNumber;

        if ((sqlca.sqlwarn[0] == 'W') || (sqlca.sqlwarn[0] == 'w')) {
            printf("\n SQL WARNING has occurred. The following row");
            printf("\n    of data may not be valid!");
        }
        if (sqlca.sqlcode == OK) {
            DisplayRow();
        }
        else if (sqlca.sqlcode == NotFound) {
            printf("\n Row not found!");
        }
        else if (sqlca.sqlcode == MultipleRows) {
            printf("\n WARNING: More than one row qualifies!");
        }
        else {
            SQLStatusCheck();          5
        }
    }
    EndTransaction();
}
} /* End do */
while (PartNumber[0] != '/');

}/* End of Select Function */

```

Figure 1-8. Program cex2: Using Simple SELECT (page 4 of 5)

```

main()          /* Beginning of program */
{

printf("\n Program to SELECT specified rows from");
printf("\n  the Parts Table - cex2");
printf("\n");
printf("\n Event List:");
printf("\n  CONNECT TO ../sampledb/PartsDBE");
printf("\n  BEGIN WORK");
printf("\n  SELECT the specified row from the Parts Table");
printf("\n    until the user enters a '/'");
printf("\n  COMMIT WORK");
printf("\n  RELEASE from ../sampledb/PartsDBE");
printf("\n");

if (ConnectDBE()) {
    Select();
    ReleaseDBE();
}
else
    printf("\n Error: Cannot Connect to ../sampledb/PartsDBE!\n");
} /* End of Program */

```

**Figure 1-8. Program cex2: Using Simple SELECT (page 5 of 5)**





## Using the Preprocessor

---

This chapter shows how to use all the preprocessor's options, and it describes the inputs and outputs of the preprocessor command. Topics are:

- Invoking the C Preprocessor.
- Identifying Preprocessor Input.
- Identifying Preprocessor Output.
- Dealing with Preprocessor Errors.
- Looking at a Modified ALLBASE/SQL Source Program.
- Looking at Preprocessor Created Include Files.

An example of a modified source file appears at the end of the chapter along with listings of the preprocessor generated include files.

---

### Invoking the C Preprocessor

You can use the preprocessor in two modes:

1. **Full preprocessing mode:** 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.
2. **Syntax checking mode:** checks your SQL syntax without doing any other preprocessor tasks.

As you develop the SQL portions of your C 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.

Command syntax for both modes is presented below.

#### Full Preprocessing Mode

Use the following preprocessor command to:

- Check the embedded SQL command syntax.
- Create compilable output files that can be processed by the C compiler
- Store a module in the DBEnvironment named.
- Create a file containing an installable version of the module.

## Syntax

```
psqlc DBEnvironmentName  $\left[ \begin{array}{l} -i \text{ SourceFileName} \\ -p \text{ ModifiedSourceFileName} \\ -o \text{ OwnerName} \\ -m \text{ ModuleName} \\ -d [-r] \end{array} \right] | \dots |$ 
```

## 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 ModifiedSourceFileName* identifies the name for 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 *.c*.
- o OwnerName* associates the stored module with a DBEUserID, a class name, or a group name. You can specify an owner name for the module if you have DBA authority in the DBEnvironment where the module is created or if you are the current owner of the module. If not specified, the owner name is your DBEUserID. Any object names in *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 (by means of ISQL or an embedded SQL program). Remember that if multiple, simultaneous users run a program, you should issue the `START DBE` command only once.
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 space management, such as using the `START DBE NEWLOG` command to increase the size of the log file and recovering log space when *rollforward* logging is in effect.
6. During preprocessing, system catalog pages accessed for embedded commands are locked. In multiuser mode, other DBE sessions accessing the same objects must wait, so there is potential for a deadlock. Therefore, minimize competing transactions if possible when preprocessing an application program. Refer to the appendix, "Locks Held on the System Catalog by SQL Commands," in the *ALLBASE/SQL Database Administration Guide* for information on operations that lock system catalog pages.
7. For improved runtime performance, use the `UPDATE STATISTICS` command in ISQL *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 n-computer (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, you need DBA or CONNECT authority for the DBEnvironment specified in the preprocessor command line. You also need table and view authorities for the tables and views which the program will access at run time.

DBEnvironment CONNECT authority can also be explicitly granted. If you have DBECreator or DBA authority or module OWNER authority, you have CONNECT authority by default.

Table authorities are implicitly specified at the time the table is created and depend on the table type (PUBLIC, PUBLICREAD, or PRIVATE). Once a table has been created, its implicit authorities can be changed by the table OWNER, the DBECreator, or another DBA. Table authorities are removed by using the REVOKE command and are added by using the GRANT command.

For example, for a PUBLIC table, you are implicitly granted authority for any type of table access when the table is created. For a PUBLICREAD table, you must have explicitly granted authority for any table access except READ access which is an implicit grant. For a PRIVATE table, there are no implicit grants at table creation time; only the table OWNER or a DBA can access a PRIVATE table, unless specific authorities are granted to others.

Note, in the case of the sample database, PartsDBE, the creation script REVOKES all implicit table authorities, and desired authorities must be explicitly granted.

---

**Note**            When preprocessing, you cannot name another user as module owner unless you are a DBA of the DBEnvironment or you are the current module owner.

---

## Example

```
MON, JUL 10, 1991, 4:48 PM
$ psqlc ../sampledb/PartsDBE -o joann -m joannpgm -i prog1.sql -d -r
HP36217-02A.E1.00          C 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          C 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           = JOANNPGM
```

```
***** SELECT PartNumber, PartName, SalesPrice INTO :PartNumber, :PartName,
          :SalesPrice WHERE PartNumber = :PartNumber
***** ALLBASE/SQL warnings. (DBERR 10602)          |
***** User joann does not have SELECT authority on PurchDB.Parts
(DBERR 2301)
1 Sections stored in DBEnvironment.
```

```
0 ERRORS 1 WARNINGS
END OF PROCESSING
```

```
$
```

## Syntax Checking Mode

The following command only checks the syntax of the SQL commands embedded in the source code file.

Syntax

```
psqlc -s [ -i SourceFileName
          -p ModifiedSourceFileName ] | ... |
```

## Parameters

- s causes the preprocessor to only check SQL syntax.
- i *SourceFileName* identifies the name of the input file containing the source code to be preprocessed.
- p *ModifiedSourceFileName* identifies the name for 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*.

## Description

1. The preprocessor does not access a DBEnvironment when it is run in this mode.
2. When performing syntax only checking, the preprocessor does not convert the SQL commands into C 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 are incomplete.

## Authorization

You do not need ALLBASE/SQL authorization when you use the preprocessor to only check SQL syntax.

## Example

```
$ psqlc -s -i mysource
```

```
MON, JUL 10, 1991, 4:48 PM  
HP36217-02A.E1.00 C 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 C 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 :PartNameInd,  
:SalesPrice :SalesPriceInd FROM PurchDB.Parts  
WHERE PartNumber = :PartNumber;
```

```
***** ALLBASE/SQL errors. (DBERR 10977)  
***** in SQL statement ending in line 128  
*** <1001> Syntax error. (DBERR 1001)
```

```
Syntax checked.  
  
1 ERRORS 0 WARNINGS  
END OF PREPROCESSING
```

*The line 128 referenced in sqlmsg is the line in mysource where the erroneous SQL command ends.*

## DBEnvironment Access

When you invoke the preprocessor in full preprocessing mode, 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.

```
$ psqlc DBEnvironment -m ModuleName  
    -i SourceFileName.sql -p ModifiedSourceFileName.c
```

When the preprocessor terminates its DBEnvironment session, it issues a COMMIT WORK command if it encountered no errors. Created sections are stored in the DBEnvironment and associated with the module name. See Figure 2-6 later in this chapter.

ALLBASE/SQL accesses the DBEnvironment you specify during preprocessing, even if your program does not use SQL statements that store sections in this DBEnvironment. Therefore, you must specify the name of a valid DBEnvironment.

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, use ISQL to install the installable module created by the preprocessor into each additional DBEnvironment accessed by your program. See the section “Installable Module File” in this chapter.

## Running the Preprocessor in the Background

You can run the C preprocessor in the background to free your terminal for performing other processes. In the following example, the C preprocessor is run in the background in the C shell to check the syntax of a program, and any messages usually sent to the terminal are redirected to an output file:

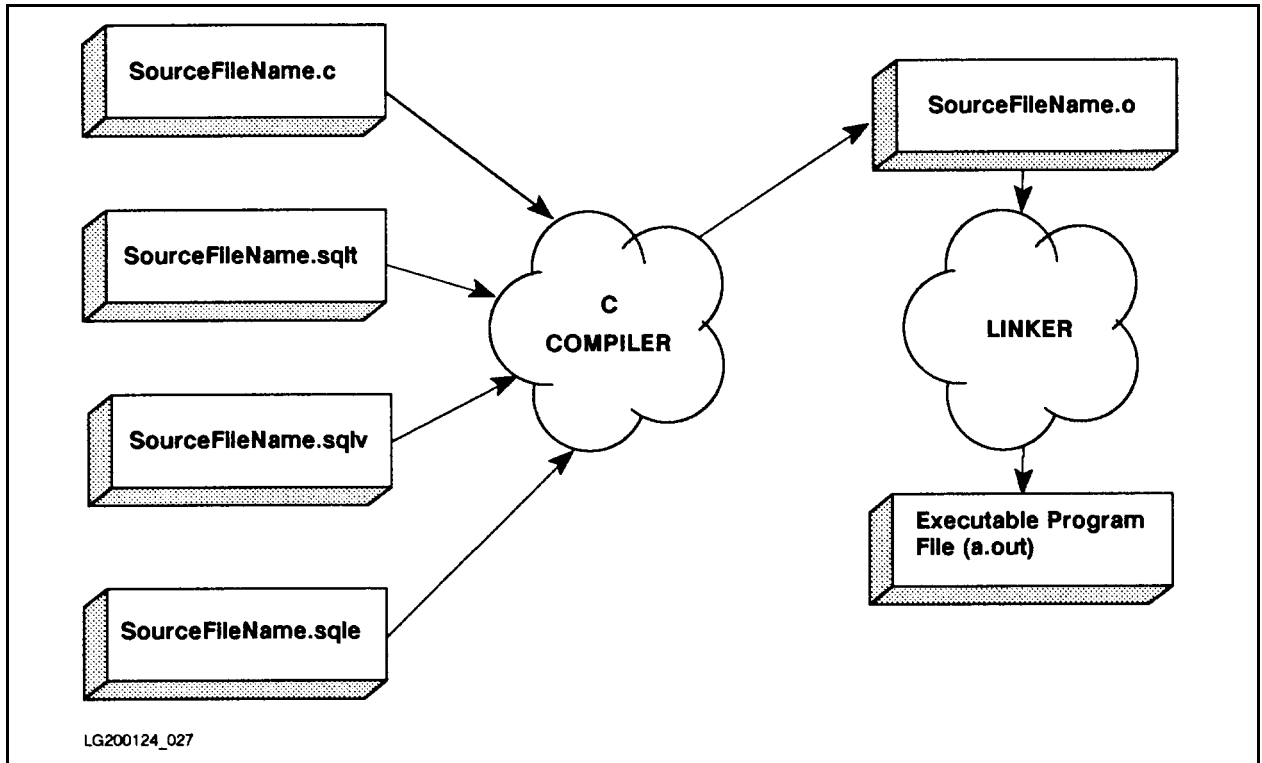
```
$ psqlc -s -i myprogram >& myprogram.out &
```

The syntax for this statement may not be the same if the command is not entered in the C shell.

## Compiling and Linking

Figure 2-1 shows the process of compiling and linking an embedded SQL C program.





**Figure 2-1. Compiling and Linking**

As shown in the figure, you submit to the C compiler one or more modified source code files and the related include files created by the preprocessor. The compiler then generates object code. By default, any object code file name consists of the source code file name with a .o file extension.

To convert these object code files into an executable program, link them after compilation by invoking the link editor. This step creates an executable program file.

The linker is automatically invoked after compilation unless the `-c` compile only option is used. If you want to link multiple object code modules together to form one executable program, list the object code modules following the C compiler command, followed by the library options.

If each of the object code modules has a .o file extension, the module will not be recompiled. If you do not use the `-o` compiler option to specify a specific output file name for the linked output, a file named a.out is created by default.

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

```
login: pgmr1
:
$ psqlc somedbe -m somemod -i mysource.sql -p mysource.c
:
$ cc mysource.c -lsql -lcl -lportnls -o someprog.r
```

The options used are:

- `-lsql`, to link the ALLBASE/SQL runtime library, in `/usr/lib/libsql.a`
- `-lcl`, to link the Pascal runtime library, in `/usr/lib/libcl.a`
- `-lportnls`, to link the native language routines, in `/usr/lib/libportnls.a`
- `-o`, to create the executable output file `someprog.r`

---

```
300/400      login: pgmr1
:
$ psqlc somedbe -m somemod -i mysource.sql -p mysource.c
:
$ cc mysource.c -lsql -lheap2 -lportnls -lpc -o someprog.r
```

The options used are:

- `-lsql`, to link the ALLBASE/SQL runtime library, in `/usr/lib/libsql.a`
- `-lheap2`, to link the random heap space management routines, in `/usr/lib/libheap2.a`
- `-lportnls`, to link the native language routines, in `/usr/lib/libprotnls.a`
- `-lpc`, to link the Pascal runtime library, in `/usr/lib/libpc.a`
- `-o`, to create the executable output file `someprog.r`

---

## Running the Program

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 the module owner:

```
$ someprog.r
```

To make the program executable by other users in other DBEnvironments:

- 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.

## Accessing Multiple DBEnvironments

An alternative method of accessing more than one DBEnvironment from the same program would be to divide the program into separate compilable files. Each source file would access a DBEnvironment. In each file, start and terminate a DBE session for the DBEnvironment accessed. Then preprocess and compile each file separately. When you invoke the preprocessor, identify the DBEnvironment accessed by the source file being preprocessed. After a file is preprocessed, it must be compiled so that no linking is performed before the next source file is preprocessed. When all source files have been preprocessed and compiled, link them together to create an executable program. An example of this technique follows:

```
$ psqlc DBEnvironment1 -i SourceFileName1.sql -p ModifiedSourceFileName1.c
.
.
$ cc -c ModifiedSourceFileName1.c
.
.
$ psqlc DBEnvironment2 -i SourceFileName2.sql -p ModifiedSourceFileName2.c
.
.
$ cc -c ModifiedSourceFileName2.c
.
.

$ cc ModifiedSourceFileName1.o ModifiedSourceFileName2.o -lsql -lcl -lportnls
-o ExecutableFileName
```

---

### 300/400

```
$ cc ModifiedSourceFileName1.o ModifiedSourceFileName2.o -lsql -lheap2 -lportnls
-lpc -o ExecutableFileName
```

---

Note that a program which accesses more than one DBEnvironment must do so in sequence since only one DBEnvironment can be accessed at a time. Such program design may adversely affect performance and requires special consideration.

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

---

## Identifying Preprocessor Input

In the simplest case, illustrated earlier in Figure 1-1, the ALLBASE/SQL C program consists of one source code file and, optionally, one or more user include files; both the source code and the include files can contain SQL commands. The preprocessor merges any include files into the source program, and preprocesses it. The resulting modified source code file is then compiled and linked in the same manner as a C program not containing embedded SQL statements. (Include files must either exist in the current directory or be specified with a full path name.)

Regardless of the preprocessing mode you use, the source file and the ALLBASE/SQL message catalog must be available when you invoke the C preprocessor, as shown in Figure 2-2.

### Source File

The **source file** is a file containing the source code of the C ALLBASE/SQL program with embedded SQL commands for one or more DBEnvironments.

An alternative name can be specified by using the DROP option in the preprocessor command line, as explained earlier in this chapter. You must have proper table access authority to drop a module.

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

- The C compiler directive, include.
- The source file name. Unless you specify a module name in the preprocessor command line, the preprocessor uses the source file name as the name for the module it stores. A module name can contain as many as 8 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 semicolon (;) suffix. These constructs follow the rules given in Chapter 1 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 C data declarations for the host variables used in the program. Host variables are described in the chapter, “Host Variables.”

### ALLBASE/SQL Message Catalog

The **ALLBASE/SQL message catalog**, contains preprocessor messages and ALLBASE/SQL error and warning messages. The full pathname for the default message 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 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.

## 2-12 Using the Preprocessor

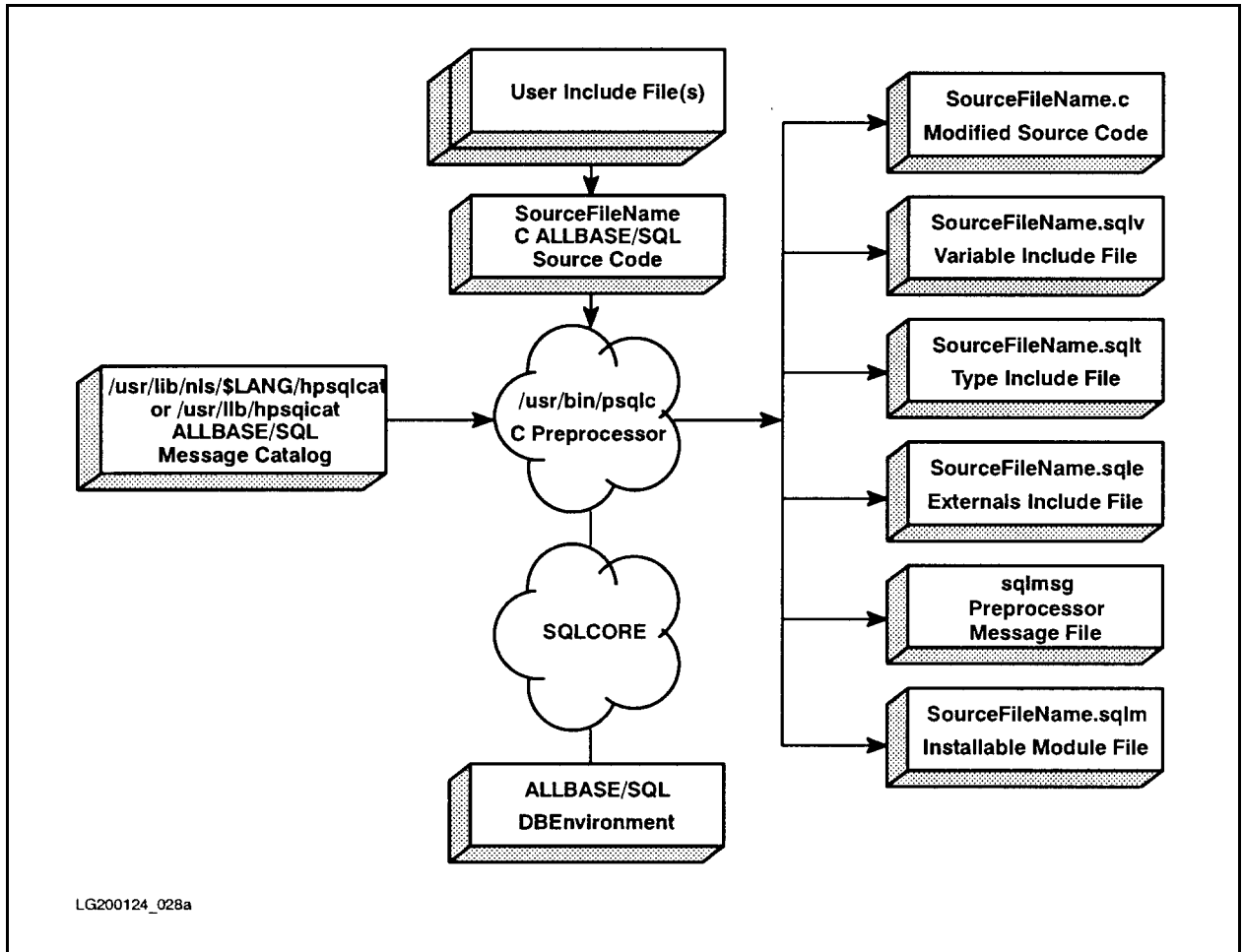


Figure 2-2. Full Preprocessing Mode Input and Output

## Identifying Preprocessor Output

As Figure 2-2 points out, running the C preprocessor in full preprocessing mode creates the following output files:

- Modified source file.
- Include files.
- SQL message file.
- Installable module file.

Also, a module is stored in the DBEnvironment specified in the preprocessor invocation line. Each of these is described in the following sections.

## Modified Source File

As the C preprocessor parses the source file, it copies lines from it and any include files into the modified source file, inserts conditional compiler directives around the embedded SQL commands, and inserts information around each embedded SQL command.

The default modified source file name is SourceFileName.c. You can specify a modified source file name by using the -p option of the preprocessor command line.

Figure 2-6 illustrates the modified source file generated for the source file pictured in Figure 1-8. The shaded lines contain information generated by the C preprocessor.

In both preprocessing modes, the C preprocessor:

- Inserts conditional compiler directives around embedded SQL commands to conditionally ignore the SQL commands.
- Inserts three #include C compiler directives within the Declaration part. These directives reference the three preprocessor generated include files. These files are included at the beginning of the modified source file created by the preprocessor.
- Places comments on the line with an embedded command following the last line the embedded command generates. Note, for example, that the comment following the EXEC SQL INCLUDE SQLCA command in the source file is in the same column, but on a different line, in the modified source file.

In full preprocessing mode, the preprocessor also:

- Generates a C declaration of the sqlca and the sqlda in the type include file.
- Generates C statements providing conditional instructions following SQL commands encountered after one of the following SQL commands: WHENEVER SQLERROR, WHENEVER SQLWARNING, and WHENEVER NOT FOUND.
- Generates C statements that call ALLBASE/SQL external procedures at run time. 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 and type include files.

---

### Caution

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

---

If you change non-preprocessor-generated constructs, make the changes to the source file, re-preprocess it, and re-compile the output files before putting the application program into production.

## Include Files

There are three include files, which contain declarations and definitions used by the C functions created by the preprocessor and inserted into the modified source code file:

- Type include file: the name for this file, which contains type declarations, is of the following syntax:

*ModifiedSourceFileName.sqlt*

- Variable include file: the name for this file, which contains variable declarations, is:

*ModifiedSourceFileName.sqlv*

- Externals include file: the name for this file, which contains external procedure declarations, is:

*ModifiedSourceFileName.sqle*

The `sqlmodulename` and `ownername` of sections are defined as global static variables in the `.sqlv` include file. It is advised that multiple source file applications be preprocessed all at one time so that only one `.sqlv` file is generated. If the source files are preprocessed and compiled separately and combined at link time, runtime errors occur because the static global `sqlmodulename` variable from the first source file preprocessed is used and incorrect sections are executed. Therefore, all preprocessed sections in your program must reside in the same file for input to the C compiler.

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

## ALLBASE/SQL Message File

The ALLBASE/SQL message file is named `sqlmsg`. Messages generated by the preprocessor in `sqlmsg` when you preprocess 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 four parts:

1. A banner, a sample of which is shown below:

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

---

### 300/400

```
MON, JUL 10, 1991, 4:48 PM
HP79725A.E1.00          C 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            = CEX2
```

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 query processing 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 2-3, a **line number** is often provided in sqlmsg. This line number references the line in the program 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                   C 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       = CEX2

      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 query processing 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.
  2 ERRORS   0 WARNINGS
END OF PROCESSING.
$
```

**Figure 2-3. Sample sqlmsg Showing Errors**

As Figure 2-4 illustrates, the preprocessor can terminate with the warning message

```
***** ALLBASE/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          C 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           = CEX2

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

***** ALLBASE/SQL warnings. (DBWARN 10602)
***** in SQL statement ending in line 128
*** Column PARNUMBER not found. (DBERR 2211)

      1 Sections stored in DBEnvironment.

      0 ERRORS    1 WARNINGS
      END OF PREPROCESSING
```

**Figure 2-4. Sample sqlmsg Showing Warning**

## Stored Module Containing Sections

In full preprocessing mode, the preprocessor stores a module in the DBEnvironment you specify at preprocessing time. By default, the preprocessor uses the source file name as the name for the module it stores. You can specify a module name with the MODULE option of the preprocessor command.

The module contains a section for each embedded SQL command in your program *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 are known as an input tree and a run tree. The **input tree** consists of the 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 section 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.

There are 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 2-5 illustrates the kind of information in the system catalog. All stored sections for each module in the DBEnvironment are referenced here. The query result illustrated was extracted from the system view named SYSTEM.SECTION by using ISQL. The columns in Figure 2-5 have the following meanings:

- **NAME:** This column contains the name of the module to which a section belongs. You can specify a module name when you invoke the preprocessor; or the module name will default to the source code file name of the C 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 can specify an owner name when you invoke the preprocessor, or the owner name will default to the login name 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 which contains the DBEFile(s) which in turn contains the section(s).
- **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
CEX2	JOANN	SYSTEM		1  0	1
CEX7	JOANN	SYSTEM		1  1	1
CEX7	JOANN	SYSTEM		2  0	1

```
-----
```

Figure 2-5. Information in SYSTEM.SECTION on Stored Sections

The first eighteen rows in the query result shown in Figure 2-5 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 cex2 contains only one section, for executing the SELECT command in the program illustrated in Figure 2-6. Program cex7 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 DBEnvironment until they are deleted by using the ISQL DROP MODULE command:

```
isql=> DROP MODULE cex2;
```

or by invoking the preprocessor with the -d option:

```
$ psqlc ../sampledb/PartsDBE -i cex2 -d
```

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

Stored sections are marked invalid when:

- 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 are made 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 re-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.

## Installable Module File

When the preprocessor stores a module in the DBEnvironment you named at preprocessing time, it places a copy of the module in an installable module file. 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. In order to install the module, you need `CONNECT` or `DBA` authority in the target DBEnvironment.

The installable module file is named `SQLMOD`. The module also has an internal, SQL name, in this case `JOANN.CEX2`, which is saved as part of the module at preprocessing time. `JOANN` is the login of the user who preprocessed source file `cex2`, and `JOANN` is the owner of the module. Note that `ALLBASE/SQL` shifts the system login and installable module file names to upper case to form the internal, installable module name.

```
$ isql

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

File name> cex2.sqlm;
Name of module in this file:  JOANN.CEX2
Number of sections installed:  1
COMMIT WORK to save to DBEnvironment.

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

---

## Handling Preprocessor Errors

Several types of errors can occur while you are using the C preprocessor:

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

### Preprocessor or DBEnvironment Termination

Whenever the 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 source file specified is currently being accessed, or if the source file named cannot be found, preprocessing terminates with the following messages, respectively:

```
System error in opening input source file. (DBERR 10922)
```

```
File open Error (DBERR 10907)
```

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

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

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

## Source File Errors

When the preprocessor encounters errors while 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 result because:

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

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



---

## Sample Modified Source File

The following modified source file is the result of preprocessing program `cex2` (shown at the end of Chapter 1). In the listing, the parts of the code are shaded for easy reference, as follows:

- Conditional compiler directives that tell the compiler to ignore the original SQL commands.
- Begin and end braces that indicate the preprocessor-generated code.

```
#include "cex2.sqlt"
#include "cex2.sqlv"
#include "cex2.sqle"
/* Program cex2 */

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

typedef int boolean;

char    response[2];
boolean Abort;

#include <stdio.h>

#define    OK                0
#define    NotFound         100
#define    MultipleRows    -10002
#define    DeadLock        -14024
#define    FALSE           0
#define    TRUE             1

#if 0
EXEC SQL INCLUDE SQLCA;
#endif
sqlca_type sqlca;
```

Figure 2-6. Sample Modified Source File

```

/* SQL Communication Area */

/* Begin Host Variable Declarations */
#if 0
EXEC SQL BEGIN DECLARE SECTION;
#endif

char      PartNumber[17];
char      PartName[31];
double    SalesPrice;
sqlind    SalesPriceInd;
char      SQLMessage[133];

#if 0
EXEC SQL END DECLARE SECTION;
#endif

/* End Host Variable Declarations */

int SQLStatusCheck() /* Function to Display Error Messages */
{
Abort = FALSE;
if (sqlca.sqlcode < DeadLock)
    Abort = TRUE;
do {

#if 0
EXEC SQL SQLEXPLAIN :SQLMessage;
#endif
{
sqlxplnc(&sqlca,&sqltempv,133,1);
sqltempv.rec1.
    SQLREC1_FIELD1[sqltempv.rec1.SQLREC1_FIELD1_LEN] = '\0';
strcpy(SQLMessage,
        sqltempv.rec1.SQLREC1_FIELD1);
}
printf("\n");
printf("%s\n",SQLMessage);
} while (sqlca.sqlcode != 0);

if (Abort)
{
EndTransaction();
ReleaseDBE();
}
} /* End SQLStatusCheck Function */

```

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



```

printf("\n");
printf("\n Begin Work");
#if 0
EXEC SQL BEGIN WORK;
#endif
{
sqlvar3 = "00A6007F00110061";
sqlxconc(&sqlca,sqlvar3);
}

if (sqlca.sqlcode != 0K)
{
BeginTransaction = FALSE;
SQLStatusCheck();
ReleaseDBE();
} /* End if */
return (BeginTransaction);
} /* End BeginTransaction Function */

int EndTransaction() /* Function to Commit Work */
{
printf("\n");
printf("\n Commit Work");

#if 0
EXEC SQL COMMIT WORK;
#endif
{
sqlvar4 = "00A10000";
sqlxconc(&sqlca,sqlvar4);
}

if (sqlca.sqlcode != 0K) SQLStatusCheck();
} /* End EndTransaction Function */

int DisplayRow() /* Function to Display Parts Table Rows */
{
printf("\n");
printf("\n Part Number: %s\n", PartNumber);
printf(" Part Name: %s\n", PartName);

if (SalesPriceInd < 0) {
printf(" Sales Price: is NULL \n");
}
else
printf(" Sales Price: %10.2f\n", SalesPrice);
} /* End of DisplayRow */

```

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

```

int Select()      /* Function to Query Parts Table */
{

do {
    printf("\n");
    printf("\n Enter Part Number within Parts Table or '/' to STOP > ");
    scanf("%s",PartNumber);

    if (PartNumber[0] != '/') {

        BeginTransaction();

        printf("\n SELECT PartNumber, PartName, SalesPrice");

#ifdef 0
        EXEC SQL SELECT PartNumber, PartName, SalesPrice
            INTO :PartNumber,
                :PartName,
                :SalesPrice :SalesPriceInd
        FROM   PurchDB.Parts
        WHERE  PartNumber = :PartNumber;
#endif
    {
        sqltempv.rec2.SQLREC2_FIELD1_LEN =
            strlen(PartNumber);
        strcpy(sqltempv.rec2.SQLREC2_FIELD1,
            PartNumber);
        sqlxfetc(&sqlca,sqlowner,sqlmodname,1,&sqltempv,24,80,1);
        if (sqlca.sqlcode == 0)
        {
            sqltempv.rec3.
                SQLREC3_FIELD1[sqltempv.rec3.SQLREC3_FIELD1_LEN] = '\0';
            strcpy(PartNumber,
                sqltempv.rec3.SQLREC3_FIELD1);
            sqltempv.rec3.
                SQLREC3_FIELD2[sqltempv.rec3.SQLREC3_FIELD2_LEN] = '\0';
            strcpy(PartName,
                sqltempv.rec3.SQLREC3_FIELD2);
            SalesPriceInd =
                sqltempv.rec3.SQLREC3_FIELD3_IND;
            if (sqltempv.rec3.SQLREC3_FIELD3_IND >= 0)
            {
                SalesPrice =
                    sqltempv.rec3.SQLREC3_FIELD3;
            }
        }
    }
}
}

```

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

```

else
  {
  }
}

if ((sqlca.sqlwarn[0] == 'W') || (sqlca.sqlwarn[0] == 'w')) {
  printf("\n SQL WARNING has occurred. The following row");
  printf("\n   of data may not be valid!");
}
if (sqlca.sqlcode == OK) {
  DisplayRow();
}
else if (sqlca.sqlcode == NotFound) {
  printf("\n Row not found!");
}
else if (sqlca.sqlcode == MultipleRows) {
  printf("\n WARNING: More than one row qualifies!");
}
else {
  SQLStatusCheck();
}
EndTransaction();
}
} /* End do */
while (PartNumber[0] != '/');

}/* End of Select Function */

main()          /* Beginning of program */
{

printf("\n Program to SELECT specified rows from");
printf("\n  the Parts Table - cex2");
printf("\n");
printf("\n Event List:");
printf("\n  CONNECT TO ../sampledb/PartsDBE");
printf("\n  BEGIN WORK");
printf("\n  SELECT the specified row from the Parts Table");
printf("\n    until the user enters a '/'");
printf("\n  COMMIT WORK");
printf("\n  RELEASE from ../sampledb/PartsDBE");
printf("\n");

if (ConnectDBE()) {
  Select();
  ReleaseDBE();
}
else
  printf("\n Error: Cannot Connect to ../sampledb/PartsDBE!\n");
} /* End of Program */

```

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

---

## Sample Preprocessor Generated Include Files

Figure 2-7 through Figure 2-9 illustrate, respectively, the type, variable, and externals include files that correspond to the modified source file in Figure 2-9. Note that the preprocessor inserts the following three C compiler directives to reference include files of the following syntax:

```
#include "ModifiedSourceFileName.sqlt"  
#include "ModifiedSourceFileName.sqlv"  
#include "ModifiedSourceFileName.sqle"
```

These three directives are always inserted into the static or global declaration part of the program.

```
typedef char  ownername_type[21];  
typedef char  modulename_type[21];  
typedef short sqlind;  
typedef struct {  
    int      SQLREC1_FIELD1_LEN;  
    char     SQLREC1_FIELD1[133];  
} SQLREC1;  
typedef struct {  
    int      SQLREC2_FIELD1_LEN;  
    char     SQLREC2_FIELD1[20];  
} SQLREC2;  
typedef struct {  
    int      SQLREC3_FIELD1_LEN;  
    char     SQLREC3_FIELD1[20];  
    int      SQLREC3_FIELD2_LEN;  
    char     SQLREC3_FIELD2[32];  
    double   SQLREC3_FIELD3;  
    sqlind   SQLREC3_FIELD3_IND;  
} SQLREC3;  
typedef struct {  
    SQLREC3  dummy1, dummy2;  
} SQLREC4;  
#define sqlwarn0 sqlwarn[0]  
#define sqlwarn1 sqlwarn[1]  
#define sqlwarn2 sqlwarn[2]  
#define sqlwarn3 sqlwarn[3]  
#define sqlwarn4 sqlwarn[4]  
#define sqlwarn5 sqlwarn[5]  
#define sqlwarn6 sqlwarn[6]  
#define sqlwarn7 sqlwarn[7]  
typedef struct {  
    char     sqlaid[8];  
    int      sqlabc;  
    int      sqlcode;
```

Figure 2-7. Sample Type Include File

```

    int    sqlerrl;
    char  sqlerrm[256];
    char  sqlerrp[8];
    int   sqlerrd[6];
    char  sqlwarn[8];
    char  sqlext[8];
} sqlca_type;
typedef struct {
    short sqlnty;
    short sqltype;
    short sqlprec;
    short sqlscale;
    int   sqltotalen;
    int   sqlvallen;
    int   sqlindlen;
    int   sqlvof;
    int   sqlnof;
    char  sqlname[20];
} sqlformat_type;
typedef struct {
    char  sqldaid[8];
    int   sqldabc;
    int   sqln;
    int   sqld;
    sqlformat_type *sqlfmtarr;
    int   sqlnrow;
    int   sqlrrow;
    int   sqlrowlen;
    int   sqlbuflen;
    int   sqlrowbuf;
} sqlda_type;
typedef union {
    int    dummy;
    SQLREC1 rec1;
    SQLREC2 rec2;
    SQLREC3 rec3;
    SQLREC4 rec4;
} sqltempv_type;

```

Figure 2-7. Sample Type Include File (page 2 of 2)



```
static ownername_type sqlowner = "JOANN      ";
static modulename_type sqlmodname = "CEX2    ";
int sqlindex;
char *sqlvar1;
char *sqlvar2;
char *sqlvar3;
char *sqlvar4;
sqltempv_type sqltempv;
```

**Figure 2-8. Sample Variable Include File**

```
extern sqlxbfec();
extern sqlxbinc();
extern sqlxcnhc();
extern sqlxconc();
extern sqlxdduc();
extern sqlxdexc();
extern sqlxdfec();
extern sqlxdopc();
extern sqlxdsbc();
extern sqlxexic();
extern sqlxexuc();
extern sqlxfetc();
extern sqlxiduc();
extern sqlxopkc();
extern sqlxopuc();
extern sqlxplnc();
extern sqlxprec();
extern sqlxsecc();
extern sqlxstpc();
extern sqlxsvpc();
```

**Figure 2-9. Sample Externals Include File**



## Host Variables

---

Host variables are variables 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 a C program must be **declared** in declaration parts of the program. The type descriptions of host variables must be compatible with ALLBASE/SQL **data types**. The type descriptions of host variables must also satisfy certain preprocessor criteria.

This chapter identifies where in a C program you can use host variables and then discusses how to write type descriptions that complement the way host variables are used. See the chapter, “Simple Data Manipulation”, for a sample program that uses host variables.

---

## 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 C programs must do the following:

- 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, an underscore (`_`), or valid characters for any native language you are using.
- Begin with an alphabetic character, although the prefix *SQL* is not recommended.
- Not be the same as any ALLBASE/SQL or C reserved word.

In all SQL commands containing host variables, syntax requires that 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 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 negative number 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 BULK SELECT command syntax, 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[0], since arrays start at an index of 0. 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

Host variables may be declared wherever you can declare variables in C programs. For the purpose of this discussion, we define **declaration section** as the portion of a C program where variables having the scope of a file, a function, or a block can be declared.

At run time, the scope of a host variable is the same as that of any other C variable declared in the same declaration part. At preprocessing time, however, *all* host variable declarations are treated as global declarations. Therefore host variables having the same name in different declaration parts must also have the same C type description in each variable declaration.

## Creating Declaration Sections

Host variables must be declared in what is known as a declare section. A declare 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 3-1). More than one declare section may appear in a given declaration part. However, a host variable name may appear only once in a given declaration part.

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

```
EXEC SQL BEGIN DECLARE SECTION;
```

```
    int           OrderNumber;  
    |            |  
    |            |  
    |            data name  
    |  
data type
```

```
EXEC SQL END DECLARE SECTION;
```

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



```

.
.
.
EXEC SQL BEGIN DECLARE SECTION;
.
.  Declarations for global host variables.
.
EXEC SQL END DECLARE SECTION;
.
.
.
int query()
{
.
.
.
EXEC SQL BEGIN DECLARE SECTION;
.
.  Declarations for local host variables.
.
EXEC SQL END DECLARE SECTION;
.
.
.
EXEC SQL BEGIN DECLARE SECTION;
.
.  Declarations for local host variables.
.
EXEC SQL END DECLARE SECTION;
.
.
}
.
.
.
main()
{
.
.
.
}

```

**Figure 3-1. Host Variable Declarations**

## Declaring Variables for Data Types

Any variable can be used as a host variable. Table 3-1 summarizes C data declaration syntax for host variables of each ALLBASE/SQL data type. Only the type descriptions shown in Table 3-1 are supported by the C preprocessor. Note in particular that the preprocessor does not support user-defined data types.

### CHAR Data

A CHAR column can be declared for character strings ranging from 1 to 3996 bytes.

A single byte of the char data type in ALLBASE/SQL is directly equivalent to a single byte of the CHAR data type in C. Strings greater than one byte in C are associated with the char array data type. Character arrays in C are not directly equivalent to the ALLBASE/SQL CHAR data type; however, they are compatible. ALLBASE/SQL handles the conversion between ALLBASE/SQL CHAR data types and C char array data types through code generated by the preprocessor and embedded in the application's modified source code file.

For C strings (character arrays), C has the convention of using an ASCII 0 ('\0'), the null character, to mark the end of the string. Therefore, char host variables declared in C must have a size one greater than their ALLBASE/SQL column definition, to allow for the null character at the end of the string. The PartNumber column in the PurchDB.Parts table is defined as CHAR(30). The associated host variable is therefore declared:

```
char    PartNumber[31];
```

When ALLBASE/SQL assigns CHAR data to a char array host variable, the total length of the ALLBASE/SQL CHAR field is stored in the host variable including any trailing blanks on the right of the data string. An ASCII 0 is then added after the last byte of the string.

### VARCHAR Data

A VARCHAR column can be declared for strings ranging from 1 to 3996 bytes. ALLBASE/SQL stores only the actual value of the string, not any trailing blanks.

Strings greater than one character in C are equivalent to the VARCHAR data type in ALLBASE/SQL. C host variables for VARCHAR data types in ALLBASE/SQL are declared the same as variables declared for CHAR data types. The only difference is that when ALLBASE/SQL assigns VARCHAR data to a char array host variable, no trailing blanks are added. An ASCII 0 is placed after the last character of the C string based on the specified length of the string.

The VendorRemarks column in the PurchDB.Vendors table is defined as VARCHAR(60). It is therefore declared:

```
char    VendorRemarks[61];
```

**Table 3-1. Data Type Declarations**

SQL DATA TYPES	C DATA DECLARATIONS
CHAR(1)	char <i>dataname</i> ;
CHAR(n)	char <i>dataname</i> [n+1];
VARCHAR(n)	char <i>dataname</i> [n+1]; *
SMALLINT	short <i>dataname</i> ; or short int <i>dataname</i> ;
INTEGER	int <i>dataname</i> ; or long int <i>dataname</i> ; or long <i>dataname</i> ;
REAL	float <i>dataname</i> ;
FLOAT(1..24)	float <i>dataname</i> ;
FLOAT(25..53)	double <i>dataname</i> ;
DOUBLE PRECISION	double <i>dataname</i> ;
BINARY	sqlbinary <i>dataname</i> ; sqlbinary <i>dataname</i> [n];
VARBINARY	sqlvarbinary <i>dataname</i> [m]; **
DECIMAL	double <i>dataname</i> ;
DATE	char <i>dataname</i> [11];
TIME	char <i>dataname</i> [9];
DATETIME	char <i>dataname</i> [24];
INTERVAL	char <i>dataname</i> [21];
	* This declaration is for non-dynamic commands only. Refer to the chapter, "Using Dynamic Operations," for a description of how to use VARCHAR dynamically. ** See the "BINARY Data" section later in this chapter for the calculation of m.

### SMALLINT Data

You can assign values ranging from -32,768 to +32,767 to a column defined as SMALLINT.

### INTEGER Data

You can assign values ranging from -2,147,483,648 to +2,147,483,647 to a column defined as INTEGER.

### 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. (This conforms to ANSI SQL86 level 2 specifications.) The keyword 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 3-2. ALLBASE/SQL Floating Point Column Specifications**

Possible Keywords	Range of Possible Values	Stored In and Boundary Aligned On
REAL or FLOAT( <i>n</i> ) where <i>n</i> = 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( <i>n</i> ) where <i>n</i> = 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 statement 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 the same manner except that CHAR data is blank padded.)

**Binary Data Compatibility.** BINARY and VARBINARY data types in an ALLBASE/SQL database 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.

When deciding on whether to use LONG BINARY versus LONG VARBINARY, and if space is your main consideration, you would choose LONG VARBINARY. 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 in this document titled “Defining and Using Long Columns” for information about using LONG column data.

**Declaring Host Variables for BINARY Data.** Host variables for BINARY data columns must be declared as sqlbinary, as in the following example:

```
EXEC SQL BEGIN DECLARE SECTION;
:
sqlbinary  BinaryHostVariableName[n];
:
EXEC SQL END DECLARE SECTION;
```

The host variable array size  $n$  equals the length of the column as defined in the database.

At preprocessing time, the ALLBASE/SQL preprocessor sqlbinary data type is defined as an unsigned char with the following statement in the SQL Type Include File:

```
typedef unsigned char sqlbinary;
```

An sqlbinary host variable is used for fixed length data. It is your responsibility to appropriately load binary data into the host variable before an insert or update operation.

In the following example, data from a binary column defined with a length of 12 is selected into an sqlbinary host variable.

```
⋮  
  
EXEC SQL BEGIN DECLARE SECTION;  
  
sqlbinary   BinaryHV[12];  
  
EXEC SQL END DECLARE SECTION;  
  
⋮  
  
EXEC SQL SELECT BinaryColumn  
           INTO :BinaryHV  
           FROM TableA;  
  
⋮
```

**Declaring Host Variables for VARBINARY Data.** Host variables for VARBINARY data columns must be declared as sqlvarbinary, as in the following example:

```
EXEC SQL BEGIN DECLARE SECTION;  
⋮  
sqlvarbinary   VarbinaryHostVariableName [m]  
⋮  
EXEC SQL END DECLARE SECTION;
```

At preprocessing time, the ALLBASE/SQL preprocessor sqlvarbinary data type is defined as an integer with the following statement in the SQL Type Include File:

```
typedef int sqlvarbinary;
```

You specify the host variable array size  $m$  based on the following formula:

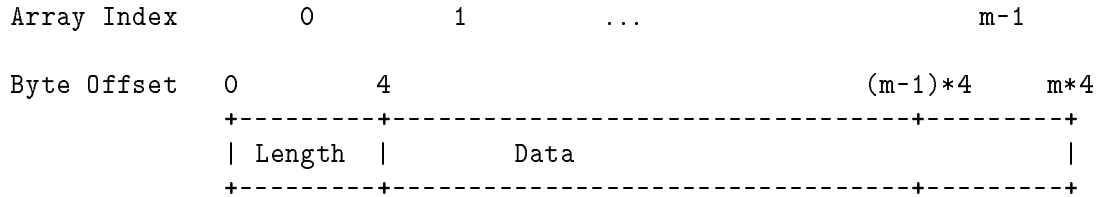
$$m = 1 + [n/4]$$

where:

- $m$  = the host variable array size
- $n$  = the length of the column as defined in the database
- $[n/4]$  = the LeastInteger of  $(n/4)$

LeastInteger is the smallest integer  $\geq (n/4)$ .

In order to pass information between an sqlvarbinary host variable and the database, a special format is used. The internal format of an sqlvarbinary host variable is illustrated below:



where:

- Length, in the first four bytes, represents the actual data length.
- Data starts from byte offset four and represents the varbinary data.
- (m-1)\*4 equals the byte offset of the last element of the sqlvarbinary array.
- m\*4 equals the end of the sqlvarbinary array.

For example, to declare an sqlvarbinary host variable for a varbinary column having a maximum length of nine, you define the host variable with an array size of four (one four byte element to hold the actual data length and three four byte elements for the data):

```
EXEC SQL BEGIN DECLARE SECTION;

sqlvarbinary   VarbinaryHV[4];

EXEC SQL END DECLARE SECTION;
```

**Inserting and Updating VARBINARY Data.** Before issuing an INSERT or UPDATE statement, you must load the sqlvarbinary host variable using the format mentioned in the previous section. Two examples of loading data into an sqlvarbinary host variable are presented below. The first example loads data from a buffer; the second example loads data from a union structure.

**Using a Buffer**

Suppose your varbinary column is defined in the database with a maximum length of 16. You want to load it with data from a buffer called bbuff as follows:

```
⋮

int length;
char bbuff[16];

EXEC SQL BEGIN DECLARE SECTION;
```

*Derive the host variable length from the formula described in the previous section, "Declaring Host Variables for Varbinary Data," as follows:  
1 + the LeastInteger of (16/4).*

```
sqlvarbinary   VarbinaryHV[5];

EXEC SQL END DECLARE SECTION;

⋮
```

*Load the first array element with the actual data length.  
Here we'll assume actual length to be 11 bytes.*

```
VarbinaryHV[0] = 11;
```

*Load the data in bbuff, starting at the second element of the array.*

```
memcpy(&VarbinaryHV + 1, &bbuff, 11);
```

```
EXEC SQL INSERT INTO TableA VALUES (:VarbinaryHV, ... );
```

```
:
```

### Using a Union Structure

Here is another method of loading the same host variable. In the following example, the data length and data are loaded into a union structure and then into the sqlvarbinary host variable:

```
:
```

```
int length;  
char bbuff[16];
```

```
union u_t {  
    struct {  
        int length;  
        int data[4];  
    } s1;  
    int ubuff[5];  
} u1;
```

```
EXEC SQL BEGIN DECLARE SECTION;
```

```
:
```

*Derive the host variable length from the formula described in the previous section,  
"Declaring Host Variables for Varbinary Data," as follows:  
1 + the LeastInteger of (16/4).*

```
sqlvarbinary VarbinaryHV[5];
```

```
EXEC SQL END DECLARE SECTION;
```

```
:
```

*Load the length field of the structure with the actual column data length.  
Here we'll assume actual length to be 11 bytes.*

```
u1.s1.length = 11;
```



*Load the data in bbuff, into the data field of the structure.*

```
memcpy(u1.s1.data, &bbuff, 11);
```

*Load the length and the data into the varbinary host variable.*

```
memcpy(&VarbinaryHV, &(u1.ubuff), sizeof(u1.ubuff));
```

```
EXEC SQL INSERT INTO TableA VALUES (:VarbinaryHV, ... );
```

```
:
```

**Selecting and Fetching VARBINARY Data.** After the successful execution of a SELECT or FETCH statement, the length of the data returned for any varbinary column is found in the first element of the related host variable array.

Two examples of retrieving data from an sqlvarbinary host variable are presented below. The first example loads retrieved data into a buffer; the second example loads retrieved data into a union structure. The same data declarations used for the examples in the previous section are assumed.

### Using a Buffer

The following example selects data into an sqlvarbinary host variable, loads it into a buffer named bbuff, and saves the length of the data in an integer variable named length:

```
:
```

```
EXEC SQL SELECT * FROM TableA INTO :VarbinaryHV, ... ;
```

```
memcpy (&bbuff, &VarbinaryHV+1, VarbinaryHV[0]);
```

```
length = VarbinaryHV[0];
```

```
:
```

### Using a Union Structure

The following example selects data into an sqlvarbinary host variable, loads it into a union structure, saves the length of the data in an integer variable named length, and saves the data in a buffer named bbuff:

```
:
```

```
EXEC SQL SELECT * FROM TableA INTO :VarbinaryHV, ... ;
```

```
memcpy(&(u1.ubuff), &VarbinaryHV, sizeof(u1.ubuff));
```

```
length = u1.s1.length;
```

```
memcpy(&bbuff, &(u1.s1.data), u1.s1.length);
```

```
:
```

## DECIMAL Data

The DECIMAL data type is not supported by ALLBASE/SQL C. The DECIMAL data type is compatible with an ALLBASE/SQL C double data type.

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.

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 in the chapter “Using Dynamic Processing” for an example of this conversion. Also refer to the chapter “BULK Table Processing” for information on using the format array record.

---

**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

DATE, TIME, DATETIME, and INTERVAL data types are declared as character strings. (See the previous section, “CHAR Data.”) For example:

```
/* Declare host variables and, where applicable, indicator variables. */

EXEC SQL BEGIN DECLARE SECTION;          /* DATETIME DATA TYPE */
char BatchStamp[24];                    /* DATE DATA TYPE */
char TestDate[11];
sqlind TestDateInd;                    /* TIME DATA TYPE */
Char TestStart[9];
sqlind TestStartInd;                   /* INTERVAL DATE TYPE */
char LabTime[21];
sqlind LabTimeInd;
EXEC SQL END DECLARE SECTION;

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

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

See the chapter, “Programming with ALLBASE/SQL Functions,” for information on using date/time data types with date/time functions.

## 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. (This conforms to ANSI SQL1 level 2 with addendum-1 and FIPS 127 standards.)

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. 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:

- NULL.
- USER (this indicates the current DBEUser ID).
- 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 document 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 under the following circumstances:

- 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 3-3 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 3-3. C Data Type Equivalency and Compatibility**

ALLBASE\SQL Data Types	char	char[n]	short shortint	int long longint	double longfloat
CHAR	E	C	I	I	I
VARCHAR	C	C	I	I	I
BINARY	C	C	I	I	I
VARBINARY	I	I	C	C	I
DATE	C	C	I	I	I
TIME	C	C	I	I	I
DATETIME	C	C	I	I	I
INTERVAL	C	C	I	I	I
SMALLINT	I	I	E	C	C
INTEGER	I	I	C	E	C
FLOAT	I	I	C	C	E
DECIMAL	I	I	C	C	C

As the following example illustrates, the ISQL INFO command 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

For example, the query in Figure 3-2 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:

- PurchasePrice is declared as a double variable because it holds the DECIMAL result of an aggregate function on a DECIMAL column.
- Discount is declared as a double variable because it is used in an arithmetic expression with a DECIMAL column, PurchasePrice.
- OrderQty is declared as a short int variable because it holds the 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.
- OrderNumber is an integer variable because the column whose data it holds is INTEGER.

```

.
EXEC SQL BEGIN DECLARE SECTION;
    double          Discount  ;
    double          PurchasePrice  ;
    short int       OrderQty  ;
    sqlind          OrderQtyInd  ;
    int             OrderNumber ;
.
.
EXEC SQL END DECLARE SECTION;
.
{
.
    EXEC SQL SELECT  PurchasePrice * :Discount  ,
                    OrderQty,
                    INTO :PurchasePrice,
                        :OrderQty :OrderQtyInd
                    FROM  PurchDB.OrderItems
                    WHERE  OrderNumber = :OrderNumber
.
.
}

```

**Figure 3-2. Declaring Host Variables for Single-Row Query Results**

The example in Figure 3-3 is similar to that in Figure 3-2. This query, however, is a BULK query, which may return a multiple-row query result. And it incorporates a HAVING clause.

- 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 short int 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.

```

.
.
EXEC SQL BEGIN DECLARE SECTION;
    double                Discount;
    struct {
        double            PurchasePrice;
        short int         OrderQty;
        sqlind            OrderQtyInd;
        int               OrderNumber;
    } OrdersArray[26];
    short int             FirstRow;
    short int             TotalRows;
    int                   LowValue;
    int                   HighValue;
    int                   GroupCriterion;
.
.
EXEC SQL END DECLARE SECTION;
.
.
{
.
.
    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 3-3. Declaring Host Variables for Multiple-Row Query Results**



## String Data Conversion

When ALLBASE/SQL stores the characters in a C string into a CHAR column, the final ASCII 0 is removed and any remaining positions to the right are padded with spaces. Internally, when ALLBASE/SQL stores the characters in a C string to a VARCHAR column, it only stores the string up to but not including the ASCII 0. The length of the string is stored in a four-byte header in the front of each VARCHAR data type.

When ALLBASE/SQL moves VARCHAR data to a character array variable, only the length of the string is moved, no trailing blanks are added. ALLBASE/SQL embeds preprocessor generated code into the modified source code file to place an ASCII 0 at the end of the string variable. Refer to the “Using Dynamic Operations” chapter in this manual for further information specific to dynamic operations.

## 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 in bytes before truncation:

```
EXEC SQL BEGIN DECLARE SECTION;
      char          LittleString[n];
      sqlind        LittleStringInd;
.
.
.
EXEC SQL END DECLARE SECTION;
.
.
.
{
.
.
.
      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 length of the string in bytes into indicator variable LittleStringInd. In this example, the maximum number of bytes that can be stored in LittleString is n-1, since the last byte is always reserved for the ASCII 0 null character.

If a column is too small to hold a string in an INSERT or an UPDATE operation, the string is truncated and stored. The sqlca sqlwarn[1] field is set to W when this occurs.

It is possible to store native language data in a character column defined as n-computer. It is the programmer's responsibility to verify the language definition of the column that is to receive data. If the character column is defined for a native language, truncation will always occur 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 of the lesser type is converted into data of the greater type, and the result is expressed in the greater type. ALLBASE/SQL numeric types 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;
      int          Discount;
      int          MaxPurchasePrice;
.
.
.
EXEC SQL END DECLARE SECTION;
.
.
.
{
.
.
.
      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 subtraction, 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. In addition, Table 3-4 provides the syntax of these special elements.

**Table 3-4. Program Element Declarations**

PROGRAM ELEMENT	C DATA DECLARATIONS
Indicator variable	<code>sqlind indvarname;</code>
Array of n rows Data values	<code>struct structtypename{     validdatatype column1name;     validdatatype column2name;</code>
Indicator variable	<code>sqlind indvarname; } structname [n];</code>
StartIndex	<code>short startindexname; or int startindexname;</code>
NumberOfRows	<code>short numrowsname; or int numrowsname;</code>
Dynamic commands	<code>char commandname [n+1];</code>
Savepoint numbers	<code>int savepointname;</code>
Message catalog messages	<code>char messagename [n+1];</code>
DBEnvironment name	<code>char DBEName [n+1];</code>

### sqlca Array

Every ALLBASE/SQL C 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 C preprocessor parses this command, it inserts the following type definition into the modified source file:

```
sqlca_type sqlca;
```

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 and the sqlca,” 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.*

```
sqlformat_type      sqlfmts [MaxFmtArray];
```

*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.*

```
char                DataBuffer [MaxDataBuff];
```

*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. Figure 3-3 provides an example.

## Indicator Variables

Each indicator variable field must be declared immediately following the host variable field it describes as shown in Figures 3-2 and 3-3. 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. In Figure 3-4, the host variable is declared to hold a command as large as 2048 bytes.

```
.  
. .  
. .  
  
EXEC SQL BEGIN DECLARE SECTION;  
    char          DynamicCommand[2048]  ;  
  
. .  
. .  
EXEC SQL END DECLARE SECTION;  
  
. .  
. .  
{  
. .  
. .  
    EXEC SQL PREPARE  CommandOnTheFly  
        FROM :DynamicCommand  ;  
. .  
. .  
}
```

Figure 3-4. 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;
      int          SavePoint1  ;
:
EXEC SQL END DECLARE SECTION;
:
{
:
      EXEC SQL SAVEPOINT :Savepoint1  ;
:
      EXEC SQL ROLLBACK WORK TO :Savepoint1  ;
:
}

```

**Figure 3-5. Declaring Host Variables for Savepoint Numbers**

## Messages from the Message Catalog

The maximum size of a message catalog message is 256 bytes. Figure 3-6 illustrates how a host variable for holding a message might be declared.

```

:
EXEC SQL BEGIN DECLARE SECTION;
      char          SQLMessage[256]  ;
:
EXEC SQL END DECLARE SECTION;
:
{
:
      EXEC SQL SQLEXPLAIN :SQLMessage  ;
      printf("%s\n",SQLMessage);
:
}

```

**Figure 3-6. 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.

```
EXEC SQL BEGIN DECLARE SECTION;
char          SomeDBE[128];
.
.
.
EXEC SQL END DECLARE SECTION;
.
.
.
{
printf("\n Enter DBEnvironment name> ");
scanf("%s",SomeDBE);
EXEC SQL CONNECT to :SomeDBE;
.
.
.
}
```

**Figure 3-7. 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.





## 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 sqlwarn[0] values for you, then takes an action based on information you provide in the WHENEVER command.
- You can write C 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 in the order in which the errors or warnings occurred. When `ALLBASE/SQL` rolls back the current transaction, it does not continue to look for errors. This means that the last message retrieved will indicate the cause of the roll back. 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. This ensures that the previous delete won't remove part of the information about this order when the rest of the information could not be deleted.*

## **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 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_type sqlca;
```

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.

The C preprocessor generates the following record declaration for sqlca\_type in the type declaration include file:

```
typedef struct {
    char    sqlaid[8];
    int     sqlabc;
    int     sqlcode;
    int     sqlerrl;
    char    sqlerrm[256];
    char    sqlerrp[8];
    int     sqlerrd[6];
    char    sqlwarn[8];
    char    sqlext[8];
} sqlca_type;
```

The following elements in this record are available for you to use in status checking and are accessed as follows. The other elements are reserved for use by ALLBASE/SQL only.

```
sqlcode or sqlca.sqlcode
sqlca.sqlerrd[2]
sqlca.sqlwarn[0] or sqlca.sqlwarn0
sqlca.sqlwarn[1] or sqlca.sqlwarn1
sqlca.sqlwarn[2] or sqlca.sqlwarn2
sqlca.sqlwarn[3] or sqlca.sqlwarn3      (used only for dynamic commands)
sqlca.sqlwarn[6] or sqlca.sqlwarn6
```

---

**Note** In conformance with the ANSI standard, either sqlcode or sqlca.sqlcode may be used to address this particular element. And each sqlwarn element can be addressed in two different ways.

---

The following table gives an overview of how ALLBASE/SQL sets these fields. Each field is then described with brief examples of how you can use it in one of two ways, including examples for using SQLEXPLAIN. Methods of handling specific status checking tasks are found in the succeeding section, "Approaches to Status Checking."

### 4-4 Runtime Status Checking and the sqlca

**Table 4-1. sqlca Status Checking Fields**

<b>FIELD NAME</b>	<b>SET TO</b>	<b>CONDITION</b>
sqlca.sqlcode or sqlcode	0 less than 0 100	no error occurred during command execution error, command not executed no rows qualify for DML operation (does not apply to dynamic commands)
sqlca.sqlerrd[2]	number of rows put into output host variables number of rows processed 0 0	data retrieval operation data change operation, [BULK]INSERT, Type 2 INSERT, UPDATE, or DELETE error in single row data change operation 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 aggregate 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 an error condition and without generating a no rows qualify condition.
- A negative number, when an error condition exists and an ALLBASE/SQL command cannot be executed.
- 100, when no rows qualify for one of the following commands, but no error condition exists:

```
SELECT
INSERT
UPDATE (non-dynamic execution only)
DELETE (non-dynamic execution only)
BULK SELECT
FETCH
BULK FETCH
UPDATE WHERE CURRENT
DELETE WHERE CURRENT
```

Note that the absolute value of sqlcode is the same as the absolute value associated with its corresponding message in the ALLBASE/SQL message catalog. This absolute value is part of the returned message. If an error occurs, the message number is preceded by DBERR. 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 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) {
        printf("No rows qualified for this operation.\n");
    }
    else
        if (sqlca.sqlcode < 0) SQLStatusCheck();
    .
    .
    .
int SQLStatusCheck()
{
    do {
        EXEC SQL SQLEXPLAIN :SQLMessage;
        printf("%s\n",SQLMessage);
    } while (sqlca.sqlcode != 0);
}
```

The function 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 *less than 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 and when sqlwarn[0] contains a W.

The sqlcode is also used in *implicit* status checking:

- 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 a2000 is executed. When ALLBASE/SQL detects an sqlcode of 100, the code routine at label a4000 is executed instead:

```
EXEC SQL WHENEVER SQLERROR GOTO a2000;
EXEC SQL WHENEVER NOT FOUND GOTO a4000;
```

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 appears.

The scope of WHENEVER commands is fully explained later in this chapter under “Implicit Error Handling Techniques.”

## sqlerrd[2]

sqlca.sqlerrd[2] 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 transaction. 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-processing 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:

```
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[2] 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.



## 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 can *not* 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) {
        do {
            DisplayMessage();
        } while (sqlca.sqlcode != 0);
    }
    .
    .
    .
int DisplayMessage()
{
    EXEC SQL SQLEXPLAIN :SQLMessage;
    printf("%s\n",SQLMessage);
}
```

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) & (sqlca.sqlwarn[0] == 'W')) {
  do {
    DisplayWarning();
  } while (sqlca.sqlwarn[0] == 'W');
}
.
.
.
int DisplayWarning()
{
  EXEC SQL SQLEXPLAIN :SQLMessage;
  printf("%s\n",SQLMessage);
}

```

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 a3000;
EXEC SQL WHENEVER SQLERROR GOTO a2000;

```

## 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:

```

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 the following occurs:

- 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)

```

## sqlwarn[2]

A W in sqlwarn[2] indicates that at least one null value was eliminated from the argument set of an aggregate function.

For example:

```
EXEC SQL SELECT  MAX(OrderQty)
           INTO  :MaxOrderQty
           FROM  PurchDB.OrderItems;
```

If any OrderQty values are null, the following occurs:

- 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)
```

## sqlwarn[3]

A W in sqlwarn[3] indicates that the number of host variables specified in a *dynamic* SELECT or FETCH statement is unequal to the number of columns in the table being operated on.

For 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. */
sqlda.sqlbuflen=sizeof(DataBuffer);
sqlda.sqlnrow=((sqlbuflen)/(sqlrowlen));
sqlda.sqlrowbuf=&databuffer;
sqlda.sqld=1; /*sqlda.sqld is incorrectly reset by the program. */
.
/* Do the fetch. */
EXEC SQL FETCH DynamicCursor USING DESCRIPTOR SQLDA;
```

The FETCH will fail and the following occurs:

- 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)
```

## 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 is 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) {
    if (sqlca.sqlwarn[6] == 'W') {
        SQLStatusCheck();
        TerminateProgram();
    }
    else
        SQLStatusCheck();
}
```

In this example, the program executes the function 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 sqlwarn[0].)
- 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 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 `sqlwarn[0]` after processing any SQL command *except* the following:

```
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 function 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.

```

int Procedure2()
{
    EXEC SQL SQLCommand2;
}
int Procedure1()
{
    EXEC SQL WHENEVER SQLERROR GOTO a2000;
    EXEC SQL SQLCommand1;
}
.
.
.
{
    Procedure1();
    Procedure2();
}
EXEC SQL WHENEVER SQLERROR CONTINUE;

```

The code that the preprocessor generates depends on the condition and action in a `WHENEVER` command. In the example above, the preprocessor inserts a test for a negative `sqlcode` and a statement that invokes the code routine located at Line Label `a2000`:

```

#if 0
EXEC SQL WHENEVER SQLERROR GOTO a2000;
#endif

#if 0
EXEC SQL SQLCommand1;
#endif

```

*Statements for executing `SQLCommand1` appear here.*

```

if (sqlca.sqlcode < 0) {
    goto a2000;
}

```

As the previous example illustrates, you pass control to an exception-handling routine with a `WHENEVER` command, by using a `GOTO` statement with an alpha-numeric line label rather than a function name. Therefore after the exception-handling routine is executed, control cannot *automatically* return to the statement which invoked it. You must use another `GOTO` statement to explicitly pass control to a specific point in your program:

```
/* WHENEVER Routine -- SQL Error */
a2000:
  if ((sqlca.sqlcode <= -14024) || (sqlca.sqlcode == -4008)) {
    TerminateProgram();
  }
  else
    do {
      EXEC SQL SQLEXPLAIN :SQLMessage;
      printf("%s\n",SQLMessage);
    } while (sqlca.sqlcode != 0);
    goto a500;      /* Goto Restart/Reentry point of function */
```

This exception-handling routine *explicitly* checks the first `sqlcode` returned. The program either 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 routine 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” later in the chapter.



## Program Illustrating Implicit and Explicit Status Checking

The program in Figure 4-1 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
RELEASE
```

- 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 routine at Label a1000 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 a2000 is executed when an error occurs during the processing of the `SELECT` command. This code routine *explicitly* examines the `sqlcode` value to determine whether it is -10002, in which case it displays a warning message. If `sqlcode` contains another value, function `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` less than `-14024`):

- If a deadlock or shared memory problem occurred, the program attempts to execute the `SelectData` function 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 a3000 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 a4000. 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.

```

/* Program cex5 */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* This program illustrates the use of SQL's SELECT command to      */
/* retrieve one row or tuple of data at a time.                      */
/* This programs is the same as cex2 with added status checking      */
/* and deadlock routines.                                           */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

typedef int boolean;

char    response[2];
boolean Abort;
boolean SQLCommandDone;
int     TryCounter;

#include <stdio.h>

#define    OK                0
#define    NotFound          100
#define    MultipleRows     -10002
#define    DeadLock         -14024
#define    FALSE            0
#define    TRUE             1
#define    NoMemory         -4008
#define    TryLimit         3

sqlca_type sqlca;    /* SQL Communication Area */

        /* Begin Host Variable Declarations */
EXEC SQL BEGIN DECLARE SECTION;
char    PartNumber[17];
char    PartName[31];
double  SalesPrice;
sqlind  SalesPriceInd;
char    SQLMessage[133];
EXEC SQL END DECLARE SECTION;
        /* End Host Variable Declarations */

```

Figure 4-1. Program cex5: Implicit and Explicit Status Checking

```

int SQLStatusCheck() /* Function to Display Error Messages */
{
if ((sqlca.sqlcode == DeadLock) || (sqlca.sqlcode == NoMemory)) {
    if (TryCounter == TryLimit) {
        SQLCommandDone = TRUE;
        printf("\n Could not complete transaction. You may want to try again.");
    }
    else
        SQLCommandDone = FALSE;
    }
else {
    Abort = FALSE;
    if ((sqlca.sqlwarn[6] = 'W') || (sqlca.sqlwarn[6] = 'w'))
        Abort = TRUE;
    }

do {
EXEC SQL SQLEXPLAIN :SQLMessage;
printf("%s\n",SQLMessage);
} while (sqlca.sqlcode != 0);

if (Abort) {

    EndTransaction();
    ReleaseDBE();
    }
} /* End SQLStatusCheck Function */

EXEC SQL WHENEVER SQLERROR GOTO a1000;

boolean ConnectDBE() /* Function to Connect to PartsDBE */
{
boolean ConnectDBE;
ConnectDBE = TRUE;
printf("\n Connect to PartsDBE");
EXEC SQL CONNECT TO 'PartsDBE';
goto exit;

a1000:          /* WHENEVER SQLERROR entry point 1 */
    SQLStatusCheck();
    EndTransaction();
    ReleaseDBE();

exit:
    return (ConnectDBE);
} /* End of ConnectDBE Function */

```

1

Figure 4-1. Program cex5: Implicit and Explicit Status Checking (page 2 of 6)

```

boolean BeginTransaction()    /* Function to Begin Work */
{
boolean BeginTransaction;
BeginTransaction = TRUE;

printf("\n");
printf("\n Begin Work");
EXEC SQL BEGIN WORK;
goto exit;

a1000:                /* WHENEVER SQLERROR entry point 1 */
    SQLStatusCheck();
    EndTransaction();
    ReleaseDBE();

exit:
    return (BeginTransaction);

} /* End BeginTransaction Function */

int EndTransaction() /* Function to Commit Work */
{
printf("\n");
printf("\n Commit Work");
EXEC SQL COMMIT WORK;
goto exit;

a1000:                /* WHENEVER SQLERROR entry point 1 */
    SQLStatusCheck();
    ReleaseDBE();

exit:
    return(0);
} /* End EndTransaction Function */

int ReleaseDBE() /* Function to Release PartsDBE */
{
printf("\n");
printf("\n Release PartsDBE");
printf("\n");
EXEC SQL RELEASE;
goto exit;
}

```

**Figure 4-1. Program cex5: Implicit and Explicit Status Checking (page 3 of 6)**

```

a1000:                /* WHENEVER SQLERROR entry point 1 */
    SQLStatusCheck();
    EndTransaction();

exit:
    return(0);
} /* End ReleasedBE Function */

EXEC SQL WHENEVER SQLERROR CONTINUE;

int DisplayRow() /* Function to Display Parts Table Rows */
{

    printf("\n");
    printf(" Part Number:   %s\n", PartNumber);
    printf(" Part Name:      %s\n", PartName);

    if (SalesPriceInd < 0) {
        printf("\n Sales Price:   is NULL");
    }
    else
        printf(" Sales Price: %10.2f\n", SalesPrice);
} /* End of DisplayRow Function */

EXEC SQL WHENEVER SQLERROR GOTO a2000;
EXEC SQL WHENEVER SQLWARNING GOTO a3000;
EXEC SQL WHENEVER NOT FOUND GOTO a4000;

int Select() /* Function to Query Parts Table */
{

do {
    if (SQLCommandDone) {
        printf("\n");
        printf("\n Enter Part Number within Parts Table or '/' to STOP > ");
        scanf("%s",PartNumber);
        printf("\n");

        TryCounter = 0;
    }

    if (PartNumber[0] != '/') {

        BeginTransaction();
        TryCounter = TryCounter + 1;

```

**Figure 4-1. Program cex5: Implicit and Explicit Status Checking (page 4 of 6)**

```

printf("\n SELECT PartNumber, PartName, SalesPrice");
EXEC SQL SELECT  PartNumber, PartName, SalesPrice
              INTO :PartNumber,
                  :PartName,
                  :SalesPrice :SalesPriceInd
              FROM  PurchDB.Parts
              WHERE PartNumber = :PartNumber;

/* If no errors occur, set command done flag and display the row. */

    SQLCommandDone = TRUE;
    DisplayRow();
    EndTransaction();
} /* End if */
} /* End do */
while (PartNumber[0] != '/');
goto exit;

a2000:          /* WHENEVER SQLERROR entry point 2 */
if (sqlca.sqlcode == MultipleRows) {
    printf("\n");
    printf("\n WARNING: More than one row qualifies!");
    EndTransaction();
    SQLCommandDone = TRUE;
    Select();      /* Call Restart/Reentry point */
}
else
    SQLStatusCheck();
    Select();      /* Call Restart/Reentry point */

a3000:          /* WHENEVER SQLWARNING entry point */
printf("\n SQL WARNING has occurred. The following row");
printf("\n      of data may not be valid!");
DisplayRow();
EndTransaction();
SQLCommandDone = TRUE;
Select();      /* Call Restart/Reentry point */

a4000:          /* WHENEVER NOT FOUND entry point */
printf("\n");
printf("\n Row not found!");
EndTransaction();
SQLCommandDone = TRUE;
Select();      /* Call Restart/Reentry point */

exit:
    return(0);
}/* End of Select Function */

```

**Figure 4-1. Program cex5: Implicit and Explicit Status Checking (page 5 of 6)**

```

main()          /* Beginning of program */
{

printf("\n Program to SELECT specified rows from");
printf("\n  the Parts Table - cex5");
printf("\n");
printf("\n Event List:");
printf("\n  CONNECT TO PartsDBE");
printf("\n  BEGIN WORK");
printf("\n  SELECT the specified row from the Parts Table");
printf("\n    until the user enters a '/'");
printf("\n  COMMIT WORK");
printf("\n  RELEASE from PartsDBE");
printf("\n");

if (ConnectDBE()) {

    SQLCommandDone = TRUE; /* Initialize command done flag */

    Select();
    ReleaseDBE();
}
else
    printf("\n Error: Cannot Connect to PartsDBE!\n");

} /* End of Program */

```

**Figure 4-1. Program cex5: Implicit and Explicit Status Checking (page 6 of 6)**

## Explicit Status Checking Techniques

With explicit error handling, you invoke a function after *explicitly* checking sqlca values rather than using the WHENEVER command. The program in Figure 4-1 has already illustrated several uses of explicit error handling to:

- 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 the SELECT operation.

The example in Figure 4-1 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 4-2 contains four data manipulation operations: INSERT, UPDATE, DELETE, and SELECT. Each of these operations is executed within its own function.

As in the program in Figure 4-1, one function is used for explicit status checking: SQLStatusCheck. Unlike the program in Figure 4-1, however, this function is invoked after an *explicit* test of sqlcode is made immediately following each data manipulation operation.

Because status checking is performed in a called function rather than in a routine following the embedded SQL command, control returns to the point in the program where SQLStatusCheck is invoked.



```

#define      Deadlock          -14024
#define      OK                0
#define      NotFound         100
#define      MultipleRows     -10002
#define      NoMemory         -4008
:
int SelectActivity()
{
    This function prompts for a number that indicates
    whether the user wants to SELECT, UPDATE, DELETE,
    or INSERT rows, then invokes a function that
    accomplishes the selected activity. The DONE flag
    is set when the user enters a slash.
}
:
int InsertData()
{
    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 != 0K) SQLStatusCheck();
    :
}

```

**Figure 4-2. Explicit Status Checking Procedures**

```

int UpdateData()
{
    This function verifies that the row(s) to be changed
    exist, then invokes function DisplayUpdate to accept
    new data from the user.

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

    switch(sqlca.sqlcode) {

    case OK:      DisplayUpdate();
                  break;
    case NotFound: printf("\n");
                  printf("\n Row not found!");
                  break;
    case MultipleRows: printf("\n");
                      printf("WARNING: More than one row qualifies!")
                      DisplayUpdate();
                      break;
    default:     SQLStatusCheck();
                  break;
    }
    :
    }
    :
int DisplayUpdate()
{
    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 != 0K) SQLStatusCheck();
    :
    }

```

Figure 4-2. Explicit Status Checking Procedures (page 2 of 5)

```

int DeleteData()
{
    This function verifies that the row(s) to be deleted
    exist, then invokes the function DisplayDelete to delete
    the row(s).

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

    switch(sqlca.sqlcode) {

    case OK:           DisplayDelete();
                      break;

    case NotFound:    printf("\n");
                      printf("\n Row not found!");
                      break;

    case MultipleRows: printf("\n");
                      printf("WARNING: More than one row qualifies!");
                      DisplayDelete();
                      break;

    default:          SQLStatusCheck();
                      break;

    }

    :
}
:
int DisplayDelete()
{
    Statements that verify that the deletion should
    actually occur appear here.

    EXEC SQL DELETE FROM PurchDB.Parts
           WHERE PartNumber = :PartNumber;

    if (sqlca.sqlcode != OK) SQLStatusCheck();

    :
}

```

**Figure 4-2. Explicit Status Checking Procedures (page 3 of 5)**

```

int SelectData()
{
    Statements that prompt for a partnumber appear here.

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

    switch(sqlca.sqlcode) {

    case OK:           DisplayRow();
                      break;

    case NotFound:    printf("\n");
                      printf("\n Row not found!");
                      break;

    case MultipleRows: printf("\n");
                      printf("WARNING: More than one row qualifies!");
                      DisplayDelete();
                      break;

    default:          SQLStatusCheck();
                      break;

    }
    :
}
:
int SQLStatusCheck()
{

if ((sqlca.sqlcode == DeadLock) || (sqlca.sqlcode=NoMemory)) {
    if (TryCounter == TryLimit) {
        SQLCommandDone = TRUE;
        printf("\n Could not complete transaction.  Try again if you
                                                    want.");
    }
else
    SQLCommandDone = FALSE;
}
}

```

Figure 4-2. Explicit Status Checking Procedures (page 4 of 5)

```

else
{
Abort = FALSE;
if (sqlca.sqlwarn[6] == 'W') {      /* The transaction was rolled back
                                     due to other than deadlock or
                                     shared memory problems. */

Abort = TRUE;
do {
EXEC SQL SQLEXPLAIN :SQLMessage;
printf("%s\n",SQLMessage);

} while (sqlca.sqlcode != 0);
}
if (Abort) {
TerminateProgram();
}
else
SQLCommandDone = TRUE;
}
}

.
if (SQLCommandDone) {
.
/* Prompt user for a part number. */
.
TryCounter = 0;
TryLimit = 3;
.
/* A transaction is started. */
.
TryCounter = TryCounter + 1;

} /* End SQLStatusCheck Procedure */

```

**Figure 4-2. Explicit Status Checking Procedures (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 4-2. 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[2] 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
```

Cursor operations:

```
FETCH
UPDATE WHERE CURRENT
DELETE WHERE CURRENT
```

The sqlerrd[2] value can be used in these cases only when sqlcode does not contain a negative number. When sqlcode is 0, sqlerrd[2] is *always* equal to 1 for SELECT, FETCH, UPDATE WHERE CURRENT, and DELETE WHERE CURRENT operations. Sqlerrd[2] may be *greater* than 1 if more than one row qualifies for an INSERT, UPDATE, or DELETE operation. When sqlcode is 100, sqlerrd[2] 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[2]` contains the number of rows that could be successfully retrieved or inserted *before* an error occurred. If `sqlcode` is 0, `sqlerrd[2]` 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 follow.

**INSERT, UPDATE, and DELETE Operations.** The example in Figure 4-2 could be modified to display the number of rows inserted, updated, or deleted by using `sqlerrd[2]`. In the case of the update operation, for example, the actual number of rows updated could be displayed after the UPDATE command is executed:

```
.
.
.
int DisplayUpdate()
{
    Code that prompts the user for new data appears here.
    EXEC SQL UPDATE PurchDB.Parts
           SET PartName = :PartName,
           SalesPrice = :SalesPrice,
           WHERE PartNumber = :PartNumber;

    switch(sqlca.sqlcode) {
    case OK:    NumberOfRows = sqlerrd[2];
               printf("The number of rows updated was: %d\n" NumberOfRows);
               break;
    default:   printf("\n No rows could be updated!");
               SQLStatusCheck();
               break;
    }
}
```

If the UPDATE command is successfully executed, `sqlcode` is 0 and `sqlerrd[2]` contains the number of rows updated. If the UPDATE command cannot be successfully executed, `sqlcode` contains a negative number and `sqlerrd[2]` contains a 0.

**BULK Operations.** When using the BULK SELECT, BULK FETCH, or BULK INSERT commands, you can use the sqlerrd[2] value in 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 4-3, the value in sqlerrd[2] 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[2] 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[2] is used to control the number of times function DisplayRow is executed.



```

#define      OK          0
#define      NotFound   100
#define      MaximumRows 200

      /*Begin Host Variable Declarations */
EXEC SQL Begin Declare Section;
struct {
    char          PartNumber[17];
    char          PartName[31];
    double        SalesPrice;
} PartsTable[MaximumRows];
char          SQLMessage[133];
EXEC SQL End Declare Section;
      /* End Host Variable Declarations */

sqlca : sqlca_type; /* SQL Communication Area */

int          i;
int          NumberOfRows;

int BulkSelect()
{
    EXEC SQL BULK SELECT PartNumber,
                        PartName,
                        SalesPrice
                        INTO :PartsTable
                        FROM PurchDB.Parts;

    switch(sqlca.sqlcode) {
        case OK:          DisplayTable();
                        break;
        case NotFound:    printf("\n");
                        printf("\n No rows qualify for this operation!");
                        break;
        default:          NumberOfRows = sqlerrd[2];
                        printf("\nOnly %d rows were retrieved",NumberOfRows);
                        printf("\n before an error occurred!");
                        DisplayTable();
                        SQLStatusCheck();
                        break;
    }
}

```

Figure 4-3. Determining Number of Rows Processed After a BULK SELECT

```

.
.
.
int DisplayTable()
{
    if (sqlerrd[2] == MaximumRows) 2
    {
        printf("\n");
        printf("\nWARNING:  There may be additional rows that qualify!");
    }
    The column headings are displayed here.
    for (i = 0; i < sqlerrd[2]; i++)
        DisplayRow(); 3
    printf("\n");
}

int DisplayRow()
{
    printf(PartNumber[i], "%s\n |");
    printf(PartName[i], "%s\n |");
    printf(SalesPrice[i], "%.2f\n |");
}

```

**Determining Number of Rows Processed After a BULK SELECT (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 can be used to determine when all rows in an active set have been fetched:

```
int FetchRow()
{
    EXEC SQL FETCH  CURSOR1
           INTO  :PartNumber,
                :PartName,
                :SalesPrice;

    switch(sqlca.sqlcode) {
        case OK:           DisplayRow();
                           break;
        case NotFound:     DoneFetch = TRUE;
                           printf("\n Row not found or no more rows!");
                           break;
        default:           SQLStatusCheck();
                           break;
    }
}
.
.
.
EXEC SQL OPEN CURSOR1;
.
.
.
do {
    FetchRow();
} while (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 has been fetched from the active set, will set sqlcode to NotFound (defined as 100 in the declaration part). If no rows qualify for the active set, sqlcode is NotFound the first time function FetchRow is executed.

## Determining When More Than One Row Qualifies

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 function is not invoked; instead a warning message is displayed:

```
int UpdateData()
{
    This function verifies that the row(s) to be changed
    exist, then invokes the function DisplayUpdate to accept
    new data from the user.

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

    switch(sqlca.sqlcode) {
    case OK:          DisplayUpdate();
                    break;
    case NotFound:   printf("\n");
                    printf("\n Row not found!");
                    break;
    case MultipleRows: printf("\n");
                    printf("\n WARNING: More than one row qualifies!");
                    DisplayUpdate();
                    break;
    default:         SQLStatusCheck();
                    break;
    }
}
```

---

**Note** The PARTS table in the sample database has a unique index on PARTNUMBER, so a test for multiple rows is not required. This test is useful for the ORDERITEMS table which does not have a unique index.

---

## 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` routine is executed to display any messages, and the program is terminated.

```
if (sqlca.sqlcode = -14046)
    COMMIT WORK;
    SQLStatusCheck();
    TerminateProgram();
```

## 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 transaction is rolled back 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)
    ROLLBACK WORK TO :SavePoint;
    COMMIT WORK;
    SQLStatusCheck();
    TerminateProgram();
```

## 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 retrieved 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 'PartsDBE';
if (sqlca.sqlcode = -2300)
    SQLStatusCheck();
    TerminateProgram();
```



## 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 satisfy the following conditions:

- 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, do not require use of commands such as PREPARE, DESCRIBE, and EXECUTE IMMEDIATE.

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.

### SELECT

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 needed 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:

```
#define MultipleRows    -10002
.
.
.
int GetRow()
{
.
.
.
    The SELECT command is executed here.

    if (sqlca.sqlcode == MultipleRows) {
        printf("\n 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. 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 insures that no more than one null value can exist for that column.

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 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 prompted whether to change the bin number. 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;
```

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 wild card 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 character 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 wild card character or the escape character itself. Complete syntax is presented in the *ALLBASE/SQL Reference Manual* .

- To prohibit the multiple-row changes possible if multiple rows qualify for an UPDATE or DELETE operation. If multiple rows qualify for the SELECT operation, the UPDATE or DELETE command 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 at a time 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 the user with the new number and prompt for information describing the new order.*

## INSERT

In simple data manipulation, you use the INSERT command to either insert a single row or copy one or more rows into one 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 Column Names 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 syntax 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 table 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;
```

## UPDATE

In simple data manipulation, you use UPDATE command syntax to change data in one or more columns:

```
UPDATE TableName
    SET ColumnName = 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, for example:

```
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 zero to the indicator variable for this column, ContactNameInd:

*The program prompts the user for new values for the four columns.*

```
printf ("\n Enter Vendor Street > ");
getline(VendorStreet);
printf ("\n Enter Vendor City > ");
getline(VendorCity);

printf ("\n Enter Vendor State > ");
getline(VendorState);

printf ("\n Enter Vendor Zip Code > ");
getline(VendorZipCode);

printf ("\n Enter Contact Name (0 for null) > ");
getline(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] == '0') {
    ContactNameInd = -1;
}
else
    ContactNameInd = 0;
```

## DELETE

In simple data manipulation, you use DELETE command syntax to delete one or more rows from a table:

```
DELETE FROM TableName
        WHERE SearchCondition
```

The WHERE clause specifies a *SearchCondition* that 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 involve 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:

- If 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) {
    EXEC SQL ROLLBACK WORK;
}
```

---

## Sample Program Using Simple DML Commands

The flow chart shown in Figure 5-1 summarizes the functionality of program `cex7`. This program uses the four simple data manipulation commands to operate on the `PurchDB.Vendors` table. A function menu determines 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 `cex7` appears in Figure 5-2, and the source code in Figure 5-3.

Function `ConnectDBE` starts a DBE session (51). This function executes the `CONNECT` command (2) for the sample DBEnvironment, `PartsDBE`. The operation performed next depends on the number entered when a function menu is displayed (52):

- The program terminates if 0 is entered.
- Function `Select` is executed if 1 is entered.
- Function `Update` is executed if 2 is entered.
- Function `Delete` is executed if 3 is entered.
- Function `Insert` is executed if 4 is entered.

The `Select` function (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, function `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 (17). Function `EndTransaction` 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 (15) is displayed and the transaction is terminated (17). The user is then re-prompted for a vendor number or a 0.
- If the `SELECT` command execution results in an error condition, function `SQLStatusCheck` is executed (16). This function executes `SQLEXPLAIN` (1) to display all error messages. Then the transaction is terminated (17) and the user re-prompted for a vendor number or a 0.
- If the `SELECT` command can be successfully executed, the `DisplayRow` function (13) is executed to display the row. This function examines the null indicators for each of the three potentially null columns (`ContactName`, `PhoneNumber`, and `VendorRemarks`). If any null indicator contains a value not equal to 0 (3), a message indicating that the value is null is displayed. After the row is completely displayed, the transaction is terminated (17) and the user re-prompted for a vendor number or a 0.



The Update function (23) lets the user UPDATE the value of a column only if it contains a null value. The function prompts (24) for a vendor number or a 0. If a 0 is entered, the function menu is re-displayed. If a vendor number is entered, function BeginTransaction is executed (25). Then a SELECT command is executed (26) 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 (28) is displayed and the transaction is terminated (31). 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 (29) is displayed and the transaction is terminated (31). The user is then re-prompted for a vendor number or a 0.
- If the SELECT command execution results in an error condition, function SQLStatusCheck is executed (30). Then the transaction is terminated (31) and the user re-prompted for a vendor number or a 0.
- If the SELECT command can be successfully executed, function DisplayUpdate (27) is executed. This function executes function DisplayRow to display the row retrieved (18). Function AnyNulls is then executed to determine whether the row contains any null values. This boolean function evaluates to TRUE (6) if the indicator variable for any of the three potentially null columns contains a non-zero value.

If function AnyNulls evaluates to FALSE, a message is displayed (7) and the transaction is terminated (31); 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 (19). A negative null indicator means the column contains a null value, and the user is prompted for a new value (20). If the user enters a 0, the program assigns a -1 to the null indicator (21) so that when the UPDATE command (22) 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 (22) command is executed, the transaction is terminated (31) and the user re-prompted for a vendor number or a 0.

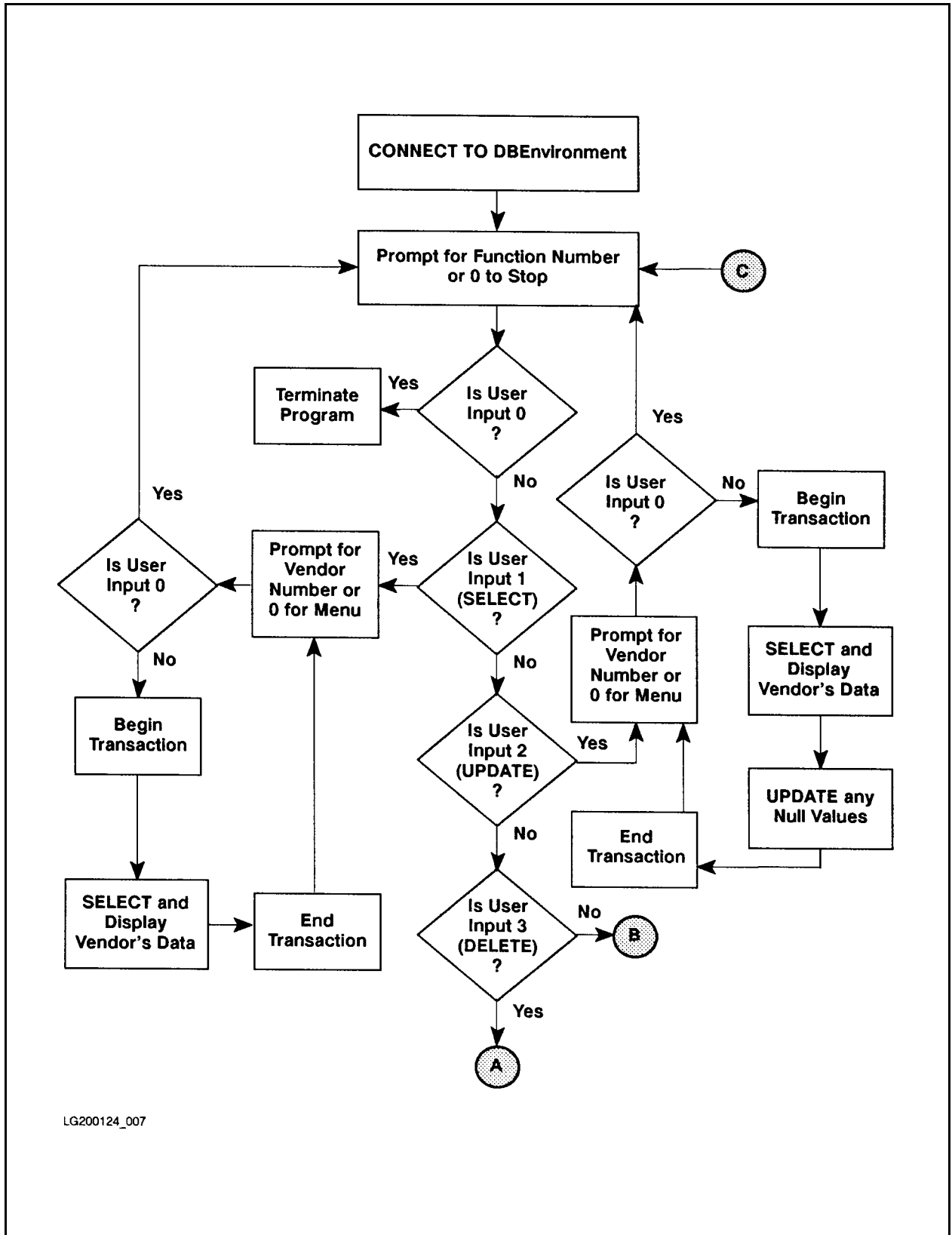
Function Delete (35) lets the user DELETE one row. The function prompts for a vendor number or a 0 (36). If a 0 is entered, the function menu is re-displayed. If a vendor number is entered, function BeginTransaction is executed (37). Then a SELECT command is executed to retrieve all data for the vendor specified from PurchDB.Vendors (38). The sqlca.sqlcode returned is examined to determine the next action:

- If no rows qualify for the SELECT operation, a message (40) is displayed and the transaction is terminated (43). 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 (41) is displayed and the transaction is terminated (43). The user is then re-prompted for a vendor number or a 0.
- If the SELECT command execution results in an error condition, function SQLStatusCheck is executed (42). Then the transaction is terminated (43) and the user re-prompted for a vendor number or a 0.

- If the SELECT command can be successfully executed, the DisplayDelete function (39) is executed. This function executes function DisplayRow to display the row retrieved (32). Then the user is asked whether she wants to actually delete the row (33). If not, the transaction is terminated (43) and the user re-prompted for a vendor number or a 0. If so, the DELETE command (34) is executed before the transaction is terminated (43) and the user re-prompted.

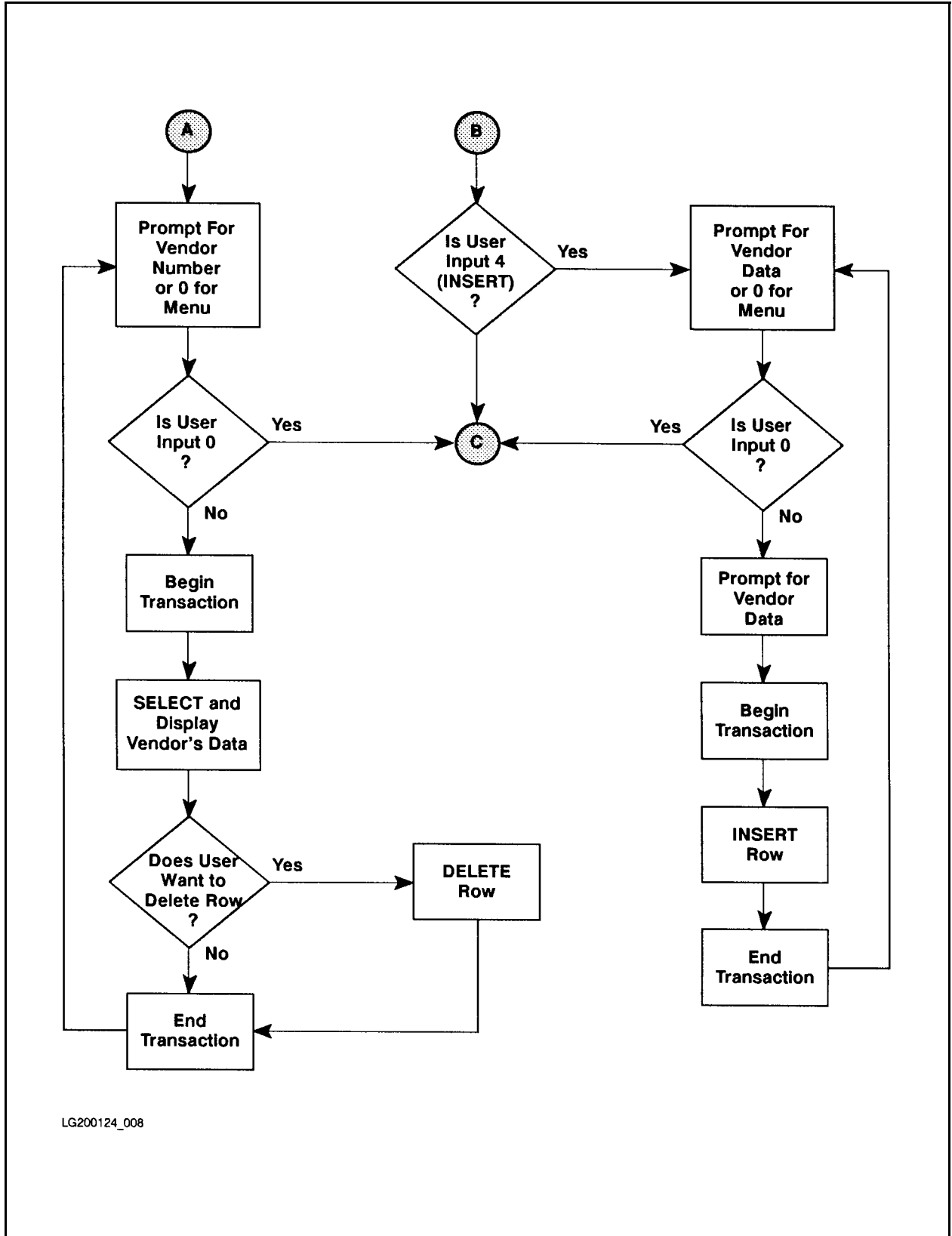
The Insert function (44) lets the user INSERT one row. The function prompts for a vendor number or a 0 (45). If a 0 is entered, the function menu is re-displayed. If a vendor number is entered, the user is prompted for values for each column. The user can enter a 0 to specify a null value for potentially null columns (46); to assign a null value, the program assigns a -1 to the appropriate null indicator (47). After a transaction is started (48), an INSERT command (49) is used to insert a row containing the specified values. After the INSERT operation, the transaction is terminated (50), 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 function ReleaseDBE (53). This function executes the RELEASE command (3).



LG200124\_007

Figure 5-1. Flow Chart of Program cex7



LG200124\_008

Figure 5-1. Flow Chart of Program cex7 (page 2 of 2)

```
Program for Simple Data Manipulation of Vendors Table - cex7

Connect to PartsDBE

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 > 4

*** Function to INSERT rows into PurchDB.Vendors ***

Enter Vendor Number to INSERT into Vendors Table or 0 for MENU > 9016

Enter Vendor Name > Wolfe Works

Enter Contact Name or a 0 for null > Stanley Wolfe

Enter Phone Number or a 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 or a 0 for null > 0

Begin Work
INSERT row into PurchDB.Vendors
Commit Work

Enter Vendor Number to INSERT into Vendors Table 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 5-2. Runtime Dialog of Program cex7

```
*** Function to SELECT a row from the Vendors Table ***

Enter Vendor Number to SELECT from Vendors Table 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 to SELECT from Vendors Table 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

*** Function to UPDATE rows in PurchDB.Vendors ***

Enter Vendor Number to UPDATE within Vendors Table 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 or 0 for null > can expedite shipments
Commit Work
```

Figure 5-2. Runtime Dialog of Program cex7 (page 2 of 3)

```
Enter Vendor Number to UPDATE within Vendors Table 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

*** Function to DELETE rows from PurchDB.Vendors ***

Enter Vendor Number to DELETE from Vendors Table 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 to DELETE from Vendors Table 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 5-2. Runtime Dialog of Program cex7 (page 3 of 3)

```

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* This program illustrates the use of SQL's data manipulation      */
/* commands for simple operations.  It uses the UPDATE             */
/* command with indicator variables to update any row in the      */
/* Vendors table that contains null values.  It also uses indicator */
/* variables in conjunction with SELECT and INSERT commands.      */
/* The DELETE command is also illustrated.                        */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

typedef int boolean;

boolean    Abort;
int        response1;
char       response2[2];

#include <stdio.h>

#define     OK                0
#define     NotFound          100
#define     MultipleRows     -10002
#define     DeadLock         -14024
#define     FALSE            0
#define     TRUE              1

sqlca_type sqlca;    /* SQL Communication Area */

        /* Begin Host Variable Declarations */
EXEC SQL BEGIN DECLARE SECTION;
int      VendorNumber;
char     VendorName[31];
char     ContactName[31];
sqlind   ContactNameInd;
char     PhoneNumber[16];
sqlind   PhoneNumberInd;
char     VendorStreet[31];
char     VendorCity[21];
char     VendorState[3];
char     VendorZipCode[11];
char     VendorRemarks[61];
sqlind   VendorRemarksInd;
char     SQLMessage[133];
EXEC SQL END DECLARE SECTION;
        /* End Host Variable Declarations */

```

**Figure 5-3. Program cex7: Using INSERT, UPDATE, SELECT and DELETE**



```

int getline(linebuff) /* Function to get a line of characters */
char linebuff[80];
{
while (strlen(gets(linebuff)) == 0);
} /* End of function to get a line of characters */

int SQLStatusCheck() /* Function to Display Error Messages */ 16
{
Abort = FALSE;
if (sqlca.sqlcode < DeadLock) Abort = TRUE;

do {
EXEC SQL SQLEXPLAIN :SQLMessage; 1
printf("%s\n",SQLMessage);
} while (sqlca.sqlcode != 0);

if (Abort) ReleaseDBE();
} /* End SQLStatusCheck Function */

boolean ConnectDBE() /* Function to Connect to PartsDBE */ 51
{
boolean ConnectDBE;
printf("\n Connect to PartsDBE");
EXEC SQL CONNECT TO 'PartsDBE'; 2

ConnectDBE = TRUE;
if (sqlca.sqlcode != OK) {
ConnectDBE = FALSE;
SQLStatusCheck();
} /* End if */
return (ConnectDBE);
} /* End of ConnectDBE Function */

int ReleaseDBE() /* Function to Release PartsDBE */ 53
{
printf("\n Release PartsDBE");
printf("\n");
EXEC SQL RELEASE; 3

if (sqlca.sqlcode != OK) SQLStatusCheck();

} /* End ReleaseDBE Function */

```

Figure 5-3. Program cex7: Using INSERT, UPDATE, SELECT and DELETE (page 2 of 12)

```

boolean BeginTransaction()    /* Function to Begin Work */    (11)
{
boolean BeginTransaction;
printf("\n");
printf("\n Begin Work");
EXEC SQL BEGIN WORK;    (4)
if (sqlca.sqlcode != OK) {
    BeginTransaction = FALSE;
    SQLStatusCheck();
    ReleaseDBE();
}
else
    BeginTransaction = TRUE;

return (BeginTransaction);

} /* End BeginTransaction Function */

int EndTransaction() /* Function to Commit Work */    (17)
{
printf("\n");
printf("\n Commit Work");
EXEC SQL COMMIT WORK;    (5)
if (sqlca.sqlcode != OK) {
    SQLStatusCheck();
    ReleaseDBE();
}

} /* End EndTransaction Function */

boolean AnyNulls()    /* Function to test a row for null values */
{
boolean AnyNulls;
AnyNulls = TRUE;
if ((ContactNameInd == 0)&(PhoneNumberInd == 0)&(VendorRemarksInd == 0)) {
    /* All columns that might be null contain non-null values */    (6)
    printf("\n No null values exist for this vendor!");    (7)
    AnyNulls = FALSE;
}
return (AnyNulls);

} /* End of AnyNulls Function */

```

Figure 5-3. Program cex7: Using INSERT, UPDATE, SELECT and DELETE (page 3 of 12)

```

int DisplayRow() /* Function to Display Parts Table Rows */ 13
{
    printf("\n");
    printf("Vendor Number:      %10d\n", VendorNumber);
    printf("Vendor Name:         %s\n", VendorName);

    if (ContactNameInd != 0) { 8
        printf("Contact Name:      is NULL \n");
    }
    else
        printf("Contact Name:         %s\n", ContactName);

    if (PhoneNumberInd != 0) { 8
        printf("Phone Number:       is NULL \n");
    }
    else
        printf("PhoneNumber:          %s\n", PhoneNumber);

    printf("VendorStreet:          %s\n", VendorStreet);
    printf("VendorCity:           %s\n", VendorCity);
    printf("VendorState:          %s\n", VendorState);
    printf("VendorZipCode:         %s\n", VendorZipCode);
    if (VendorRemarksInd != 0) { 8
        printf("Vendor Remarks:      is NULL \n");
    }
    else
        printf("VendorRemarks:        %s\n", VendorRemarks);

} /* End of DisplayRow */

int Select() /* Function to Query Parts Table */ 9
{
    printf("\n");
    printf("\n *** Function to SELECT a row from the Vendors table. ***");
    printf("\n");

    do {
        printf("\nEnter Vendor Number to SELECT from Vendors Table or 0 for
MENU>");
        while (scanf("%d%c", &VendorNumber) == 0; 10

            if (VendorNumber != 0) {

                BeginTransaction(); 11

```

Figure 5-3. Program cex7: Using INSERT, UPDATE, SELECT and DELETE (page 4 of 12)

```

printf("\n");
printf("\n SELECT * from PurchDB.Vendors");
EXEC SQL SELECT  VendorNumber,
                 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;

switch(sqlca.sqlcode) {
case OK:          DisplayRow();
                  break;
case NotFound:   printf("\n");
                  printf("\n Row not found!");
                  break;
case MultipleRows: printf("\n");
                  printf("\nWARNING: More than one row qualifies!");
                  break;
default:         SQLStatusCheck();
                  break;
} /* End switch */

EndTransaction();

} /* End if */
} while (VendorNumber != 0);
} /* End of Select Function */

```

Figure 5-3. Program cex7: Using INSERT, UPDATE, SELECT and DELETE (page 5 of 12)

```

int DisplayUpdate() /* Display & Update row in Parts Table */ (27)
{
DisplayRow(); (18)
if (AnyNulls) {
    if (ContactNameInd != 0) { (19)
        printf("\n");
        printf("\n Enter new Contact Name or 0 for NULL > "); (20)
        getline(ContactName);
    }
    if (PhoneNumberInd != 0) { (19)
        printf("\n");
        printf("\n Enter new Phone Number of 0 for NULL > "); (20)
        getline(PhoneNumber);
    }
    if (VendorRemarksInd != 0) { (19)
        printf("\n");
        printf("\n Enter new Vendor Remarks or 0 for NULL > "); (20)
        getline(VendorRemarks);
    }

    if (ContactName[0] == '0') ContactNameInd = -1; (21)
    else
        ContactNameInd = 0;

    if (PhoneNumber[0] == '0') PhoneNumberInd = -1; (21)
    else
        PhoneNumberInd = 0;

    if (VendorRemarks[0] == '0') VendorRemarksInd = -1; (21)
    else
        VendorRemarksInd = 0;

printf("\n");
printf("\n UPDATE the Vendors Table");
EXEC SQL UPDATE PurchDB.Vendors (22)
    SET ContactName = :ContactName :ContactNameInd,
        PhoneNumber = :PhoneNumber :PhoneNumberInd,
        VendorRemarks = :VendorRemarks :VendorRemarksInd
    WHERE VendorNumber = :VendorNumber;

if (sqlca.sqlcode != 0K ) SQLStatusCheck();

} /* End if AnyNulls */
} /* End of DisplayUpdate Function */

```

Figure 5-3. Program cex7: Using INSERT, UPDATE, SELECT and DELETE (page 6 of 12)

```

int Update()      /* Update a row within the Parts Table */      (23)
{
printf("\n");
printf("\n *** Function to UPDATE rows in PurchDB.Vendors ***");
printf("\n");

do {
printf("\n Enter Vendor Number to UPDATE in Vendors Table or 0 for MENU >");
while (scanf("%d%c", &VendorNumber) == 0);      (24)

if (VendorNumber != 0) {

    BeginTransaction();      (25)

    printf("\n");
    printf("\n 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;

    if (sqlca.sqlcode == 0K)
        DisplayUpdate();      (27)
    else if (sqlca.sqlcode == NotFound)
        printf("\n Row not found!");      (28)
    else if (sqlca.sqlcode == MultipleRows)
        printf("\n WARNING: More than one row qualifies!");      (29)
    else
        SQLStatusCheck();      (30)
}
}

```

Figure 5-3. Program cex7: Using INSERT, UPDATE, SELECT and DELETE (page 7 of 12)

```

EndTransaction();
} /* End if */
} while (VendorNumber != 0);
} /* End of Update Function */

int DisplayDelete() /* Display & optionally Delete Rows */
{
DisplayRow();

printf("\n");
printf("\n Is it OK to DELETE this row (N/Y) ? > ");
scanf("%s",response2);
if ((response2[0] == 'Y') || (response2[0] == 'y')) {
printf("\n");
printf("\n DELETE Row from PurchDB.Vendors Table");
EXEC SQL DELETE FROM PurchDB.Vendors
WHERE VendorNumber = :VendorNumber;

if (sqlca.sqlcode != 0K) SQLStatusCheck();
}
} /* End of DisplayDelete */

int Delete() /* Function to Delete a row from the Parts Table */
{
printf("\n");
printf("\n *** Function to DELETE rows from PurchDB.Vendors ***");
printf("\n");

do {
printf("\nEnter Vendor Number to DELETE from Vendors Table or 0 for MENU>");
while(scanf("%d%c", &VendorNumber) == 0);
if (VendorNumber != 0) {

BeginTransaction();

```

Figure 5-3. Program cex7: Using INSERT, UPDATE, SELECT and DELETE (page 8 of 12)

```

printf("\n");
printf("\n SELECT * from PurchDB.Vendors");
EXEC SQL SELECT VendorNumber,
                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;

if (sqlca.sqlcode == OK)
    DisplayDelete();
else if (sqlca.sqlcode == NotFound)
    printf("\n Row not found!");
else if (sqlca.sqlcode == MultipleRows)
    printf("\n WARNING: More than one row qualifies!");
else
    SQLStatusCheck();

EndTransaction ();

} /* End if */

} while (VendorNumber != 0);

} /* End of Delete */

int Insert() /* Insert a row into the Parts Table */
{
printf("\n");
printf("\n *** Function to INSERT rows into PurchDB.Vendors ***");
printf("\n");

```

Figure 5-3. Program cex7: Using INSERT, UPDATE, SELECT and DELETE (page 9 of 12)



```

do {
VendorNumber = 0;
printf("\nEnter Vendor Number to INSERT into Vendors Table or 0 for MENU>");
while(scanf("%d%c", &VendorNumber) == 0); 45

if (VendorNumber != 0) {

    printf("\n Enter Vendor Name > ");
    getline(VendorName);

    printf("\n Enter Contact Name or a 0 for NULL > "); 46
    getline(ContactName);

    if (ContactName[0] == '0') ContactNameInd = -1; 47
    else
        ContactNameInd = 0;

    printf("\n Enter Phone Number or a 0 for NULL > "); 46
    getline(PhoneNumber);

    if (PhoneNumber[0] == '0') PhoneNumberInd = -1 47
    else
        PhoneNumberInd = 0;

    printf("\n Enter Vendor Street > ");
    getline(VendorStreet);

    printf("\n Enter Vendor City > ");
    getline(VendorCity);

    printf("\n Enter Vendor State > ");
    getline(VendorState);

    printf("\n Enter Vendor Zip Code > ");
    getline(VendorZipCode);

    printf("\n Enter Vendor Remarks or a 0 for NULL > "); 46
    getline(VendorRemarks);

    if (VendorRemarks[0] == '0') VendorRemarksInd = -1; 47
    else
        VendorRemarksInd = 0;

    BeginTransaction(); 48
}
}

```

**Figure 5-3. Program cex7: Using INSERT, UPDATE, SELECT and DELETE (page 10 of 12)**

```

printf("\n");
printf("\n INSERT row into PurchDB.Vendors");
EXEC SQL INSERT
        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);

```

49

```

if (sqlca.sqlcode != OK) SQLStatusCheck();
EndTransaction();

```

50

```

} /* End if */
} while (VendorNumber != 0);

} /* End of Insert */

```

```

main()          /* Beginning of program */
{
printf("\n Program for Simple Data Manipulation of Vendors Table-cex7")
printf("\n");

```

```

if (ConnectDBE()) {

```

51

Figure 5-3. Program cex7: Using INSERT, UPDATE, SELECT and DELETE (page 11 of 12)

```

do {
printf("\n");
printf("\n 1 . . .SELECT rows from PurchDB.Vendors Table");
printf("\n 2 . . .UPDATE rows with NULL values in PurchDB.Vendors Table");
printf("\n 3 . . .DELETE rows from PurchDB.Vendors Table");
printf("\n 4 . . .INSERT rows into PurchDB.Vendors Table");
printf("\n");
printf("\n Enter choice or 0 to STOP > ");
scanf("%2d%c", &response1);

if (response1 != 0) {

switch (response1) {
case 1: Select();
break;
case 2: Update();
break;
case 3: Delete();
break;
case 4: Insert();
break;
default: printf("\n Enter 0-4 only, please!");
break;
} /* switch */

} /* End if response1 */

} while ( response1 != 0);

ReleaseDBE();
} /* End if Connect */

else
printf("\n Error: Cannot Connect to PartsDBE!\n");

} /* End of Program */

```

Figure 5-3. Program cex7: Using INSERT, UPDATE, SELECT and DELETE (page 12 of 12)



## 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 (for example, inventory adjustments) may also find this technique useful.

This chapter presents:

- SQL Cursor Commands.
- Transaction Management for Cursor Operations
- Sample Program Using Cursor Operations.

The emphasis in this chapter is on **FETCH**ing one row at a time. For an example of using the **FETCH** command with the **BULK** option, see the “**BULK FETCH**” section of the chapter, “**BULK Table Processing**.”

---

### 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 names a cursor and associates it with a particular SELECT command, as shown in the following syntax:

```
DECLARE CursorName
      [IN DBEFileSetName]
CURSOR FOR
      SelectCommand
      [FOR UPDATE OF ColumnName [, ColumnName...]]
```

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 intend to 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 SELECT command. Updating a current row is discussed fully later in this chapter under “Update Where Current.”

The SELECT command for cursor declarations that do not include the FOR UPDATE clause can consist of any of the SELECT command clause *except* the INTO clause, as shown in the following syntax:

```
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, and 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 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. The section is executed when the `OPEN` command is encountered. Because the `DECLARE CURSOR` command is not executed at run time, you do not need to perform error status checking in your program following this command.

## OPEN

The `OPEN` command allocates internal buffer space and defines the active set, as shown in the following syntax:

```
OPEN CursorName [KEEP CURSOR [ [WITH LOCKS ] ] ]
                        [ [NOLOCKS ] ] ]
```

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.

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.

## FETCH

The `FETCH` command defines a current row and delivers the row into output host variables:

```
FETCH CursorName INTO OutputHostVariables
```

Remember to include indicator variables when one or more columns in the query result may contain a null value, for example:

```
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 the current row. After the last row in the query result has been fetched, `ALLBASE/SQL` sets `sqlca.sqlcode` to 100. `ALLBASE/SQL` also sets `sqlca.sqlcode` to 100 if no rows qualify for the active set. You should test for an `sqlca.sqlcode` value of 100 after each execution of the `FETCH` command to determine whether to re-execute the command.

```

#define TRUE 0
#define FALSE 1
.
.
.
int GetARow()
{
int DoFetch;
.
.
.
DoFetch = TRUE;
do {
.
.
The FETCH command appears here.
.
switch (sqlca.sqlcode) {
case 0:      DisplayRow();
              break;
case 100:    DoFetch = FALSE;
              CloseCursor();
              CommitWork();
              break;
default:     DoFetch = FALSE;
              SQLStatusCheck();
              CloseCursor();
              RollBack();
              break;
            }
} while (DoFetch == TRUE);
}

```

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:

- DISTINCT clause in the select list.
- Aggregate function in the select list.
- FROM clause with more than one table.
- ORDER BY clause.
- GROUP BY clause.

The UPDATE WHERE CURRENT command identifies the active set to be updated by naming the cursor and the column(s) to be updated, as shown in the following syntax:

```
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, for example:

```
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;
```

*In this case, 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 the previous example, the order of the rows in the query result is not important. Therefore the SELECT command associated with the 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 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 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 that your UPDATE commands change only one row (the current row) or perform the UPDATE operations outside the FETCH loop.

## 6-6 Processing with Cursors

## 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:

- DISTINCT clause in the select list.
- Aggregate function in the select list.
- FROM clause with more than one table.
- ORDER BY clause.
- 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 with 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, you use CLOSE command 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 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 followed by COMMIT WORK to explicitly close any open cursors before ending a transaction.

---

## Transaction Management for Cursor Operations

The time at which ALLBASE/SQL obtains locks during cursor processing depends on the type of index ALLBASE/SQL uses 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 locks not associated with the kept cursor and begins a new transaction without changing the current 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 the page of data corresponding to the current cursor position are either held until the transaction ends (the default) or released depending on whether you specify WITH LOCKS or WITH NOLOCKS. (Pages held include data and system pages.)

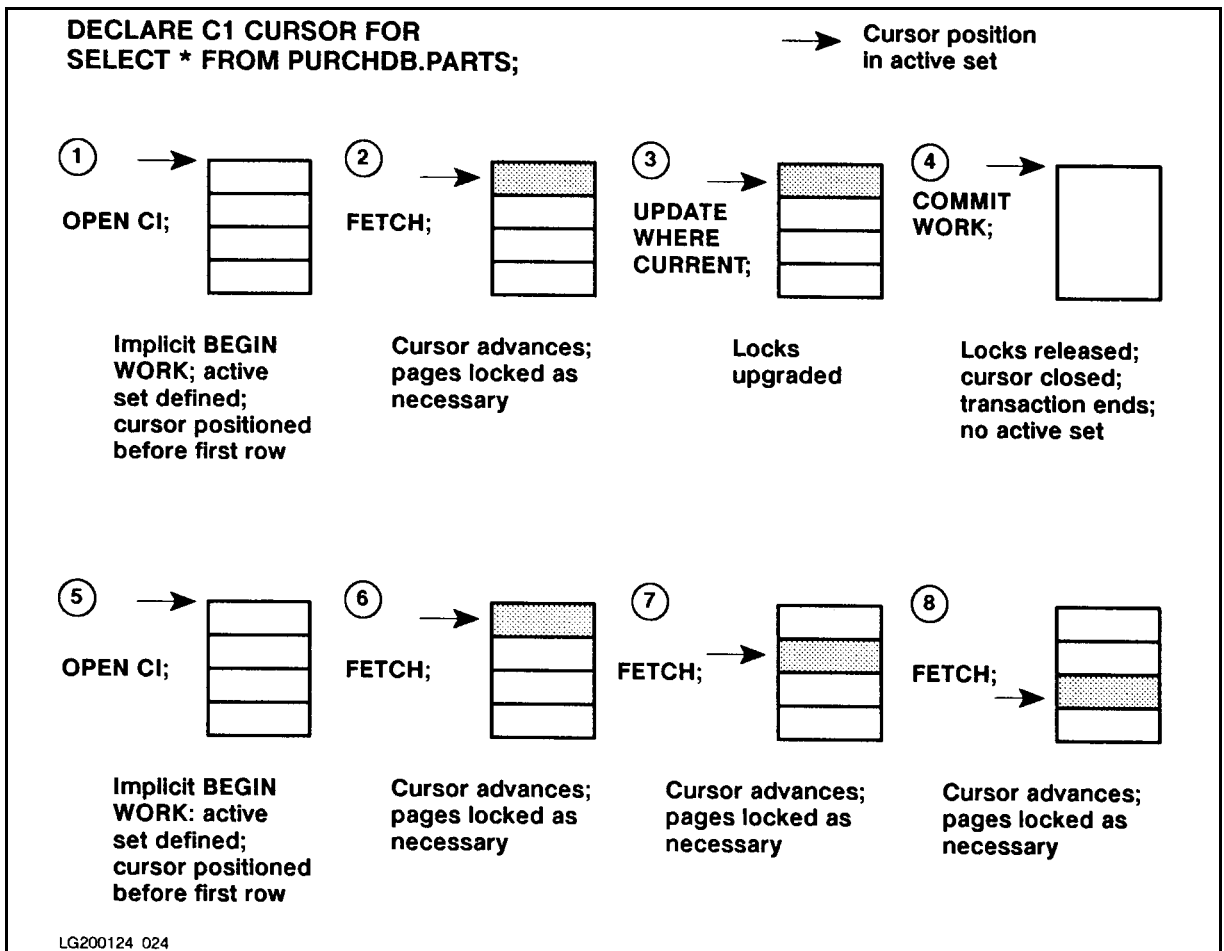
### KEEP CURSOR and Isolation Levels

The KEEP CURSOR option retains the current isolation level (RR, CS, or RC) 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. With the READ COMMITTED isolation level, no locks are maintained across transactions because locks are released at the end of the FETCH. Therefore, KEEP CURSOR WITH LOCKS does not make sense at a RC isolation level.

For additional information on isolation levels, refer to the chapter “Concurrency Control through Locks and Isolation Levels” in the *ALLBASE/SQL Reference Manual* .

### OPEN Command Without KEEP CURSOR

Figure 6-1 shows the operation of cursors when you do *not* select the KEEP CURSOR option.



**Figure 6-1. 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, shared 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 6-2 shows the effect of the `KEEP CURSOR WITH LOCKS`.

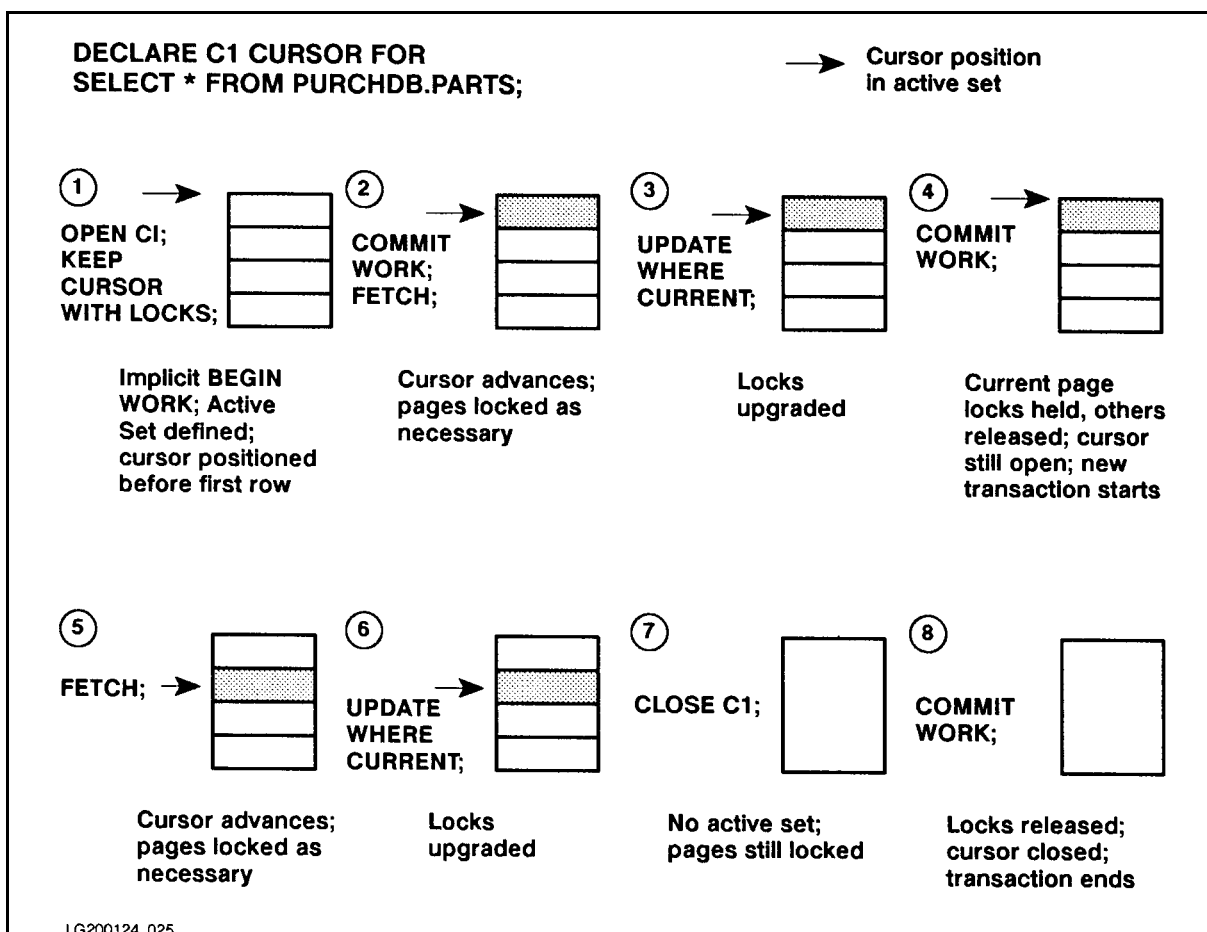


Figure 6-2. Cursor Operation Using KEEP CURSOR WITH LOCKS

## 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 position 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 message `TABLE NOT FOUND (DBERR 137)`.

Figure 6-3 shows the effect of `KEEP CURSOR WITH NOLOCKS`.

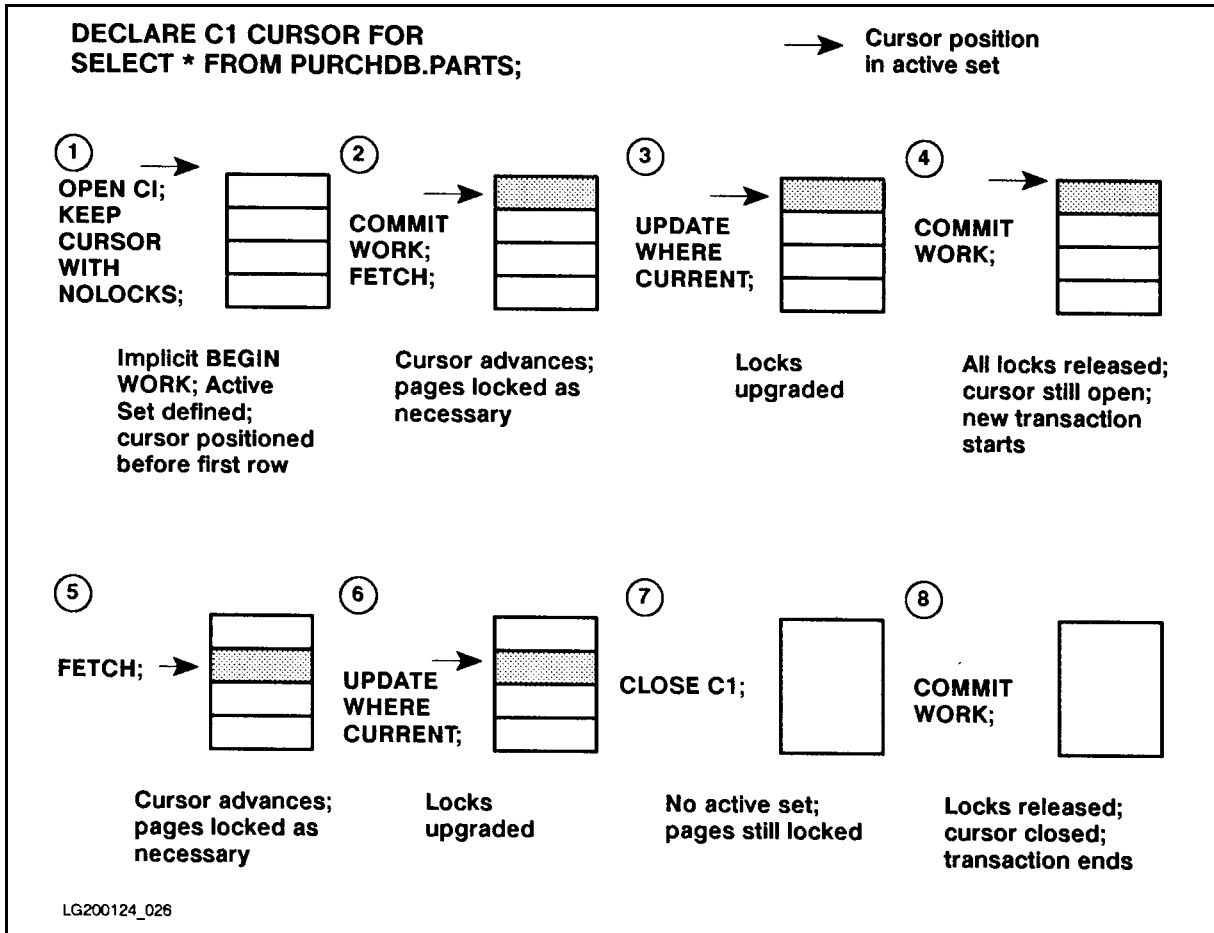


Figure 6-3. Cursor Operation Using `KEEP CURSOR WITH NOLOCKS`

## 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 was saved with a `COMMIT WORK` command immediately following 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 was saved with a `COMMIT WORK` command immediately following 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

Because of the potential for deadlock, you must be careful to test for that condition frequently in applications using `KEEP CURSOR`. 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 `SQLStatusCheck` 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.
  - First, open any non-kept cursors. Do *not* include a `COMMIT WORK` after opening the non-kept cursors. If a deadlock is detected at this point, the transaction will automatically be reapplied.
  - Next, execute any `FETCH`, `UPDATE WHERE CURRENT`, or `DELETE WHERE CURRENT` commands. Be sure to test for unexpected errors and branch to `SQLStatusCheck` to display messages and `RELEASE` in the event of a non-deadlock error. In the event of deadlock, the transaction will automatically be reapplied.
  - 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 `SQLStatusCheck` if necessary.
6. Execute a final `COMMIT WORK` to release the `KEEP` state.

Figure 6-4 is a skeleton outline of a `KEEP CURSOR` application showing the sections outlined above. Specific examples follow in the next section.

```

\
/* First, Declare all your Kept and Non-Kept Cursors */ (1)

do {
    EXEC SQL BEGIN WORK [RR/CS/RC];
    /* Open Kept Cursors in a Loop */
}while (!DeadLockFree); (2)

EXEC SQL COMMIT WORK; /* COMMIT to Save Cursor Positions */ (3)

do {
    /* Open any Non-Kept Cursors */

    /* Execute SQL Commands, i.e., FETCH, UPDATE, etc */

    if (DeadLockFree)
    {
        EXEC SQL COMMIT WORK; /* Save Cursor Positions */
        /* Write any messages or output to file or terminal */
    };
}while (!EndofScan); (4)

do {

    /* Close all Cursors Opened with KEEP CURSOR Option */

}while (!DeadlockFree); (5)

EXEC SQL COMMIT WORK; /* Final COMMIT to end KEEP state */ (6)

```

**Figure 6-4. Keep Cursor Application Program**

---

## 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. Four generalized code segments are presented:

- A status checking routine to be used 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

```
SQLStatusCheck ()
{
    /***/
    /* Deadlock occurred: Set DeadLockFree to FALSE */
    /***/

    if (sqlca.sqlcode = -14024) {
        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. */
        /***/

    }

    /***/
    /* Fatal Error: Show Messages, Release DBE, and terminate */
    /***/

    else if (sqlca.sqlcode < 0) {
        do {
            EXEC SQL SQLEXPLAIN :SQLMessage;
            printf(SQLMessage);
        } while (sqlca.sqlcode != 0);

        EXEC SQL RELEASE;
    }

    /***/
    /* On fatal errors, halt program to avoid an endless loop */
    /***/

    halt ();

    /***/
    /* No More Rows Found: Set EndofScan to TRUE */
    /***/

    if (sqlca.sqlcode = 100)
        EndofScan = TRUE;
} /* SQLStatusCheck function */
```

## Single Cursor WITH LOCKS

```
SQLSingleCursor ()
{
  /******
  /*      First, declare the cursor:      */
  /******

  EXEC SQL DECLARE C1 CURSOR FOR SELECT PartName,
      FROM PurchDB.Parts WHERE SalesPrice > 500.00;

  /******
  /* Next, open the cursor using KEEP CURSOR WITH LOCKS option: */
  /******

  do {
    DeadLockFree = TRUE;
    EXEC SQL OPEN C1 KEEP CURSOR WITH LOCKS;
    if (sqlca.sqlcode != 0)
      SQLStatusCheck ();
  } while (!DeadLockFree);

  /******
  /*      COMMIT WORK in order to preserve the cursor position */
  /******

  EXEC SQL COMMIT WORK;
  if (sqlca.sqlcode != 0)
    SQLStatusCheck ();

  /******
  /*      Execute BULK FETCH option until there is no deadlock      */
  /******

  EndofScan = FALSE;
  while (! EndofScan) {
    do {
      DeadlockFree = TRUE;
      EXEC SQL BULK FETCH C1 INTO :PriceList, 1, 20;
      if (sqlca.sqlcode != 0)
        SQLStatusCheck ();
      else {
        EXEC SQL COMMIT WORK;
        if (sqlca.sqlcode != 0)
          SQLStatusCheck ();
      }
    } while (!DeadLockFree);
  }
}
```

```

/*****/
/* Display qualified rows.  SQLERRD[3] contains the      */
/* number of qualified rows.                            */
/*****/

    printf ("\n Part Name           Sales Price\n");
    for (i = 0; i < sqlca.sqlerrd[2]; i++)
        printf("%s           %10.2f",PriceList[i].PartName,PriceList[i].SalesPrice);

} /* WHILE not EndofScan */
/*****/
/*      Execute the CLOSE command until no deadlock occurs      */
/*****/

do {
    EXEC SQL CLOSE C1;
    if (sqlca.sqlcode != 0)
        SQLStatusCheck();
} while (!DeadLockFree);

EXEC SQL COMMIT WORK;
if (sqlca.sqlcode != 0)
    SQLStatusCheck();

} /* SQLSingleCursor function */

```

## Multiple Cursors and Cursor Stability

```
SQLMultiCursor ()
{
    /*****
    /*      First, declare your cursors:
    *****/

    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;

    /*****
    /* Next, Open cursor C1. Use explicit BEGIN WORK in do while loop. */
    /* loop to make sure ALLBASE/SQL will use the CS isolation level */
    /* in case the program runs into a deadlock condition.
    *****/

    do {
        DeadLockFree = TRUE;
        EXEC SQL BEGIN WORK CS;
        if (sqlca.sqlcode != 0)
            SQLStatusCheck();
        else {
            EXEC SQL OPEN C1 KEEP CURSOR WITH LOCKS;
            if (sqlca.sqlcode != 0)
                SQLStatusCheck();
        }
    } while (!DeadLockFree);

    /*****
    /* COMMIT WORK in order to preserve the cursor position
    *****/

    EXEC SQL COMMIT WORK;
    if (sqlca.sqlcode != 0)
        SQLStatusCheck();

    /*****
    /* Initialize EndofScan to FALSE for the outer and inner loops */
    *****/

    EndofScan = FALSE;
    while (! EndofScan) {
```



```

/*****
/* The following do while loop is executed once per */
/* deadlock. We FETCH again using Cursor C1, reopen */
/* Cursor C2, then start to fetch rows using C2. */
/* Note that there is a deadlock, and when the */
/* transaction is aborted, Cursor */
/* C2 is closed and Cursor C1 returns to the beginning */
/* of the transaction. Any work done by the current */
/* transaction on the database is undone. This provides */
/* a method of reapplying the transaction when a */
/* deadlock at any point rolls it back. */
*****/

do {
    DeadLockFree = TRUE;
    EXEC SQL FETCH C1 INTO :HostBranchNo1;
    if (sqlca.sqlcode != 0)
        SQLStatusCheck();
    else /* 1 */ {
        EXEC SQL OPEN C2;
        if (sqlca.sqlcode != 0)
            SQLStatusCheck();
        else /* 2 */ {
            while ((! EndofScan) && DeadLockFree) {
                EXEC SQL FETCH C2 INTO :HostBranchNo2;
                if (sqlca.sqlcode != 0)
                    SQLStatusCheck();
                else /* 3 */ {
                    if (HostBranchNo1 == HostBranchNo2) {

/*****
/* Update the Branches table. Note: You might wish */
/* to include a DateUpdated column in the Branches */
/* table that can be updated with a timestamp every */
/* time the Credit column is updated. Then, in case */
/* the program terminates abnormally, you will know */
/* which values in the Credit column were updated */
/* before termination. */
*****/

                        EXEC SQL UPDATE Branches
                            SET Credit = Credit*0.005
                            WHERE CURRENT OF C2;
                        if (sqlca.sqlcode != 0)
                            SQLStatusCheck();
                    } /* if HostBranchNo1 */
                } /* else 3 */
            } /* inner while not EndofScan clause */

```

```

        if (EndofScan) {
            EndofScan = FALSE;
            EXEC SQL CLOSE C2;
            if (sqlca.sqlcode != 0)
                SQLStatusCheck();
            else /* 4 */ {
                EXEC SQL UPDATE TELLERS SET Status = :NewStatus
                    WHERE CURRENT OF C1;
                if (sqlca.sqlcode != 0)
                    SQLStatusCheck();
                else /* 5 */ {

/*****
/* Changes are committed and a new transaction begins. */
/* Cursor C1 still open; locks associated with the page */
/* pointed to by the cursor are kept. All locks for C2 */
/* are released when the transaction is committed */
*****/

                    EXEC SQL COMMIT WORK;
                    if (sqlca.sqlcode != 0)
                        SQLStatusCheck();
                } /* else 5 */
            } /* else 4 */
        } /* if EndofScan */
    } /* else 2 */
} /* else 1 */
} while (!DeadLockFree);
} /* outer while not EndofScan clause */

/*****
/* Repeat the CLOSE command until no deadlock is found: */
*****/

do {
    EXEC SQL CLOSE C1;
    if (sqlca.sqlcode != 0)
        SQLStatusCheck();
} while (!DeadLockFree);

/*****
/* Final COMMIT WORK: current transaction ends, and no new */
/* transaction begins. THIS STEP IS ABSOLUTELY ESSENTIAL */
/* TO END THE KEEP STATE! */
*****/

EXEC SQL COMMIT WORK;
if (sqlca.sqlcode != 0)
    SQLStatusCheck();

} /* SQLMultiCursor function */

```

## Avoiding Locks on Terminal Reads

```
SQLNoTermLock ()
{
    /******
    /*      First, declare the cursor:          */
    /******
    EXEC SQL DECLARE C1 Cursor FOR
        SELECT PartNumber, SalesPrice FROM PurchDB.Parts;

    /******
    /* Execute the OPEN command until there is no deadlock:      */
    /******
    do {
        DeadLockFree = TRUE;
        EXEC SQL OPEN C1 KEEP CURSOR WITH NOLOCKS;
        if (sqlca.sqlcode != 0)
            SQLStatusCheck;
    } while (!DeadLockFree);

    /******
    /* COMMIT WORK to preserve the cursor position:          */
    /******
    EXEC SQL COMMIT WORK;
    if (sqlca.sqlcode != 0)
        SQLStatusCheck;

    /******
    /* Now FETCH a row from the active set until EndofScan      */
    /******
    while (! EndofScan) {
        do {
            DeadLockFree = TRUE;
            EXEC SQL FETCH C1 INTO :PartNumber, :PresentSalesPrice;
            if (sqlca.sqlcode != 0)
                SQLStatusCheck();
            else {
                EXEC SQL COMMIT WORK;
                if (sqlca.sqlcode != 0)
                    SQLStatusCheck();
            }
        } while (!DeadLockFree);

        /******
        /* Display the present price and prompt for a new one:  */
        /******

        printf("\n Part Number:          %d", PartNumber);
        printf("\n Current Sales Price: %10.2f", PresentSalesPrice);
        scanf ("\n Enter New Sales Price: %10.2f", NewSalesPrice);
    }
}
```

```

/*****
/* Verify that the value of PresentSalesPrice has not */
/* changed. If not, update with NewSalesPrice */
*****/

do {
    DeadLockFree = TRUE;
    EXEC SQL SELECT SalesPrice INTO :SalesPrice FROM PurchDB.Parts
    WHERE PartNumber = :PartNumber;
    if (sqlca.sqlcode < 0)
        SQLStatusCheck();
    if (sqlca.sqlcode = 100)
        printf("\n Part Number no longer in database. Not updated.")
    else {
        if (SalesPrice == PresentSalesPrice) {
            EXEC SQL UPDATE PurchDB.Parts
            SET SalesPrice = :NewSalesPrice
            WHERE PartNumber = :PartNumber;
            if (sqlca.sqlcode != 0)
                SQLStatusCheck();
        }
        else printf("\n Current price has changed. Not updated.");
    }
} while (!DeadLockFree);

} /* while not EndofScan */

/*****
/* Execute the CLOSE command until there is no deadlock: */
*****/

do {
    EXEC SQL CLOSE C1;
    if (sqlca.sqlcode != 0)
        SQLStatusCheck();
} while (!DeadLockFree);

/*****
/* Final COMMIT WORK: current transaction ends, and no new */
/* transaction begins. THIS STEP IS ABSOLUTELY ESSENTIAL */
/* TO END THE KEEP STATE! */
*****/

EXEC SQL COMMIT WORK;
if (sqlca.sqlcode != 0)
    SQLStatusCheck();

} /* SQLNoTermLock function */

```

---

## Sample Program Using Cursor Operations

The flow chart in Figure 6-5 summarizes the functionality of program cex8. This program uses a cursor and the UPDATE WHERE CURRENT command to update column ReceivedQty in table PurchDB.OrderItems. The runtime dialog for cex8 appears in Figure 6-6, and the source code in Figure 6-7.

The program first executes function 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, function DeclareCursor only displays the message Declare the 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 cex8 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 function FetchUpdate until the Done flag is set to TRUE (28).

Function FetchUpdate prompts for an order number or a zero (17). When the user enters a zero, function FetchUpdate ends and the main program prompts the user to indicate whether another OrderNumber should be FETCHed. When the user enters an order number, the program begins a transaction by executing function 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 function DisplayUpdate (22) to display the current row and optionally update it.

Function DisplayUpdate executes function DisplayRow (10) to display the current row (6). If column ReceivedQty in the current row contains a null value, the message, ReceivedQty is NULL, is displayed.

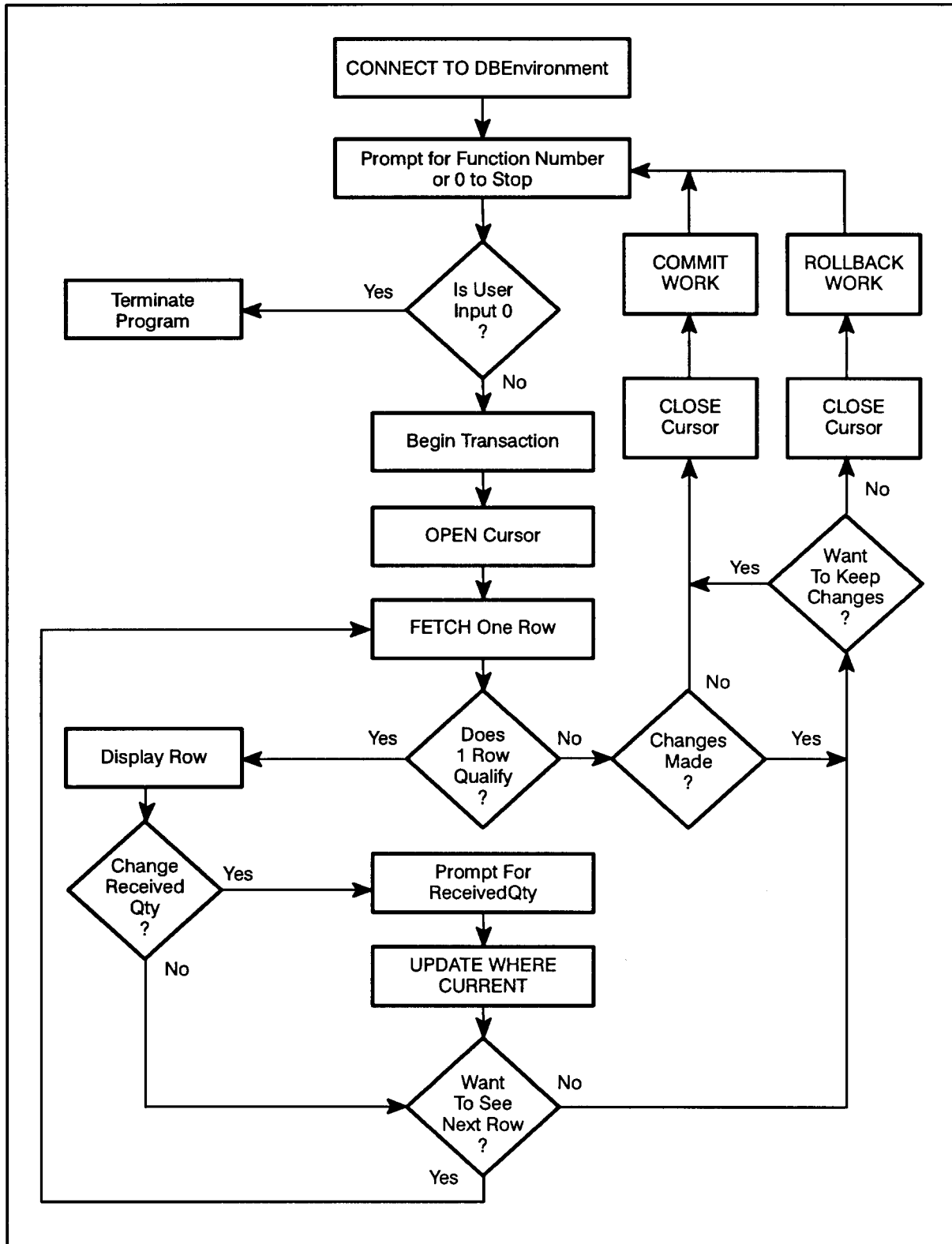
Function `DisplayUpdate` then prompts whether the user wants to update the current `ReceivedQty` value (11). If so, the user is prompted for a new value. The value accepted is used in an `UPDATE WHERE CURRENT` command (12). If the user entered a zero, a null value is assigned to this column.

The program then prompts whether to `FETCH` another row (13). If so, the `FETCH` command is re-executed. If not, the program prompts the user whether to make permanent any updates that may have made to the active set (14). To keep any row changes, the program executes function `EndTransaction` (16) which executes the `COMMIT WORK` command (4). To undo any row changes, the program executes function `RollBack` (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 an N in response to the main program's prompt to `FETCH` another order number, the program terminates by executing function `ReleaseDBE` (29), which executes the `RELEASE` command (2).



LG200124\_009

Figure 6-5. Flow Chart of Program cex8

```

Program to UPDATE OrderItems Table via a CURSOR - cex8
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 the DBEnvironment

Connect to PartsDBE

Declare Cursor

Enter OrderNumber or 0 to STOP > 30520

Begin Work

Open the Cursor

Fetch the next row.
Order Number:          30520
Item Number:           1
Vendor Part Number:   9375
Received Quantity:    9

Do you want to change ReceivedQty (Y/N)? > n

Do you want to see another row (Y/N)? > y

Fetch the next row.
Order Number:          30520
Item Number:           2
Vendor Part Number:   9105
Received Quantity:    3

Do you want to change ReceivedQty (Y/N)? > y

Enter New ReceivedQty or 0 for NULL > 15

```

**Figure 6-6. Runtime Dialog of Program cex8**



```
Update the PurchDB.OrderItems table

Do you want to see another row (Y/N)? > y

Fetch the next row.
  Order Number:          30520
  Item Number:           3
  Vendor Part Number:   9135
  Received Quantity:    3

Do you want to change ReceivedQty (Y/N)? > n

Do you want to see another row (Y/N)? > y

Fetch the next row.

Row not found or no more rows!

Do you want to save your changes (Y/N)? > y

Close the Cursor

Commit Work
1 rows changed!
Do you want to FETCH another OrderNumber (Y/N)? > y

Enter an OrderNumber or a 0 to STOP > 30510

Begin Work

Open the Cursor

Fetch the next row.
  Order Number:          30510
  Item Number:           1
  Vendor Part Number:   1001
  Received Quantity:    3

Do you want to change ReceivedQty (Y/N)? > n

Do you want to see another row (Y/N)? > n

Close Cursor

Commit Work
Do you want to FETCH another OrderNumber (Y/N)? > n
Release PartsDBE
```

Figure 6-6. Runtime Dialog of Program cex8 (page 2 of 2)

```

/* Program cex8 */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* This program illustrates the use of SQL's UPDATE WHERE          */
/* CURRENT command using a cursor to update a single row          */
/* at a time.                                                    */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

typedef int boolean;

boolean    Abort;
boolean    Done;
boolean    DoFetch;
char       response[2];
int        RowCounter;

#include <stdio.h>

#define     OK                0
#define     NotFound          100
#define     MultipleRows     -10002
#define     DeadLock         -14024
#define     FALSE            0
#define     TRUE              1

sqlca_type sqlca;    /* SQL Communication Area */

        /* Begin Host Variable Declarations */
EXEC SQL BEGIN DECLARE SECTION;
int      OrderNumber;
int      ItemNumber;
char     VendPartNumber[17];
int      ReceivedQty;
sqlind   ReceivedQtyInd;
char     SQLMessage[133];
EXEC SQL END DECLARE SECTION;
        /* End Host Variable Declarations */

```

Figure 6-7. Program cex8: Using UPDATE WHERE CURRENT

```

int SQLStatusCheck() /* Function to Display Error Messages */
{
Abort = FALSE;
if (sqlca.sqlcode < DeadLock) Abort = TRUE;

do {
EXEC SQL SQLEXPLAIN :SQLMessage;
printf("\n");
printf("%s\n",SQLMessage);
} while (sqlca.sqlcode != 0);

if (Abort) {

    ReleaseDBE();
    }

} /* End SQLStatusCheck Function */

boolean ConnectDBE() /* Function to Connect to PartsDBE */
{
boolean ConnectDBE;
printf("\n Connect to PartsDBE");
EXEC SQL CONNECT TO 'PartsDBE';

ConnectDBE = TRUE;
if (sqlca.sqlcode != OK) {
    ConnectDBE = FALSE;
    SQLStatusCheck();
    } /* End if */
return (ConnectDBE);
} /* End of ConnectDBE Function */

int ReleaseDBE() /* Function to Release PartsDBE */
{
printf("\n Release PartsDBE");
printf("\n");
EXEC SQL RELEASE;

Done = TRUE;
if (sqlca.sqlcode != OK) SQLStatusCheck();

} /* End ReleaseDBE Function */

```

27

1

29

2

**Figure 6-7. Program cex8: Using UPDATE WHERE CURRENT (page 2 of 8)**

```

boolean BeginTransaction()    /* Function to Begin Work */    (18)
{
boolean BeginTransaction;
printf("\n");
printf("\n Begin Work");

EXEC SQL BEGIN WORK;    (3)

if (sqlca.sqlcode != OK) {
    SQLStatusCheck();
    ReleaseDBE();
}

} /* End BeginTransaction Function */

int EndTransaction()    /* Function to Commit Work */    (16)
{
printf("\n");
printf("\n Commit Work");
EXEC SQL COMMIT WORK;    (4)
if (sqlca.sqlcode != OK) {
    SQLStatusCheck();
    ReleaseDBE();
}

} /* End EndTransaction Function */

int RollBack()    /* Function to RollBack work */    (15)
{
printf("\n");
printf("\n RollBack Work");
EXEC SQL ROLLBACK WORK;    (5)
if (sqlca.sqlcode != OK) {
    SQLStatusCheck();
    ReleaseDBE();
}

} /* End of RollBack Function */

```

Figure 6-7. Program cex8: Using UPDATE WHERE CURRENT (page 3 of 8)

```

int DisplayRow() /* Function to Display Parts Table Rows */ (10)
{
    printf("\n");
    printf("Order Number:          %10d\n", OrderNumber); (6)
    printf("Item Number:           %10d\n", ItemNumber);
    printf("Vendor Part Number:       %s\n", VendPartNumber);

    if (ReceivedQtyInd != 0)
        printf("Received Quantity:      is NULL \n");
    else
        printf("Received Quantity:          %5d\n", ReceivedQty);

} /* End of DisplayRow */

int DeclareCursor() /* Function to Declare the Cursor */ (26)
{
    printf("\n");
    printf("\n Declare the 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 DeclareCursor Function */

boolean OpenCursor() /* Function to Open the Declared Cursor */ (19)
{
    boolean OpenCursor;
    OpenCursor = TRUE;

    printf("\n");
    printf("\n Open the Cursor");
    EXEC SQL OPEN OrderReview; (8)
    if (sqlca.sqlcode != OK) {
        OpenCursor = FALSE;
        SQLStatusCheck();
        RollBack();
    }
    return(OpenCursor);
} /* End of OpenCursor Function */

```

Figure 6-7. Program cex8: Using UPDATE WHERE CURRENT (page 4 of 8)

```

int CloseCursor() /* Function to Close the Declared Cursor */
{
printf("\n");
printf("\n Close the Cursor");
EXEC SQL CLOSE OrderReview;
if (sqlca.sqlcode != OK) {
    SQLStatusCheck();
    EndTransaction();
}

} /* End of CloseCursor Function */

int DisplayUpdate() /* Display & Update row in OrderItems Table*/
{
DisplayRow();
printf("\n");

printf("\n Do you want to change ReceivedQty (Y/N)? > ");
scanf("%s",response);
if ((response[0] == 'Y') || (response[0] == 'y')) {
    printf("\n");
    printf("\n Enter new ReceivedQty or a 0 for NULL > ");
    scanf("%d5",&ReceivedQty);
    if (ReceivedQty == 0)
        ReceivedQtyInd = -1;
    else
        ReceivedQtyInd = 0;

    printf("\n UPDATE the PurchDB.OrderItems table");
    EXEC SQL UPDATE PurchDB.OrderItems
        SET ReceivedQty = :ReceivedQty :ReceivedQtyInd
        WHERE CURRENT OF OrderReview;

    if (sqlca.sqlcode != OK)
        SQLStatusCheck();
    else
        RowCounter = RowCounter + 1;
}

printf("\n");
printf("\n Do you want to see another row (Y/N)? > ");
scanf("%s",response);
if ((response[0] == 'N') || (response[0] == 'n')) {
    if (RowCounter > 0) {
        printf("\n");
        printf("\n Do you want to save the changes you made (Y/N)?>");
        scanf("%s",response);
    }
}
}

```

Figure 6-7. Program cex8: Using UPDATE WHERE CURRENT (page 5 of 8)

```

    if ((response[0] == 'N') || (response[0] == 'n')) {
        CloseCursor();
        RollBack();
        DoFetch = FALSE;
    }
    else {
        CloseCursor();
        EndTransaction();
        printf(RowCounter," %d\n rows were changed!");
        DoFetch = FALSE;
    }
}
if (RowCounter == 0) {
    CloseCursor();
    EndTransaction();
    DoFetch = FALSE;
}
}

} /* End DisplayUpdate Function */

int FetchUpdate() /* Fetch a row to Update within OrderItems */
{
    printf("\n");
    printf("\n Enter an OrderNumber or a 0 to STOP > ");
    scanf("%d",&OrderNumber);
    RowCounter = 0;
    if (OrderNumber != 0) {
        BeginTransaction();
        if (OpenCursor()) {
            DoFetch = TRUE;
            do {
                printf("\n");
                printf("\n FETCH the next row.");

                EXEC SQL FETCH OrderReview
                    INTO :OrderNumber,
                        :ItemNumber,
                        :VendPartNumber,
                        :ReceivedQty :ReceivedQtyInd;
            }
        }
    }
}

```

Figure 6-7. Program cex8: Using UPDATE WHERE CURRENT (page 6 of 8)

```

switch(sqlca.sqlcode) {

case OK:          DisplayUpdate();           (22)
                  break;

case NotFound:   DoFetch = FALSE;           (20)
                  printf("\n");
                  printf("\n Row not found or no more rows!");
                  if (RowCounter > 0) {
                    printf("\n");
                    printf("\n Do you want to save your changes (Y/N)?>");
                    scanf("%s",response);
                    if ((response[0] == 'N') || (response[0] == 'n')) {
                      CloseCursor();         (9)
                      RollBack();
                    }
                    else {
                      CloseCursor();         (9)
                      EndTransaction();      (23)
                      printf(RowCounter,"%d\n rows were changed!");
                    }
                  }
                  if (RowCounter == 0) {
                    CloseCursor();           (9)
                    EndTransaction();       (23)
                  }
                  break;

default:         DoFetch = FALSE;           (20)
                  SQLStatusCheck();
                  CloseCursor();           (9)
                  RollBack();             (24)
                  break;

} /* End switch */

} while (DoFetch != FALSE); /* End do */

} /* End if open */

} /* End if OrderNumber */

} /* End of FetchUpdate Function */

```

Figure 6-7. Program cex8: Using UPDATE WHERE CURRENT (page 7 of 8)



```

main()          /* Beginning of program */
{
printf("\n Program to UPDATE the OrderItems table via a CURSOR - cex8");
printf("\n");
printf("\n Event list:");
printf("\n  CONNECT TO PartsDBE");
printf("\n  Prompt for an Order Number");
printf("\n  BEGIN WORK");
printf("\n  OPEN CURSOR");
printf("\n  FETCH a row");
printf("\n  Display the retrieved row");
printf("\n  Prompt for new Received Quantity");
printf("\n  UPDATE row within the OrderItems table");
printf("\n  FETCH the next row, if any, with the same Order Number");
printf("\n  Repeat the above five steps until there are no more rows");
printf("\n  CLOSE CURSOR");
printf("\n  End Transaction");
printf("\n  Repeat the above eleven steps until the user enters a 0");
printf("\n  RELEASE the DBEnvironment");
printf("\n");

if (ConnectDBE()) {
DeclareCursor();

  Done = FALSE;
  do {
    FetchUpdate();
    printf("\n Do you want to FETCH another OrderNumber (Y/N)?>");
    scanf("%s",response);
    if ((response[0] == 'N') || (response[0] == 'n')) Done = TRUE;
  } while (Done != TRUE);

  ReleaseDBE();
}
else
  printf("\n Error: Cannot Connect to PartsDBE!\n");

} /* End of Program */

```

27

26

28

29

Figure 6-7. Program cex8: Using UPDATE WHERE CURRENT (page 8 of 8)



## 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.
- Sample 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:

```
struct StructName {  
    data type      Column1Name;  
    data type      Column2Name;  
    sqlind         Col2IndVar;  
    .  
    .  
    .  
    data type      ColumnnName;  
    sqlind         ColnIndVar;  
} ArrayName[n];
```

You reference the name of the array in the BULK SQL command:

```
EXEC SQL BEGIN DECLARE SECTION;
struct {
    char          PartNumber[17];
    char          PartName[31];
    sqlind        PartNameInd;
} PartsArray[26];
double          SalesPrice;
EXEC SQL END DECLARE SECTION;
.
.
.
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 in the array. If not specified, the assumed subscript is zero.
- 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, the default number of rows is the number of records in the array from the `StartIndex` to the end of the array for an `INSERT` operation. For a retrieval operation, the default number of rows is the smaller of two values; 1) the number of records in the array from the `StartIndex` to the end of the array, or 2) 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;
struct {
    char          PartNumber[17];
    char          PartName[31];
    sqlind        PartNameInd;
} PartsArray[26];
short int        StartIndex;
short int        NumberOfRows;
double           SalesPrice;
EXEC SQL END DECLARE SECTION;
.
.
.
EXEC SQL BULK SELECT  PartNumber, PartName
                    INTO :PartsArray,
                        :StartIndex,
                        :NumberOfRows
                    FROM  PurchDB.Parts
                    WHERE SalesPrice < :SalesPrice;
```

---

**Note** 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;
struct {
    short int          CountCycle;
    int               PartCount;
} PartsPerCycle[3];
EXEC SQL END DECLARE SECTION;
.
.
.
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;
struct {
    short int          CountCycle;
    int               PartCount;
} PartsPerCycle[15];
short int            StartIndex;
short int            NumberOfRows;
char                 LowBinNumber[16];
char                 HighBinNumber[16];
EXEC SQL END DECLARE SECTION;
:
:
int DisplayRows()
{
int    i;

for (i = 0; i < StartIndex; i++) {
    printf("CountCycle:  %d\n", PartsPerCycle[i].CountCycle);
    printf("PartCount:   %d\n", PartsPerCycle[i].PartCount);
} /* end for */
} /* end of procedure DisplayRows */
.
.
.
main()
{

#define    TRUE    1
#define    FALSE   0

typedef    int    boolean;

int        StartIndex;
int        NumberOfRows;
boolean    LessThanFive;

/* Initialize variables. */

StartIndex = 0;
NumberOfRows = 3;
LessThanFive = TRUE;

do {

```

*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.*

```

printf("\n Enter a low bin number or 0 to STOP > ");
scanf("%d",LowBinNumber);

if (LowBinNumber != 0) {
    printf("\n Enter a high bin number > ");
    scanf("%d\n",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 == 15) LessThanFive = FALSE;

    } /* if LowBinNumber */
    else
        LessThanFive = FALSE;
} while (LessThanFive == TRUE);

```

*The final StartIndex value is used to display the final contents of the host variable array.*

```

if (StartIndex > 0) {
    DisplayRows();
}

}

```

The following example illustrates the use of `sqlca.sqlerrd[2]` to display rows stored in the host variable array. It also checks `sqlca.sqlcode` in conjunction with `sqlca.sqlerrd[2]`, 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.

```

int DisplayRows()
{
    int  i;

    for (i = 0; i < sqlca.sqlerrd[2]; i++) {
        printf("OrderNumber:  %s\n", OrdersArray[i].OrderNumber);
        printf("VendorNumber: %s\n", OrdersArray[i].VendorNumber);
    }
}      (* end of procedure DisplayRows *)
:
:

```

*The variable MaximumRows is set to the number of records in the host variable array.*



```

Main()
{
:
MaximumRows = 25;
:
EXEC SQL BULK SELECT  OrderNumber, VendorNumber
                    INTO :OrdersArray
                    FROM  PurchDB.Orders;

switch (sqlca.sqlcode) {
  case 0:      if (sqlca.sqlerrd[2] == MaximumRows) {
                printf("\n There may be additional rows ");
                printf("\n that cannot be displayed.");
                }
                DisplayRows();
                break;
  case 100:    printf("\n No rows were found!");
                break;
  default:    if (sqlca.sqlerrd[2] > 0) {
                printf("\n The following rows were retrieved ");
                printf("\n before an error occurred:");
                }
                DisplayRows();
                SQLStatusCheck();
                break;
} /* End switch */
:
}

```

## BULK FETCH

The BULK FETCH command is useful for reporting applications that operate on large query results or query results whose maximum 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 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:** opens the cursor.
- **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 sqlca.sqlcode to determine when you have fetched the last row in the active set:

```

:
int DisplayRows()
{
int    i;

for (i = 0; i < sqlca.sqlerrd[2]; i++) {

    The values in each row returned by the BULK FETCH command are displayed here.
    }

if (sqlca.sqlcode != 0) {
    printf("\n Do you want to see additional rows? (yes/no) > ");
    scanf("%s", Response);
    if ((Response[0] == 'N') || (Response[0] == 'n')) {
        DoFetch = FALSE;
    }
}    (* end of DisplayRows procedure *)
:
main()
{
EXEC SQL BEGIN DECLARE SECTION;
struct {
    char          PartNumber[17];
    char          VendorName[31];
    short int     DeliveryDays;
    sqlcint       DeliveryDaysInd;

```

```

    } SupplierBuffer[20];
EXEC SQL END DECLARE SECTION;
typedef int boolean;
boolean          DoFetch;
char             Response[2];
#define TRUE     1
#define FALSE    0
:
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;
do {
    EXEC SQL BULK FETCH SupplierInfo
            INTO :SupplierBuffer;

    switch (sqlca.sqlcode) {
        case 0:      DisplayRows();
                    break;
        case 100:   printf("\n No rows were found!");
                    DoFetch = FALSE;
                    break;
        default:    DisplayRows();
                    SQLStatusCheck();
                    DoFetch = FALSE;
                    break;
    } /* End switch */

} while (DoFetch != TRUE);

EXEC SQL CLOSE SupplierInfo;

```

After the BULK FETCH command is executed, the last row ALLBASE/SQL put into the host variable array is the current row. If the BULK FETCH command is re-executed, the first row in the next set of rows fetched is the row following the current row, and the last row fetched becomes the current row. When the last row in the active set has been fetched, ALLBASE/SQL sets sqlca.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 form 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;

struct {
    char          PartNumber[17];
    char          PartName[31];
    sqlind       PartNameInd;
    double        SalesPrice;
    sqlind       SalesPriceInd;
} NewParts[20];

short int       StartIndex;
short int       NumberOfRows;

EXEC SQL END DECLARE SECTION;

typedef int boolean;
boolean         DoneEntry;
char           Response[2];

#define         TRUE         1
#define         FALSE        0

StartIndex = 0;
NumberOfRows = 0;
DoneEntry = FALSE;

do {
    PartEntry();
} while (DoneEntry != TRUE);
.
.
.
int BulkInsert()
{
```

```

EXEC SQL BULK INSERT INTO  PurchDB.Parts
                        (PartNumber,
                         PartName,
                         SalesPrice)
VALUES (:NewParts,
       :StartIndex,
       :NumberOfRows);

```

```

.
.
} /* End of function BulkInsert */

```

```

int PartEntry()
{

```

```

.
.

```

*You are 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 you are asked whether you want to specify another line item.*

```

.
.

```

```

NumberOfRows = NumberOfRows + 1;
printf("\n Do you want to specify another line item (Y/N)? > ");
scanf("%s",Response);

```

```

if ((Response[0] == 'N') || (Response[0] == 'n')) {
    DoneEntry = TRUE;
    BulkInsert();
}

```

```

else {
    if (NumberOfRows == 20) {
        BulkInsert();
        NumberOfRows = 0;
    }
} /* End else */

```

```

.
.

```

```

} /* End of PartEntry Function */

```

---

## 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* .

---

## Sample Program Using BULK Processing

The flow chart in Figure 7-1 summarizes the functionality of program cex9. 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 consists of one or more line items, represented by one or more rows in the table PurchDB.OrderItems:

```
OrderNumber (defined NOT NULL)
ItemNumber (defined NOT NULL)
VendPartNumber
PurchasePrice (defined NOT NULL)
OrderQty
ItemDueDate
ReceivedQty
```

Program cex9 uses a simple INSERT command to create the order header and, optionally, a BULK INSERT command to insert line items.

The runtime dialog for program cex9 appears in Figure 7-2, and the source code in Figure 7-3.

To establish a DBE session (54), cex9 executes function ConnectDBE. This function evaluates to TRUE when the CONNECT command (5) is successfully executed.

The program then executes function CreateOrder until the Done flag is set to TRUE (55).

Function CreateOrder prompts for a vendor number or a zero (0) to stop the program (48). When the user enters a zero, Done is set to TRUE (53) and the program terminates. When the user enters a vendor number, program cex9:

- 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, function ValidateVendor is executed (49). Function ValidateVendor starts a transaction by invoking function 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 function EndTransaction (11). EndTransaction 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 function `CreateOrder`, the user is again prompted for a vendor number.

- If the `SELECT` command fails, function `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, program `cex9` invokes function `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`.

Function `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:

- Function `CreateHeader` invokes function `ComputeOrderNumber` (36) to compute the order number and the order date.
- Function `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.
- Function `ComputeOrderNumber` then executes function `SystemDate` (32). This function calls the system time library (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`.
- Function `ComputeOrderNumber` then executes function `InsertRow` (33). This function 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 a `COMMIT WORK` command, but the `HeaderOK` flag is set to `FALSE` (23) so that the user is prompted for another vendor number when control returns to function `CreateOrder`.

To create line items, function `CreateOrder` executes function `CreateOrderItems` until the `DoneItems` flag is set to `TRUE` (51). Function `CreateOrderItems` prompts for whether the user wants to specify line items (44).

If the user wants to create line items, function `CreateOrderItems` executes function `ItemEntry` until the `DoneItems` flag is set to `TRUE` (46), then executes function `BulkInsert` (47):

- Function `ItemEntry` assigns values to the host variable structure array `OrderItems` (1); each record in the array corresponds to one line item, or row in `PurchDB.OrderItems`. The function first assigns the order number and a line number to each row (37), beginning at one. `ItemEntry` then prompts for a vendor part number (38), which is validated by invoking function `ValidatePart` (39).



Function `ValidatePart` starts a transaction (14). Then it executes a `SELECT` command (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 (16) and the `PartOK` flag set to `TRUE`. If the part number is invalid, `COMMIT WORK` is executed (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 function `ItemEntry`.

If the part number is valid, function `ItemEntry` completes the line item. It prompts for values to assign to columns `PurchasePrice`, `OrderQty`, and `ItemDueDate` (40). The function then assigns a negative value to the indicator variable for column `ReceivedQty` (41) in preparation for inserting a null value into this column.

Function `ItemEntry` terminates when the user indicates that the user does not want to specify any more line items (42) or when the host variable array is full (43).

- Function `BulkInsert` starts a transaction (25), then executes the `BULK INSERT` command (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 (26). If the `BULK INSERT` command succeeds, the `COMMIT WORK` command is executed (29) and the `ItemsOK` flag set to `TRUE`. If the `BULK INSERT` command fails, function `RollBackWork` is executed (28) to process the `ROLLBACK WORK` command (8) so that any rows inserted prior to the failure are rolled back.

If the user does not want to create line items, function `CreateOrderItems` displays the order header by invoking function `DisplayHeader` (45). Function `DisplayHeader` displays the row (18) that was created earlier in function `CreateHeader` (50) and inserted into `PurchDB.Orders` in the function `InsertRow` (33).

If line items were successfully inserted into the `PurchDB.OrderItems` table in function `BulkInsert`, then function `CreateOrder` invokes function `DisplayOrder` (52) to display the order created. Function `DisplayOrder` invokes function `DisplayHeader` (20) to display the order header. Then it executes function `DisplayItems` (21) to display each row inserted into the `PurchDB.OrderItems` table. Function `DisplayItems` displays values from the structure array `OrderItems` (19).

When the program user enters a 0 in response to the vendor number prompt in function `CreateOrder`, `Done` is set to `TRUE` (53) and the program terminates by executing function `ReleaseDBE` (56), which executes the `RELEASE` command (3).

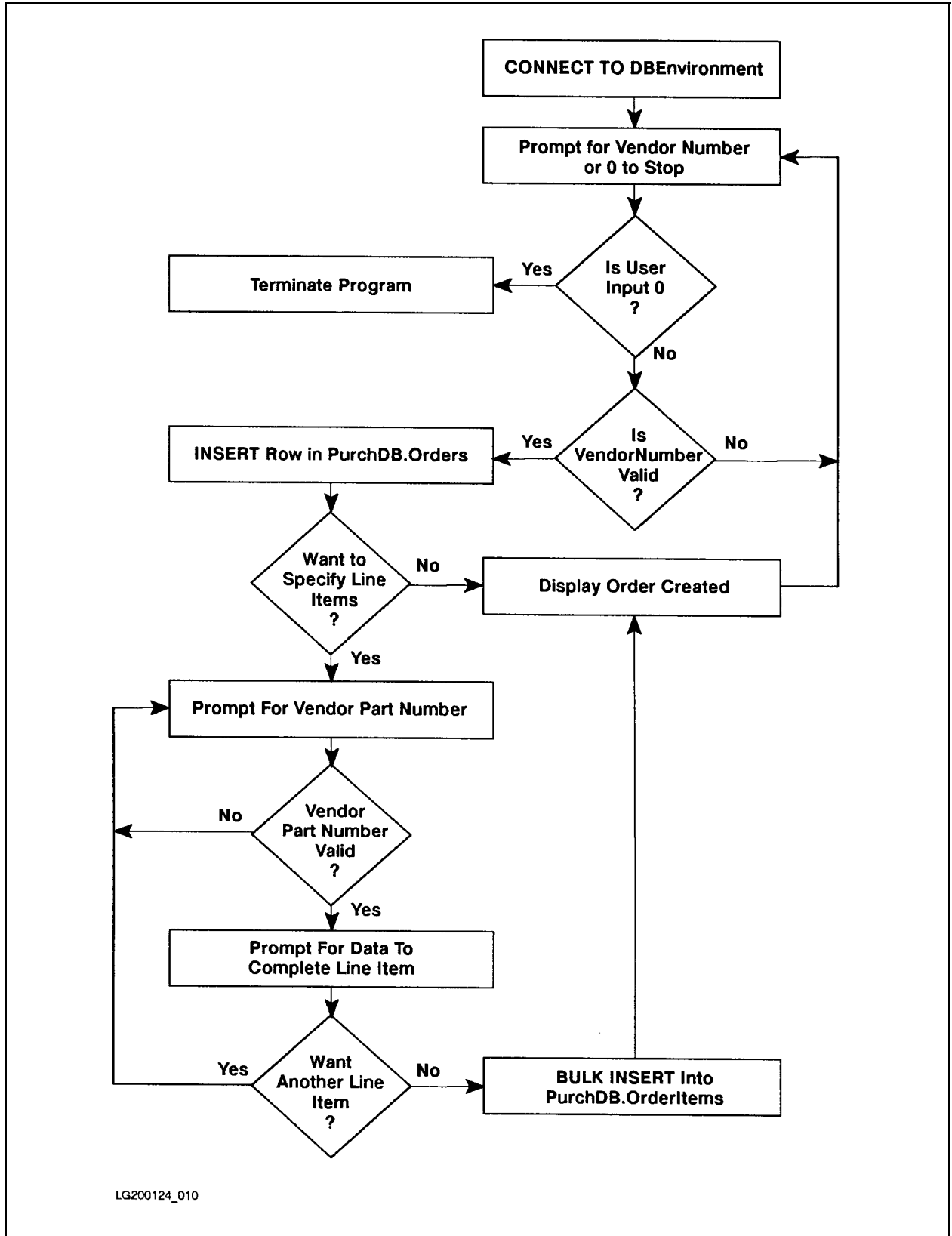


Figure 7-1. Flow Chart of Program cex9

```

Program to Create an Order - cex9

Event List:
CONNECT TO PartsDBE
Prompt for VendorNumber
Validate VendorNumber
BEGIN WORK
INSERT a row into PurchDB.Orders
Prompt for a line item
Validate the Vendor Part Number for each line item
BULK INSERT rows into PurchDB.OrderItems
Repeat the above six steps until the user enters a 0
RELEASE the DBEnvironment

Connect to PartsDBE

Enter a Vendor Number or a 0 to STOP> 9015

Begin Work
Validating VendorNumber

Commit Work

Begin Work
LOCK the PurchDB.Orders table.
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

Vendor Part Number > 9040

Begin Work
Validating VendPartNumber

Commit Work

Purchase Price > 1500

```

**Figure 7-2. Runtime Dialog of Program cex9**

```
Order Quantity > 5

Item Due Date (yyyymmdd) > 19860630

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

Vendor Part Number > 9055

Begin Work
Validating VendPartNumber

Commit Work

The vendor has no part with the number
you specified!

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

Vendor Part Number > 9050

Begin Work
Validating VendPartNumber

Commit Work

Purchase Price > 345

Order Quantity > 2

Item Due Date (yyyymmdd) > 19860801

Do you want to specify another line item (Y/N)?> n

Begin Work
BULK INSERT into PurchDB.OrderItems

Commit Work
```

Figure 7-2. Runtime Dialog of Program cex9 (page 2 of 3)

The following order has been created:

Order Number: 30524  
Vendor Number: 9015  
Order Date: 19860603

Item Number: 1  
Vendor Part Number: 9040  
Purchase Price: 1500.00  
Order Quantity: 5  
Item Due Date: 19860630  
Received Quantity: is NULL

Item Number: 2  
Vendor Part Number: 9050  
Purchase Price: 345.00  
Order Quantity: 2  
Item Due Date: 19860801  
Received Quantity: is NULL

Enter a Vendor Number or a 0 to STOP > 0

Release PartsDBE

Figure 7-2. Runtime Dialog of Program cex9 (page 3 of 3)

```

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* This program illustrates the use of SQL's BULK INSERT          */
/* command to insert mulitple rows or tuples at a time.          */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

typedef int boolean;

boolean    Abort;
boolean    Done;
boolean    DoneItems;
boolean    VendorOK;
boolean    HeaderOK;
boolean    PartOK;
boolean    ItemsOK;
char       response[2];
int        counter1;
int        counter2;
int        i,j,k;
long       sec;

char       *Date;

#include <stdio.h>
#include <time.h>

#define     OK                0
#define     NotFound          100
#define     MultipleRows     -10002
#define     DeadLock         -14024
#define     FALSE            0
#define     TRUE             1

sqlca_type sqlca;    /* SQL Communication Area */

        /* Begin Host Variable Declarations */
EXEC SQL BEGIN DECLARE SECTION;
int      OrderNumber1;
int      VendorNumber;
char     OrderDate[9];
char     PartSpecified[17];
int      MaxOrderNumber;
short    StartIndex;
short    NumberOfRows;

```

Figure 7-3. Program cex9: Using BULK INSERT

```

    struct    {
        int      OrderNumber2;
        int      ItemNumber;
        char     VendPartNumber[17];
        double   PurchasePrice;
        short    OrderQty;
        char     ItemDueDate[9];
        short    ReceivedQty;
        sqlind   ReceivedQtyInd;
    } OrderItems[25];           1
    char        SQLMessage[133];
    EXEC SQL END DECLARE SECTION;
        /* End Host Variable Declarations */

int SystemDate() /* Function to get the system date */           32
{

    sec = time(0);
    printf("\n Calculating OrderDate");
    ptr = localtime(&sec);
    sprintf(OrderDate, "%2s%2.2d%2.2d%2.2d",
    "19",ptr->tm_year,++ptr->tm_mon,ptr->tm_mday);

} /* End of SystemDate Function */

int SQLStatusCheck() /* Function to Display Error Messages */   13
{
    Abort = FALSE;
    if (sqlca.sqlcode < DeadLock) Abort = TRUE;

    do {
        EXEC SQL SQLEXPLAIN :SQLMessage;           4
        printf("\n");
        printf("%s\n",SQLMessage);
    } while (sqlca.sqlcode != 0);

    if (Abort) {
        ReleaseDBE();
    }

} /* End SQLStatusCheck Function */

```

Figure 7-3. Program cex9: Using BULK INSERT (page 2 of 11)

```

boolean ConnectDBE() /* Function to Connect to PartsDBE */
{
boolean ConnectDBE;
printf("\n Connect to PartsDBE");
EXEC SQL CONNECT TO 'PartsDBE';

ConnectDBE = TRUE;
if (sqlca.sqlcode != OK) {
    ConnectDBE = FALSE;
    SQLStatusCheck();
} /* End if */
return (ConnectDBE);
} /* End of ConnectDBE Function */

int ReleaseDBE() /* Function to Release PartsDBE */
{
printf("\n Release PartsDBE");
EXEC SQL RELEASE;

Done = TRUE;
if (sqlca.sqlcode != OK)
    SQLStatusCheck();

} /* End ReleaseDBE Function */

boolean BeginTransaction() /* Function to Begin Work */
{
boolean BeginTransaction;
printf("\n");
printf("\n Begin Work");

EXEC SQL BEGIN WORK;

if (sqlca.sqlcode != OK) {
    SQLStatusCheck();
    ReleaseDBE();
}

} /* End BeginTransaction Function */

```

Figure 7-3. Program cex9: Using BULK INSERT (page 3 of 11)



```

int EndTransaction() /* Function to Commit Work */
{
printf("\n");
printf("\n Commit Work");
EXEC SQL COMMIT WORK;
if (sqlca.sqlcode != OK) {
    SQLStatusCheck();
}

} /* End EndTransaction Function */

int RollBackWork() /* Function to RollBack work */
{
printf("\n");
printf("\n RollBack Work");
EXEC SQL ROLLBACK WORK;
if (sqlca.sqlcode != OK) {
    SQLStatusCheck();
    ReleaseDBE();
}

} /* End of RollBack Function */

int ValidateVendor() /* Function that ensures vendor number is valid */
{
BeginTransaction();

printf("\n Validating VendorNumber");
EXEC SQL SELECT VendorNumber
        INTO :VendorNumber
        FROM PurchDB.Vendors
        WHERE VendorNumber = :VendorNumber;

switch (sqlca.sqlcode) {

    case OK:
        VendorOK = TRUE;
        EndTransaction();
    case NotFound:
        EndTransaction();
        printf("\n");
        printf("\n No vendor has the VendorNumber you");
        printf("\n specified!");
}

VendorOK = FALSE;
HeaderOK = FALSE;
ItemsOK = FALSE;
break;
}

```

Figure 7-3. Program cex9: Using BULK INSERT (page 4 of 11)

```

default:          SQLStatusCheck();           (13)
                  EndTransaction();
                  VendorOK = FALSE;
                  HeaderOK = FALSE;
                  ItemsOK = FALSE;
                  break;

} /* End switch */

} /* End ValidateVendor Function */

\mf="Function"
int ValidatePart() /* Function that ensures vendor part number is valid */
{
BeginTransaction();           (39)
                               (14)

sscanf(OrderItems[i].VendPartNumber,"%s",PartSpecified);

printf("\n Validating VendPartNumber");
EXEC SQL SELECT  VendPartNumber           (15)
                INTO :PartSpecified
                FROM  PurchDB.SupplyPrice
                WHERE VendorNumber = :VendorNumber
                AND  VendPartNumber = :PartSpecified;

switch (sqlca.sqlcode) {

case OK:          EndTransaction();           (16)
                  PartOK = TRUE;
                  break;

case NotFound:   EndTransaction();           (17)
                  printf("\n");
                  printf("\n The vendor has no part with the number"
                  printf("\n      you specified!");
                  PartOK = FALSE;
                  break;

```

Figure 7-3. Program cex9: Using BULK INSERT (page 5 of 11)

```

default:          SQLStatusCheck();
                  EndTransaction();
                  PartOK = FALSE;
                  break;
} /* End switch */

} /* End ValidatePart Function */

int DisplayHeader() /* Function to Display row from PurchDB.Orders */
{
    printf("\n");
    printf("\n The following order has been created.");
    printf("\n");
    printf("Order Number:          %d\n", OrderNumber1);
    printf("Vendor Number:             %d\n", VendorNumber);
    printf("Order Date:                 %s\n", OrderDate);
} /* End of DisplayRow Function */

int DisplayItems() /* Function to Display Rows from PurchDB.OrderItems */
{
    j = counter2;
    printf("\n");
    printf("Item Number:                %d\n", OrderItems[j].ItemNumber);
    printf("Vendor Part Number:         %s\n", OrderItems[j].VendPartNumber);
    printf("Purchase Price:             %10.2f\n", OrderItems[j].PurchasePrice);
    printf("Order Quantity:            %d\n", OrderItems[j].OrderQty);
    printf("Item Due Date:              %s\n", OrderItems[j].ItemDueDate);
    printf("Received Quantity:         is NULL \n");
    counter2 = j + 1;
} /* End of DisplayRow */

int DisplayOrder() /* Function to Display Order Created */
{
    DisplayHeader();
    printf("\n");
    i = counter1;
    counter2 = 1;
    do {
        DisplayItems();
        j = j + 1;
    } while (j < i);
} /* End of DisplayOrder Function */

```

Figure 7-3. Program cex9: Using BULK INSERT (page 6 of 11)

```

int InsertRow() /* Function to insert row in PurchDB.Orders */ (33)
{
printf("\n");
printf("\n INSERT into PurchDB.Orders");
EXEC SQL INSERT INTO   PurchDB.Orders (22)
                    (OrderNumber,
                     VendorNumber,
                     OrderDate)
                    VALUES (:OrderNumber1,
                             :VendorNumber,
                             :OrderDate);

if (sqlca.sqlcode != 0) {
    SQLStatusCheck();
    EndTransaction();
    HeaderOK = FALSE; (23)
}
else {
    EndTransaction();
    HeaderOK = TRUE; (24)
}

} /* End of InsertRow Function */

int BulkInsert() /* Function to bulk insert into PurchDB.OrderItems */
{ (47)
BeginTransaction(); (25)
NumberOfRows = counter1; (26)
StartIndex = 1;

printf("\n 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 7-3. Program cex9: Using BULK INSERT (page 7 of 11)

```

if (sqlca.sqlcode != 0) {
    SQLStatusCheck();
    RollBackWork();
    ItemsOK = FALSE;
}
else {
    EndTransaction();
    ItemsOK = TRUE;
}

} /* End of BulkInsert Function */

int ComputeOrderNumber() /* Function to assign a number to an order */
{
EXEC SQL SELECT  MAX(OrderNumber)
                INTO :MaxOrderNumber
                FROM  PurchDB.Orders;

if (sqlca.sqlcode != 0) {
    SQLStatusCheck();
    EndTransaction();
    HeaderOK = FALSE;
}
else {
    printf("\n Calculating OrderNumber");
    OrderNumber1 = MaxOrderNumber + 1;
    SystemDate();
    InsertRow();
}

} /* End ComputeOrderNumber Function */

int CreateHeader() /* Function to create order header */
{
BeginTransaction();

printf("\n LOCK the PurchDB.Orders table.");
EXEC SQL LOCK TABLE PurchDB.Orders IN EXCLUSIVE MODE;

if (sqlca.sqlcode != 0) {
    SQLStatusCheck();
    EndTransaction();
    HeaderOK = FALSE;
}

```

Figure 7-3. Program cex9: Using BULK INSERT (page 8 of 11)

```

else
    ComputeOrderNumber();                                36

} /* End CreateHeader Function */

int ItemEntry() /* Function to put line items into OrderItems array */
{
    i = counter1;                                        46
    OrderItems[i].OrderNumber2 = OrderNumber1;          37
    OrderItems[i].ItemNumber = i;
    printf("\n");
    printf("\n You can specify as many as 25 line items.");
    printf("\n");
    printf("\n Enter data for ItemNumber:  %d\n", OrderItems[i].ItemNumber);
    printf("\n");
    printf("\n Vendor Part Number > ");                38
    scanf("%s%c",OrderItems[i].VendPartNumber);
    ValidatePart();                                    39
    DoneItems = FALSE;
    if (PartOK != FALSE) {
        printf("\n");
        printf("\n Purchase Price > ");                40
        scanf("%1f",&OrderItems[i].PurchasePrice);

        printf("\n Order Quantity > ");                40
        scanf("%5d%c",&OrderItems[i].OrderQty);

        printf("\n Item Due Date (yyyymmdd) > ");      40
        scanf("%s%c",OrderItems[i].ItemDueDate);

        OrderItems[i].ReceivedQtyInd = -1;              41

        counter1 = i + 1;
    }
    if (i < 25) {
        printf("\n");
        printf("\n Do you want to specify another line item (Y/N)? >");
        scanf("%s%c",response);                        42
        if ((response[0] == 'N') || (response[0] == 'n')) {
            DoneItems = TRUE;
        }
    }
    else
        DoneItems = TRUE;                                43
} /* End ItemEntry Function */

```

Figure 7-3. Program cex9: Using BULK INSERT (page 9 of 11)

```

int CreateOrderItems() /* Function to create line items */ 51
{
ItemsOK = FALSE;
printf("\n");
printf("\n Do you want to specify line items (Y/N)? > "); 44
scanf("%s%c",response);
if ((response[0] == 'N') || (response[0] == 'n')) {
    DoneItems = TRUE;
    DisplayHeader(); 45
}
else {
    counter1 = 1;
    do {
        ItemEntry(); 46
    } while (DoneItems == FALSE);
    if (counter1 != 1) {
        BulkInsert(); 47
        ItemsOK = TRUE;
    }
}
} /* End of CreateOrderItems Function */

int CreateOrder() /* Function to create an order */ 55
{
printf("\n");
printf("\n Enter a Vendor Number or a 0 to STOP > "); 48
scanf("%10d%c",&VendorNumber);
if (VendorNumber != 0) {
    ValidateVendor(); 49
    if (VendorOK == TRUE) CreateHeader(); 50
    if (HeaderOK == TRUE) {
        DoneItems = FALSE;
        do {
            CreateOrderItems(); 51
        } while (DoneItems == FALSE);
    }
    if (ItemsOK == TRUE) DisplayOrder(); 52
}
else
    Done = TRUE; 53
} /* End CreateOrder Function */

```

Figure 7-3. Program cex9: Using BULK INSERT (page 10 of 11)

```

main()          /* Beginning of program */
{
printf("\n Program to CREATE an Order - cex9");
printf("\n");
printf("\n Event list:");
printf("\n  CONNECT TO PartsDBE");
printf("\n  Prompt for a Vendor Number");
printf("\n  Validate the Vendor Number");
printf("\n  BEGIN WORK");
printf("\n  INSERT a row into PurchDB.Orders");
printf("\n  Prompt for a line item");
printf("\n  Validate the Vendor Part Number for each line item");
printf("\n  BULK INSERT rows into PurchDB.OrderItems");
printf("\n  Repeat the above six steps until the user enters a 0");
printf("\n  RELEASE the DBEnvironment");
printf("\n");

if (ConnectDBE()) {
    Done = FALSE;
    do {
        CreateOrder();
    } while (Done != TRUE);

    ReleaseDBE();
}
else
    printf("\n Error: Cannot Connect to PartsDBE!\n");

} /* End of Program */

```

54

55

56

Figure 7-3. Program cex9: Using BULK INSERT (page 11 of 11)



## 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 C 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 and other data structures. These statements and data structures, used with a cursor, are described further in a later section.

---

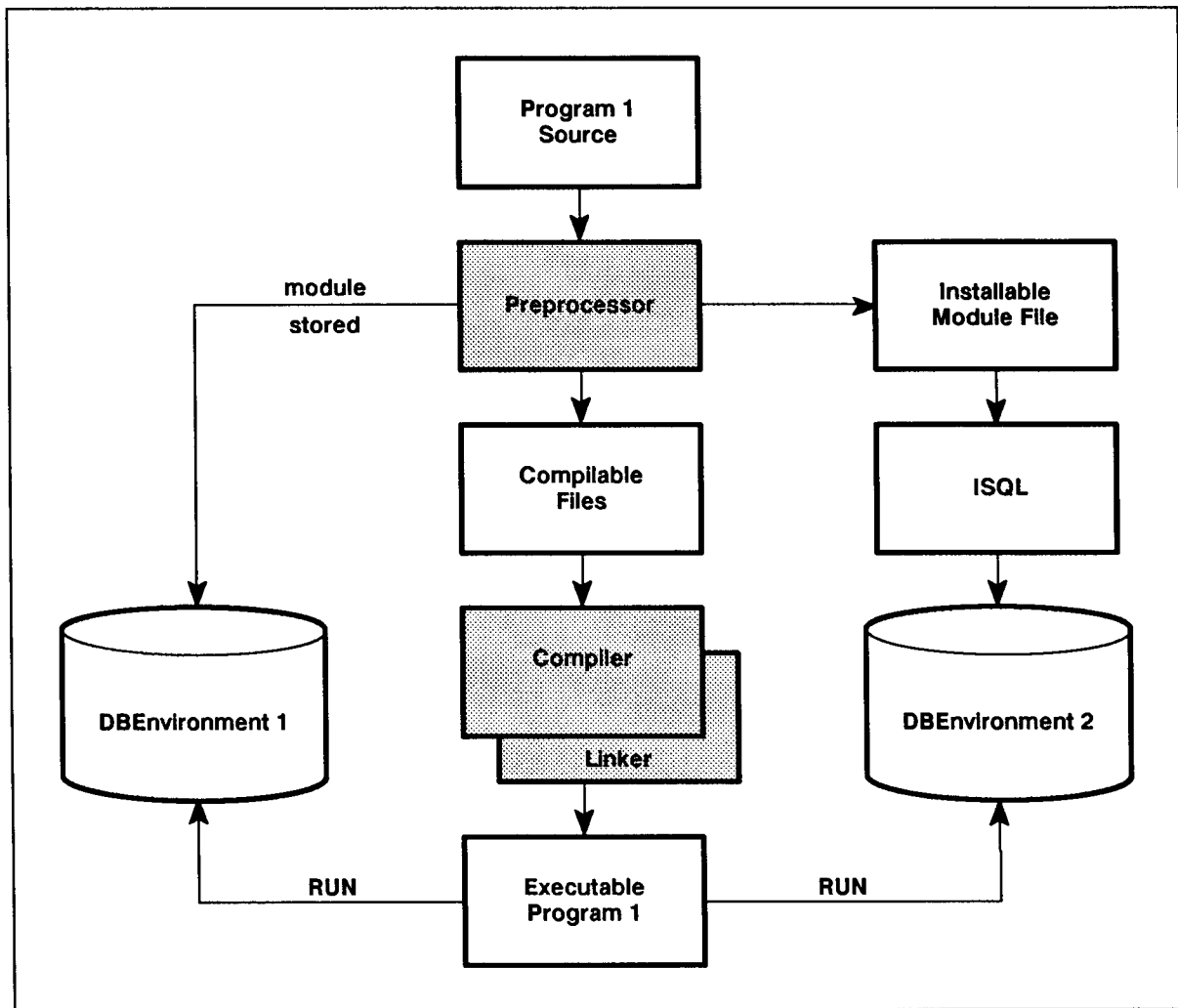
## Differences between Dynamic and Non-Dynamic Preprocessing

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 C 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 8-1 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 8-1. 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 8-2 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.

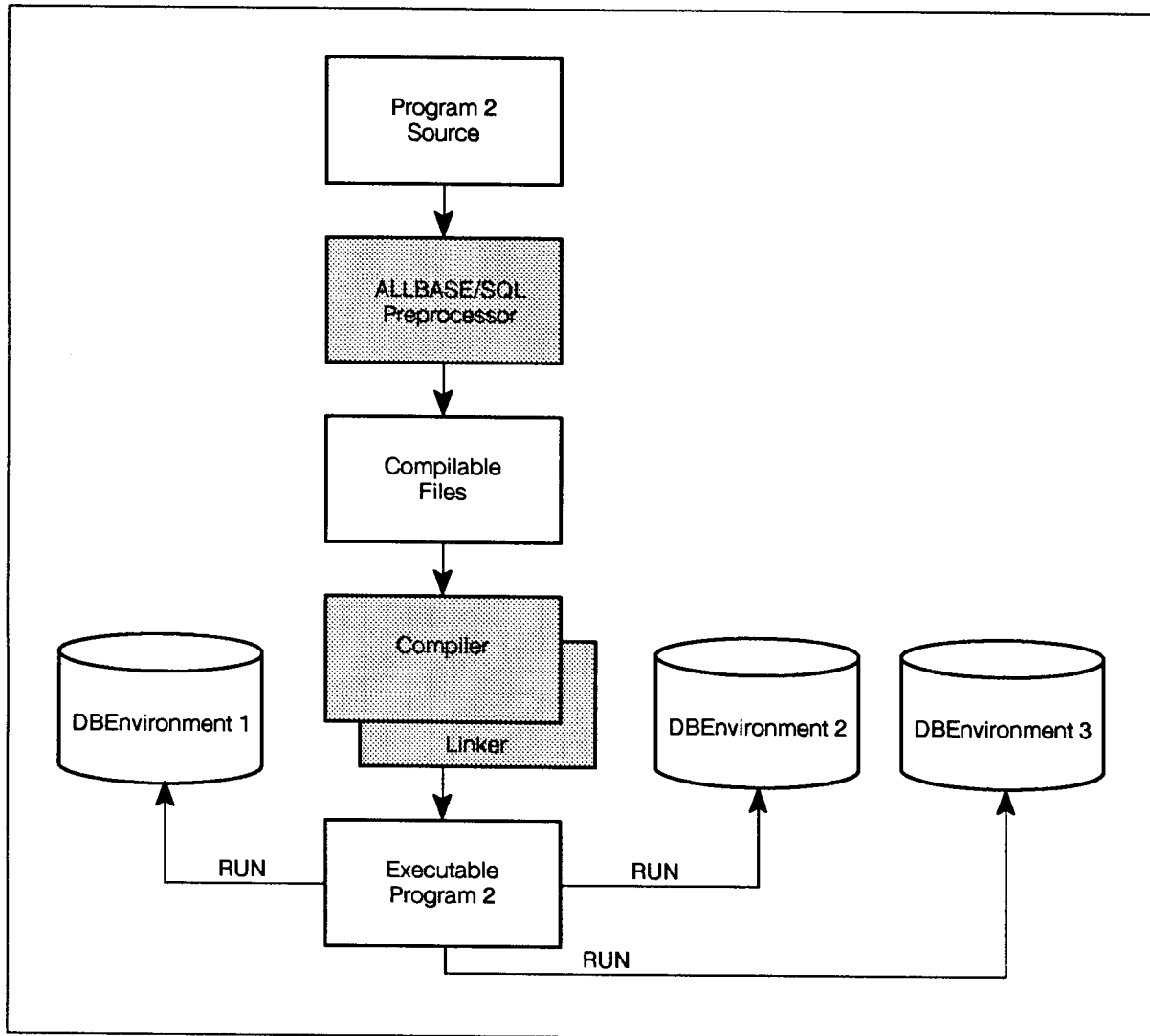


Figure 8-2. Creation and Use of a Program that has no Stored Module

### 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;
```

### 8-4 Using Dynamic Operations

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:

```
char          DynamicHostVar[2048];  
.  
.  
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 accommodate *either* non-queries or queries at run time. An example is shown later in this chapter in program cex10a (Figures 8-7, 8-8, 8-9).

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 8-3 illustrates a procedure hosting a dynamic UPDATE STATISTICS command that can be handled in this fashion.

Function 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 “Dynamic Commands of Unknown Format” illustrates how to handle this situation.

```
.
.
.
EXEC SQL BEGIN DECLARE SECTION;
char          CmdLine[100];
EXEC SQL END DECLARE SECTION;

char          TableName[50];
.
.
.
int UpdateStatistics()                                (1)
{
do {
    sprintf(CmdLine,"UPDATE STATISTICS FOR TABLE ");

printf("\n Enter name of table or / to terminate > ");    (2)
gets(TableName);
if (TableName[0] != '/') {

    sprintf(CmdLine+strlen(CmdLine)," %s;",TableName);    (3)

EXEC SQL EXECUTE IMMEDIATE :CmdLine;                        (4)

if (sqlca.sqlcode == 0) {
    EXEC SQL COMMIT WORK;                                (5)
}
else
    EXEC SQL ROLLBACK WORK;                              (6)

}
/* End of if TableName */
} while (TableName[0] != '/');

}
/* End of UpdateStatistics function */
.
.
.
```

Figure 8-3. Procedure Hosting Dynamic Non-Query Commands

## Using PREPARE and EXECUTE

Use the 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=0;i<MaxIterations;i++)
EXEC SQL EXECUTE MyNonQuery;
```

As soon as you process a COMMIT WORK or ROLLBACK WORK command, the temporary section is deleted.

---

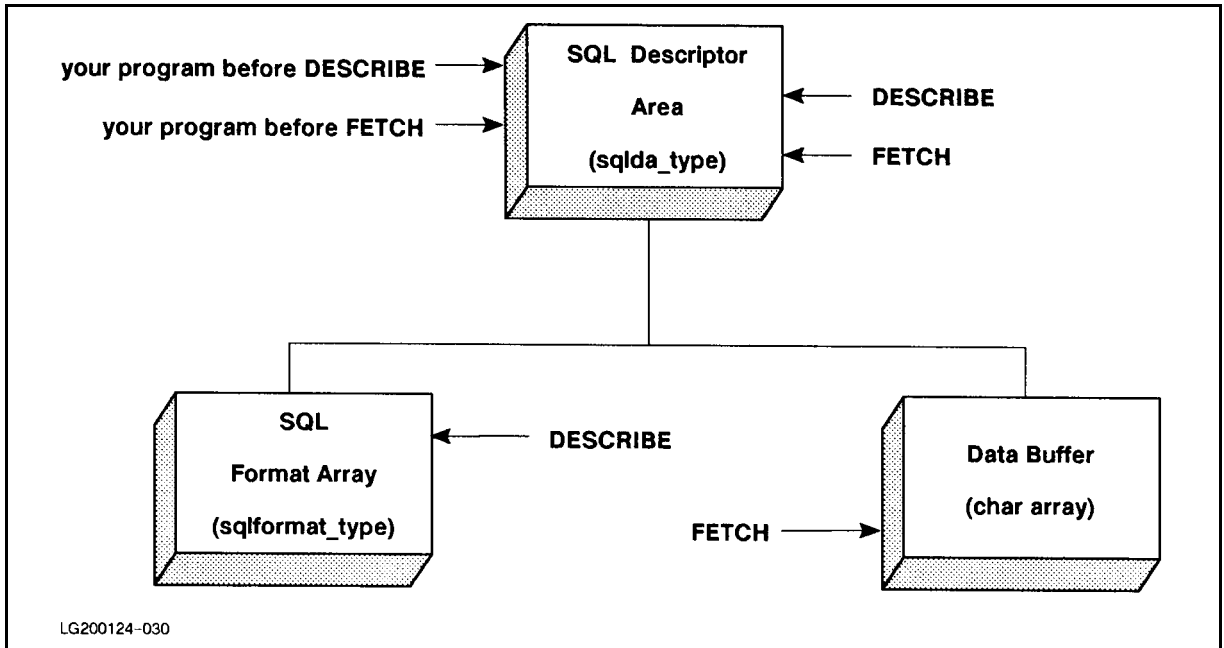
## Preprocessing of Dynamic Queries

Preprocessing 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 8-4 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 8-4. 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 query as follows:

- Define a host variable (or a string) 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 section 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 Chapter 6). 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 C preprocessor parses this command, it inserts a type declaration for this data structure into the modified source code file:

```
# if 0
  EXEC SQL INCLUDE SQLDA;
# endif
sqlda_type  sqlda;
```

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:

```
typedef struct {
  char      sqldaid[8];           reserved for ALLBASE/SQL
  int       sqldabe;             reserved for ALLBASE/SQL
  int       sqln;                number of format array records
  int       sqld;                number of columns
  sqlformat_type *sqlfmtarr;     format array address
  int       sqlnrow;             number of rows to FETCH
  int       sqlrrow;             number of rows fetched
  int       sqlrowlen;           bytes in each row
  int       sqlbuflen;           bytes in data buffer
  int       sqlrowbuf;           data buffer address
} sqlda_type;
```

Values are assigned to SQLDA fields by you or by ALLBASE/SQL, as summarized in Table 8-1.

**Table 8-1. SQLDA Fields**

<b>FIELD NAME</b>	<b>FIELD DESCRIPTION</b>	<b>C DATA TYPE</b>	<b>YOU SET BEFORE DESCRIBE</b>	<b>YOU SET BEFORE FETCH</b>	<b>ALLBASE/SQL SETS AT DESCRIBE</b>	<b>ALLBASE/SQL SETS AT FETCH</b>
sqldaid	reserved	char[8]				
sqldabc	reserved	int				
sqln	number of format array records (one record (column) per select list item)	int	X			
sqld	number of columns in query result (0 if non-query)	int			X	
sqlfmtarr	address of format array	sqlformat_type*	X			
sqlnrow	number of rows to FETCH into the data buffer	int		X		
sqlrrow	number of rows put into the data buffer	int				X
sqlrowlen	number of bytes in each row	int			X	
sqlbufen	number of bytes in the data buffer	int		X		
sqlrowbuf	address of data buffer	int		X		

## Setting Up the Format Array

You declare the format array as an array of records having the type `SqlFormat_Type`:

```
sqlformat_type    sqlfmts[NbrFmtRecords];
```

You 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:

```
#define 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:

```
#define NbrFmtRecords 6;
```

The definition for the type `SqlFormat_Type` appears in the full preprocessor generated type include file:

```
typedef struct {
    short  sqlnty;
    short  sqltype;
    short  sqlprec;
    short  sqlscale;
    int    sqltotalen;
    int    sqlvallen;
    int    sqlindlen;
    int    sqlvof;
    int    sqlnof;
    char   sqlname[20];
} sqlformat_type;
```

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. 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 8-2 explains the meaning of each field in a format array record.

**Table 8-2. Fields in a Format Array Record**

FIELD NAME	MEANING OF FIELD	C DATA TYPE
sqlnty	reserved; always set to 110	short
sqltype	<p>data type of column:</p> <p>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*</p> <p>* 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 and Date/Time/Interval character 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.</p>	short
sqlprec	precision of DECIMAL data	short
sqlscale	scale of DECIMAL data	short
sqltotalen	byte sum of sqlvallen, sqlindlen, indicator alignment bytes, and next data value alignment bytes	int
sqlvallen	number of bytes in data value, including a 4-byte prefix containing actual length of VARCHAR data	int
sqlindlen	<p>number of bytes null indicator occupies in the data buffer:</p> <p>0 bytes: column defined NOT NULL            2 bytes: column allows null values</p>	short
sqlvof	byte offset of value from the beginning of a row	int
sqlnof	byte offset of null indicator from the beginning of a row, dependent on the value of sqlindlen	int
sqlname	defined name of column or, for computed expression, EXPR	char[30]

## 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:

```
#define MaxDataBuff 2500 /* bytes in data buffer */
.
.
.
char DataBuffer[MaxDataBuff]; /* the data buffer */
```

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 sqlndlen 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:

```
sqlda.sqlnrow = ((sqlbuflen) / (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:

```

struct {
    int     strlen_col1; /* string length of column1 */
    char    column1[20];
    short   column2;
    int     column3;
    short   column3Ind; /* indicator variable */
} databuffer[MaxDataBuff];

```

When a column contains a VARCHAR data type, a 4 byte integer must be declared immediately before the variable for that column to hold the string length. The application itself needs to properly place the string terminator, ASCII 0, into the string.

```

struct SQLVarChar {
    int length; /* actual length of VARCHAR value */
    char VarCharCol[MaxColSize]; /* VARCHAR value */
};

```

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:

- 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:

```

#define MaxDataBuff 1000 /* bytes in data buffer */

char DataBuffer[MaxDataBuff]; /* the data buffer */

```

When the select list is known, you can define the data buffer as an array of records having the expected format:

```

#define MaxNbrRows 200
char DataBuffer[MaxNbrRows];

struct {
    Column1DataType Column1;
    Column2DataType Column2;
} DataBuffer[MaxNbrRows];

```

- You 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.



```

#define  NbrFmtRecords  1024    /* Columns expected */
.
.
sqlformat_type  sqlfmts[NbrFmtRecords]; /* sqlfmts is the format array */

```

- Use a host variable for the SELECT command, and pass it to ALLBASE/SQL in the PREPARE command:

```

EXEC SQL BEGIN DECLARE SECTION;
char    DynamicCommand[2048];
EXEC SQL END DECLARE SECTION;
.
.
EXEC SQL PREPARE Cmd1 FROM :DynamicCommand;

```

- Initialize two sqllda fields, **sqln** and **sqlfmtarr**. **sqln** is set to the size of the format array, and **sqlfmtarr** is set to its address.

```

.
sqllda.sqln      = NbrFmtRecords; /* Maximum select list elements*/
sqllda.sqlfmtarr = sqlfmts;      /* Format array address */

```

- Execute the DESCRIBE command:

```
EXEC SQL DESCRIBE Cmd1 INTO sqllda;
```

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
  sqllda.sqlbuflen = sizeof(DataBuffer); /* bytes in data buffer */
  sqllda.sqlrowbuf = int(DataBuffer);   /* data buffer address */
                                          /* cast as INT */
  sqllda.sqlnrow   = ((sqllda.sqlbuflen) / (sqllda.sqlrowlen));
                                          /* number of rows to FETCH */
end;

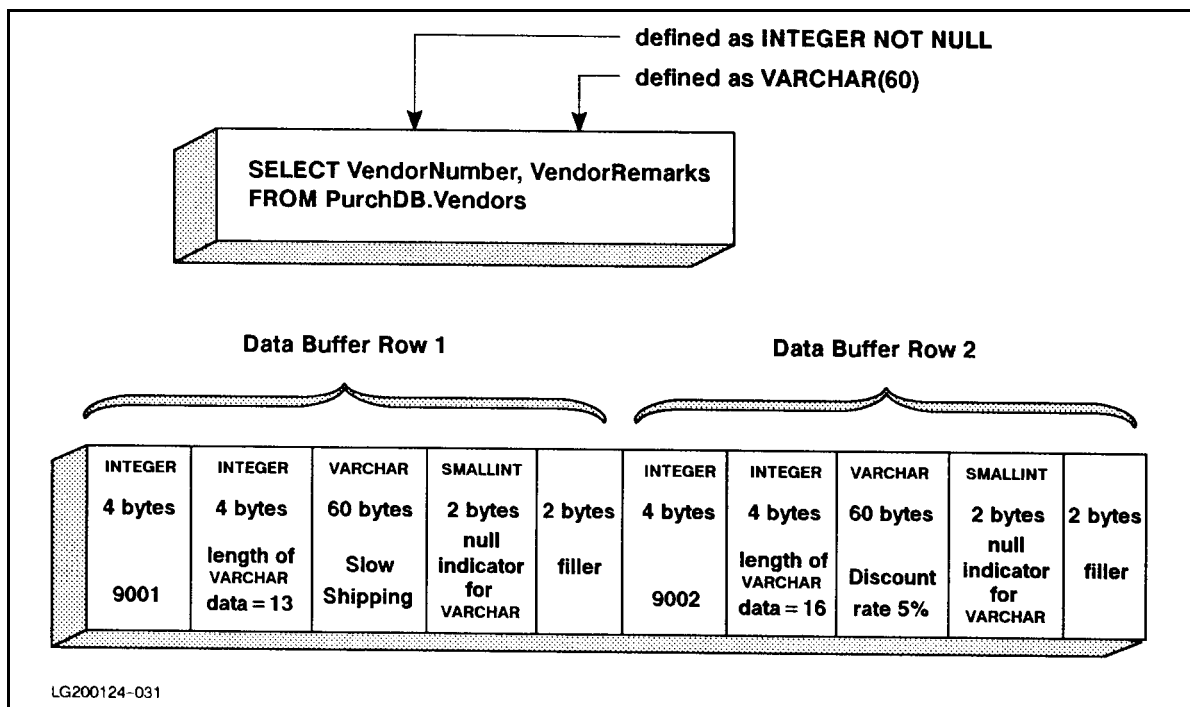
```

- 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 C 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 8-5, 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. Figure 8-5 illustrates the relationships between column definitions and the layout of data in the data buffer for a series 800 system.

**300/400** Data is aligned on 2 byte boundaries in the Series 300 and 400.

The figure illustrates the relationships between column definitions and the layout of data in the data buffer.



**Figure 8-5. Format of the Data Buffer for a Series 700 and 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.

```
do {
    EXEC SQL FETCH Cursor1 USING DESCRIPTOR sqlda;
    DisplayRow();
} while (sqlca.sqlcode !=100)
```

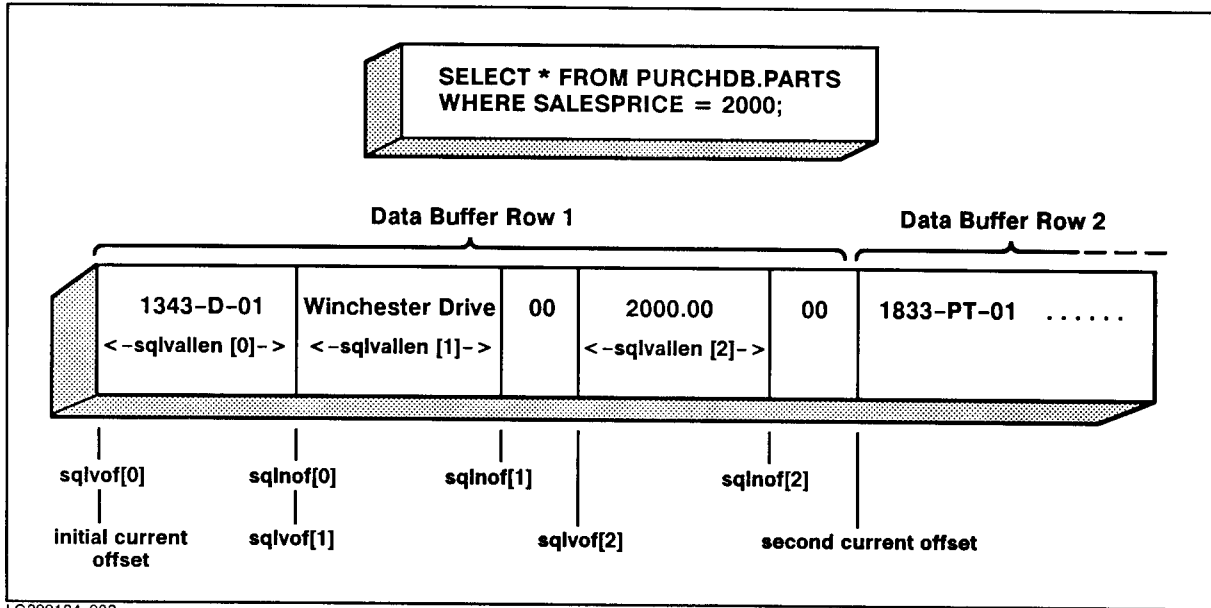
- If the query result is of unknown format, parse rows out of the data buffer after each execution of the `FETCH` command. The technique for parsing is shown in detail in the next section.

### 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 `cex10a`. The listing is found in Figure 8-9 in the following section, “`cex10a: Dynamic Queries 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.
- 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 `cex10a`. The data buffer shown is for the first query executed in the dialog in Figure 8-8.



**Figure 8-6. Parsing the Data Buffer in cex10a**

Program cex10a 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 + sqlfmts[i].SqlNOF;
```

To move a data value into a variant record, cex10a uses the following statement:

```
StrMove(sqlfmts[1].(SqlValLen, DataBuffer,  
CurrentOffset + sqlfmts[i].SqlVOF, OneColumn.CharData, 0);
```

---

## Preprocessing of 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 function NonQuery() and use the EXECUTE or EXECUTE IMMEDIATE command to execute it. If it is a query, you branch to function 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;

if (sqlda.sqld == 0) {
    Nonquery();
}
else if (sqlda.sqld > 0) {
    Query();
}
```

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, 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.

```

EXEC SQL BEGIN DECLARE SECTION;
char          DynamicCommand[2048];
EXEC SQL END DECLARE SECTION;
char          DynamicClause[80];
short int     Pos;
.
.
.
printf("\n Enter your SQL command or clause ");
printf("\n");
DynamicCommand[0] = '\0';
do {
    printf("\n > ");
    getline(DynamicClause);
    if (DynamicClause[0] != '/') {
        strcat(DynamicCommand, " ");
        strcat(DynamicCommand, DynamicClause);
        i = 0;
        while (DynamicClause[i] != '\0' && DynamicClause[i++] != ';'');
        if (DynamicClause[i-1] == ';'') {
            DynamicClause[0] = '/';
            DynamicClause[1] = '\0';
        }
    }
    else {
        DynamicCommand[0] = '/';
        DynamicCommand[1] = '\0';
    }
} while (DynamicClause[0] != '/');
.
.
EXEC SQL PREPARE SQLCommand FROM :DynamicCommand;

```

---

## Sample 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:

- `cex10a`, which contains statements for executing *any* dynamic command (non-query or query with unknown format).
- `cex10b`, 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.

### `cex10a`: Dynamic Commands of Unknown Format

Programs that host queries having query result formats unknown at programming time must use format array information to parse the data buffer. Figure 8-7 illustrates the logic for one such program, `cex10a`. The run-time dialog and source code for this program are shown in Figures 8-8 and 8-9, respectively.

Program `cex10a` executes function `ConnectDBE` (4) to invoke the `CONNECT` command (37) to start a DBE session in the sample database. It then executes the function named `Describe` (23). This function:

- 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 function `GetCommand` (25) and processes commands accepted from the user in that function until the user enters a slash (/).

Function `GetCommand` (25) accepts SQL commands into the host variable named `DynamicCommand`. This variable is declared (1) as `char DynamicCommand[2048]` to allow for a dynamic command of up to 2048 bytes, including the semicolon. `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.

After SQL command entry is complete, control returns to function `Describe` (23), which:

- Starts a transaction by executing 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 (29). If this value is 0, the command is not a query and function `NonQuery` (28) is invoked to execute the command. If the `sqlda.sqld` value is not 0, function `Query` (29) is invoked to execute the command.

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 function `Describe`, which calls both function `NonQuery` and function `Query`, which follow after function `Describe` sequentially in the source code.

Function Query (29):

- 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.sqlbufen` (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 function `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.

Function `DisplaySelect` (36) parses the data buffer after each FETCH operation and displays the fetched rows:

- The function keeps track of the beginning of each row by using a local variable, `CurrentOffset`, as a pointer. `CurrentOffset` is initialized to 0 (10) at the beginning of function `DisplaySelect`.
- Column headings are written from the `sqlfmts[x].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 `sqlfmts[i].sqlindlen` field of each column's format array record is examined (14) to determine whether a null value may exist.
- If a column can contain null values, `sqlfmts[i].sqlindlen` is greater than zero, and the function 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`. `Sqlfmts[i].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.
- `DataBuffer` and `NullIndOffset` are used to determine whether or not a null value exists. If a null value exists, the function displays the message `Column is NULL` (17).
- 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` function (30) is computed by adding the current value of `sqlfmts[i].sqlvof` to the current value of `CurrentOffset`. `Sqlfmts[i].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 `sqlfmts[i].sqlvallen`. `OneColumn.CharData` is one of the variations of the variant record, `GenericType` (9).
- The `GenericType` type definition 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



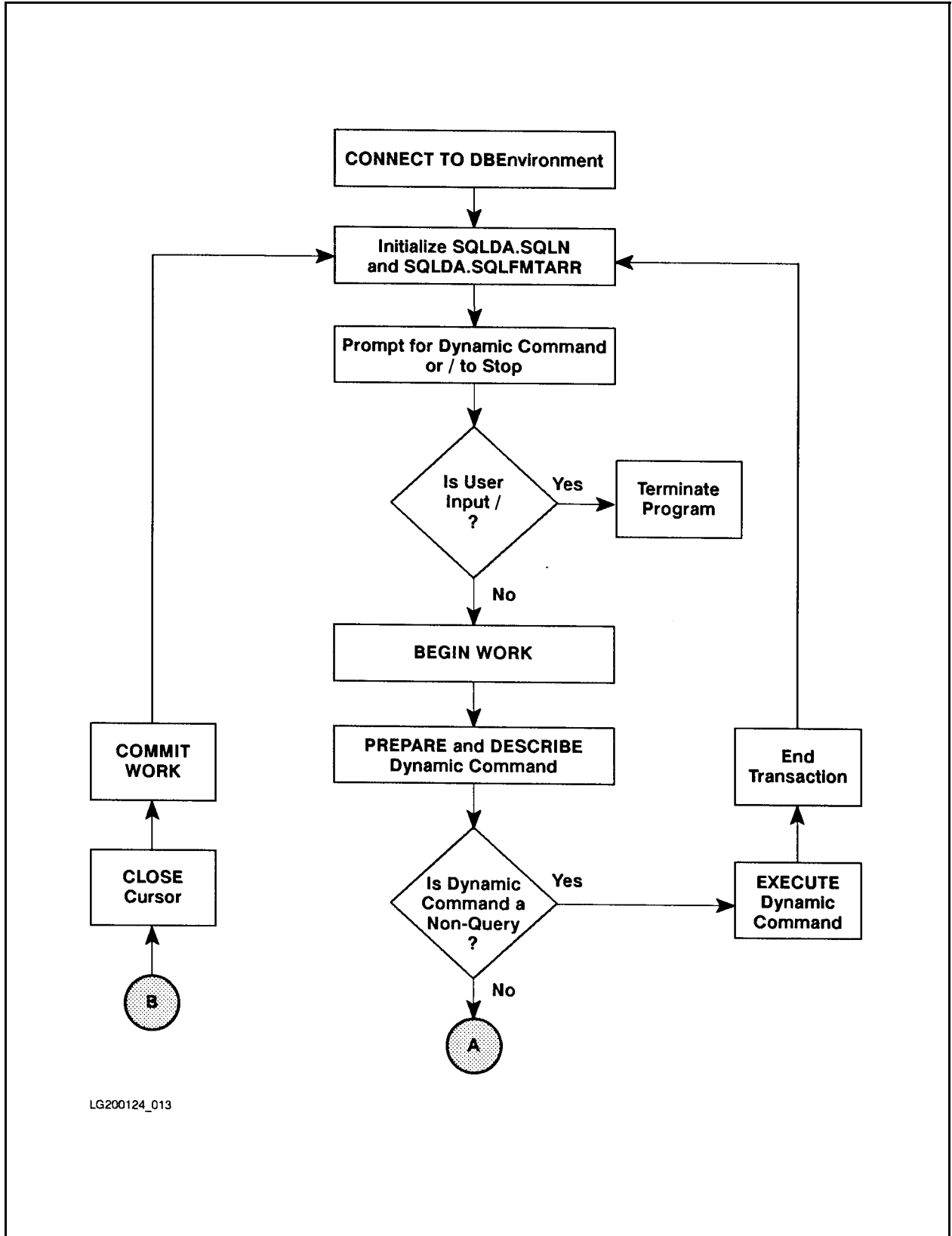
types. The record variation used depends on the value of `sqlfmts[i].sqltype` (19), the format array record field describing the data type of a select list item. In the case of DECIMAL data, a function named `BCDToString` (2) converts the binary coded decimal (BCD) information in the data buffer into 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, function `Query` calls function `EndTransaction` (7) to process a COMMIT WORK command. Thus each dynamic query hosted by this program is executed in a separate transaction.

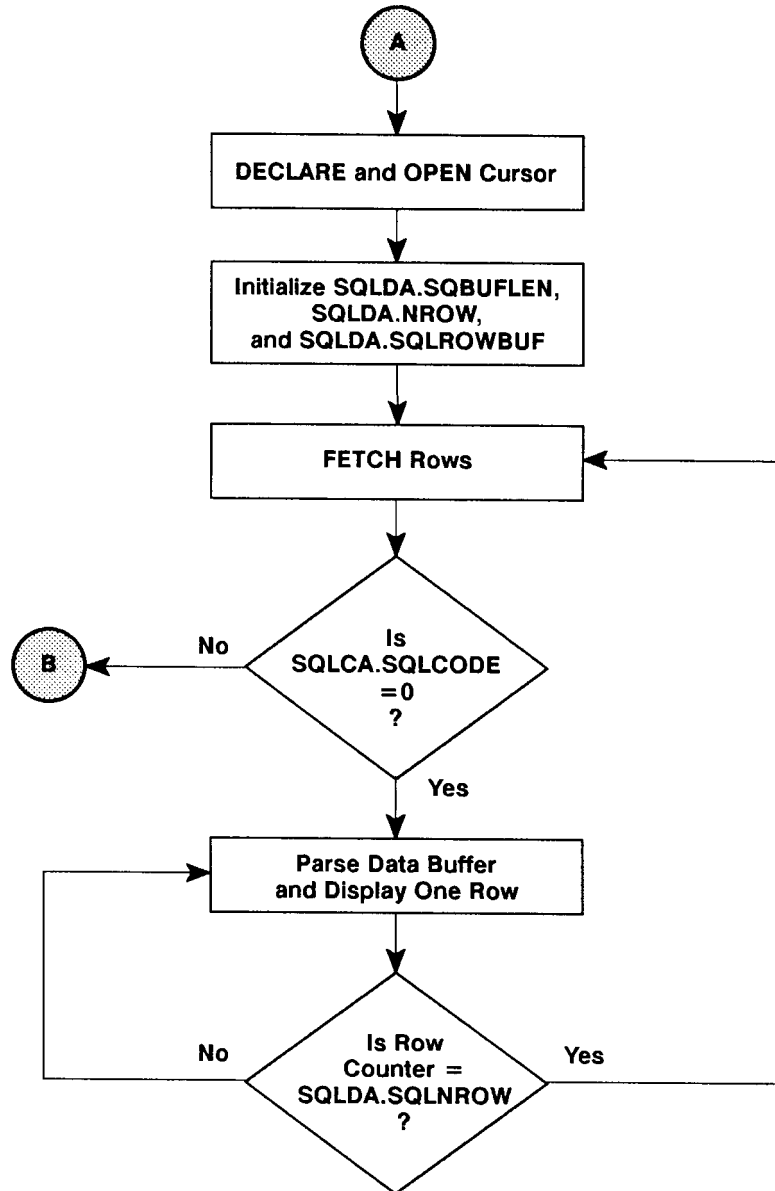
To determine whether each SQL command executed successfully, the program examines the value of `sqlca.sqlcode` after each SQL command is executed. Function `SQLStatusCheck` (3) is invoked to display one or more messages from the ALLBASE/SQL message catalog. Any other action taken depends on the SQL command:

- If the CONNECT command fails, function `ConnectDBE` (4) sets the Connect flag to FALSE, calls function `SQLStatusCheck`, and then terminates the program.
- If the BEGIN WORK command fails, function `BeginTransaction` (6) calls `SQLStatusCheck` to display messages, then calls function `ReleaseDBE` (5) to end the DBE session. The program then terminates because function `Describe` (23) sets `DynamicCommand` to a slash (31).
- If other SQL commands fail, function `SQLStatusCheck` terminates the program whenever the error is serious enough to return an `sqlca.sqlcode` less than -14024.



LG200124\_013

Figure 8-7. Flow Chart of Program cex10a



LG200124\_014

Figure 8-7. Flow Chart of Program cex10a (page 2 of 2)

```
C program illustrating dynamic command processing -- cex10a

Event List:
CONNECT TO PartsDBE
Prompt for any SQL command
BEGIN WORK
PREPARE
DESCRIBE
If the command is a non-query command, execute it;
  otherwise execute the following:
DECLARE CURSOR
OPEN Cursor
FETCH a row
CLOSE CURSOR
COMMIT WORK
Repeat the above ten steps until the user enters a /
RELEASE 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 your SQL command or clause >

> SELECT * FROM PURCHDB.PARTS WHERE SALESPRICE = 2000;

Begin Work
Prepare SELECT * FROM PURCHDB.PARTS WHERE SALESPRICE = 2000;

Describe
A Query SQL command was entered.

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 8-8. Run Time Dialog of Program cex10a

```
Enter your SQL command or clause >

> DELETE FROM PURCHDB.PARTS WHERE PARTNUMBER = '1343-D-01';

Begin Work
Prepare DELETE FROM PURCHDB.PARTS WHERE PARTNUMBER = '1343-D-01';

Describe
A Non-Query SQL command was entered.
Execute
The 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 your SQL command or clause >
> SELECT * FROM PURCHDB.PARTS WHERE SALESPRICE = 2000;

Begin Work
Prepare SELECT * FROM PURCHDB.PARTS WHERE SALESPRICE = 2000;

Describe
A Query SQL command was entered.

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 your SQL command or clause > /

Release PartsDBE
```

Figure 8-8. Run Time Dialog of Program cex10a (page 2 of 2)



```

/* DataBuffer is the buffer containing retrieved data as a result */
/* of a dynamic SELECT. */

char    DataBuffer[MaxDataBuff];
boolean Abort;

struct SQLVarChar {
    int    Length;
    char    VarCharCol[MaxColSize];
};

main()    /* Beginning of Program */
{
printf("\nC program illustrating dynamic command processing -- cex10a");
printf("\n");
printf("\nEvent List:");
printf("\n  CONNECT TO PartsDBE");
printf("\n  Prompt for any SQL command");
printf("\n  BEGIN WORK");
printf("\n  PREPARE");
printf("\n  DESCRIBE");
printf("\n  If command is a non-query command, EXECUTE it");
printf("\n  Otherwise execute the following:");
printf("\n  DECLARE CURSOR");
printf("\n  OPEN Cursor");
printf("\n  FETCH a row");
printf("\n  CLOSE Cursor");
printf("\n  COMMIT WORK");
printf("\n  Repeat the above ten steps");
printf("\n  RELEASE PartsDBE\n");

if (ConnectDBE()) {
    Describe();
    ReleaseDBE();
    printf("\n");
}
else
    printf("\nError: Cannot Connect to PartsDBE");
printf("\n");
}    /* End of Main Program */

/* Function BCDToString converts a binary field in the "DataBuffer" */
/* buffer to its ASCII representation.  Input parameters are */
/* the Length, Precision and Scale.  The input decimal field is passed */
/* via "DataBuffer" and the output String is passed via "result".  */

```

4

23

**Figure 8-9. Program cex10a: Dynamic Commands of Unknown Format (page 2 of 11)**

```

int BCDToString(DataBuffer, Length, Precision, Scale, Result0)
char  DataBuffer[];
short Length, Precision, Scale;
char  Result0[];
{
#define  hexd          '0123456789ABCDEF'
#define  ASCIIZero    '0'
#define  PlusSign     12
#define  MinusSign    13
#define  Unsigned     14
#define  btod(d,i)    ((i&1)?((d[i/2])&0xf):((d[i/2]>>4)&0xf))

    int      i;
    int      DecimalPlace;
    int      PutPos=0;
    int      DataEnd;
    int      DataStart;
    boolean  done;
    char      space[MaxStr];
    char      *Result;

    Result = space;
    DataEnd = (Length*2) - 2;
    DataStart = (DataEnd - Precision);
    for (i = 0; i < MaxStr; i++) Result[i] = '\0';
    DecimalPlace = (Precision-Scale);

    /* convert decimal to character String */
    if (DecimalPlace == 0) Result[PutPos++] = '.';

    /* convert each Nibble into a character */
    for (i = DataStart; i <= DataEnd; i++) {
        Result[PutPos] = ASCIIZero + btod(DataBuffer,i);
        if (PutPos == DecimalPlace) Result[++PutPos] = '.';
        PutPos++;
    }

    i = 0;
    done = FALSE;
    while (i<strlen(Result) && Result[i]!='0') ++Result;

    if (Result[0] == '\0')
        Result[0] = '0';
    else {
        /* place a zero at the left of the decimal point */
        if (Result[0] == '.') StrInsert('0', Result);
    }
}

```

2

Figure 8-9. Program cex10a: Dynamic Commands of Unknown Format (page 3 of 11)



```

    /* insert sign */
    switch (btod(DataBuffer,DataEnd + 1)) {
        case PlusSign: StrInsert(' ', Result);
                        break;
        case MinusSign: StrInsert('-', Result);
                        break;
        default:        break;
    } /* End switch */
} /* End else */
strcpy(Result0, Result);
} /* End BCDToString */

int getline(linebuff) /*Function to get a line of characters */
char linebuff[80];
{
while (strlen(gets(linebuff)) ==0);
} /* End of function to get a line of characters */

int SQLStatusCheck() /* Function to Display Error Messages */
{
Abort = FALSE;
if (sqlca.sqlcode < DeadLock) Abort = TRUE;

do {
EXEC SQL SQLEXPLAIN :SQLMessage;
printf("\n");
printf("%s\n",SQLMessage);
} while (sqlca.sqlcode != 0);

if (Abort) {

EXEC SQL COMMIT WORK RELEASE;
DynamicCommand[0] = '/';
DynamicCommand[1] = '\0';
}

} /* End SQLStatusCheck Function */

int ConnectDBE() /* Function to Connect to PartsDBE */
{
boolean Connect;
printf("\nConnect to PartsDBE");
EXEC SQL CONNECT TO 'PartsDBE';

```

**Figure 8-9. Program cex10a: Dynamic Commands of Unknown Format (page 4 of 11)**

```

Connect = TRUE;
if (sqlca.sqlcode != OK) {

    Connect = FALSE;
    SQLStatusCheck();
} /* End if */
return(Connect);
} /* End of ConnectDBE Function */

int ReleaseDBE() /* Function to Release PartsDBE */ 5
{
printf("\nRelease PartsDBE");
EXEC SQL RELEASE;
if (sqlca.sqlcode != OK) SQLStatusCheck();

} /* End ReleaseDBE Function */

boolean BeginTransaction() /* Function to Begin Work */ 6
{
boolean BeginTransaction;
printf("\n");
printf("\nBegin Work");
EXEC SQL BEGIN WORK;
if (sqlca.sqlcode != OK) {

    BeginTransaction = FALSE;
    SQLStatusCheck();
    ReleaseDBE(); 5
}
else
    BeginTransaction = TRUE;
return(BeginTransaction);

} /* End BeginTransaction Function */

int EndTransaction() /* Function to Commit Work */ 7
{

printf("\n");
printf("\nCommit Work");
EXEC SQL COMMIT WORK;
if (sqlca.sqlcode != OK) SQLStatusCheck();

} /* End EndTransaction Function */

```

Figure 8-9. Program cex10a: Dynamic Commands of Unknown Format (page 5 of 11)

```

/* Function DisplaySelect deblocks the result of the dynamic */
/* SELECT in "DataBuffer". */

int DisplaySelect()
{
    typedef union gt {
        char          CharData[MaxColSize];
        char          VarCharData[MaxColSize];
        int           IntegerData;
        short         SmallIntData;
        double        FloatData;
        float         DecimalData;
    } GenericType;

    short           CurrentOffset;
    short           NullIndOffset;
    GenericType     OneColumn;
    char            DecString[20];
    boolean         IsNull;
    short           n,i,j,x;          /* local loop counters */

    CurrentOffset = 0;

    for (x = 0; x < sqlda.sqld; x++) { /* display column names */
        printf("%s | ",sqlfmts[x].sqlname);
    }
    printf("\n");

    for (n = 0; n < sqlda.sqlrrow; n++) { /* for each FETCHed row */
        for (i = 0; i < sqlda.sqld; i++) { /*for each column in a FETCHed row*/
            /* 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 (sqlfmts[i].sqlindlen > 0) {
                NullIndOffset = CurrentOffset + sqlfmts[i].sqlnof;

                if ((DataBuffer[NullIndOffset] == '\0') &&
                    (DataBuffer[NullIndOffset+1] == '\0')) {
                    IsNull = FALSE;
                }
            }
            else
                IsNull = TRUE;
        }
    }
}

```

**Figure 8-9. Program cex10a: Dynamic Commands of Unknown Format (page 6 of 11)**

```

} /* End if sqlfmts[i].sqlindlen > 0 .. */

if (IsNull) {
    printf(" Column is NULL |");
}
else {

    /* Now bring down the actual value of this column. */

    StrMove(sqlfmts[i].sqlvallen,DataBuffer,
            CurrentOffset + sqlfmts[i].sqlvof, OneColumn.CharData, 0);

    switch (sqlfmts[i].sqltype) {
        case 0: /* Integer number */
            switch (sqlfmts[i].sqlvallen) {
                case 2: printf("%d | ",OneColumn.SmallIntData);
                        break;
                case 4: printf("%d | ",OneColumn.IntegerData);
                        break;
            } /* End switch statement */
            break;
        case 2: /* fixed-length character */
        case 8: /* fixed-length native character */
            for (j = 0; j < sqlfmts[i].sqlvallen; j++)
                printf("%c",OneColumn.CharData[j]);
            printf(" | ");
            break;
        case 3: /* variable-length char */
        case 9: /* variable-length native char */
            for (j = 4; j < sqlfmts[i].sqlvallen; j++)
                printf("%s | ",OneColumn.VarCharData[j]);
            printf(" | ");
            break;
        case 4: /* floating point */
            printf("%f | ",OneColumn.FloatData);
            break;
        case 5: /* Packed decimal */
            BCDToString(OneColumn.CharData, sqlfmts[i].sqlvallen,
                sqlfmts[i].sqlprec, sqlfmts[i].sqlscale, DecString);
            printf("%s | ",DecString);
            break;
        default: printf("SQLType = %1s\n",sqlfmts[i].sqltype);
            break;
    } /* End switch statement */

} /* End if IsNull else */

```

Figure 8-9. Program cex10a: Dynamic Commands of Unknown Format (page 7 of 11)

```

        } /* End for i/with sqlfmts[i] ... */

        CurrentOffset = CurrentOffset + sqlda.sqlrowlen;

        printf("\n");

    } /* End for n = ... */

    printf("\n");

} /* End of DisplaySelect function */

int GetCommand()
{
    char        DynamicClause[80];
    short      i;

    printf("\n");
    printf("\nYou may enter any SQL command or a '/' to STOP the program.");
    printf("\nThe command can be continued on the next line.  The command");
    printf("\nmust be terminated with a semicolon.");
    printf("\n");
    printf("\nEnter your SQL command or clause ");
    printf("\n");
    DynamicCommand[0] = '\0';      /* @001 */
    do {
        printf("\n >");
        getline(DynamicClause);
        if (DynamicClause[0] != '/') {
            strcat(DynamicCommand, " ");
            strcat(DynamicCommand, DynamicClause);
            i = 0;
            while (DynamicClause[i] != '\0' && DynamicClause[i++] != ';');
            if (DynamicClause[i-1] == ';') {

                DynamicClause[0] = '/';      /* @001 */
                DynamicClause[1] = '\0';      /* @001 */
            }
        }
    }
    else {

        DynamicCommand[0] = '/';
        DynamicCommand[1] = '\0';      /* @001 */
    }
} while (DynamicClause[0] != '/'); /* End do */      /* @001 */
} /* End of GetCommand function */

```

**Figure 8-9. Program cex10a: Dynamic Commands of Unknown Format (page 8 of 11)**

```

int Describe() /* Describe Function */
{
    /* set up SQLDA fields */
    sqlda.sqln = NbrFmtRecords; /* number of columns expected */
    sqlda.sqlfmtarr = sqlfmts;

do {
    GetCommand();

    if (DynamicCommand[0] != '/') { /* @001 */

        if (BeginTransaction()) {

            printf("\nPrepare");
            printf("%s\n",DynamicCommand);
            EXEC SQL PREPARE CMD1 FROM :DynamicCommand;
            if (sqlca.sqlcode != OK) {
                SQLStatusCheck();
                EndTransaction();
            }
            else {

                printf("\nDescribe");
                EXEC SQL DESCRIBE CMD1 INTO SQLDA;
                if (sqlca.sqlcode != OK) {
                    SQLStatusCheck();
                    EndTransaction();
                }
                else {

                    if (sqlda.sqld == 0) NonQuery();
                    else
                        Query();
                } /* End if sqlca.sqlcode != OK after DESCRIBE */

            } /* End if sqlca.sqlcode != OK after PREPARE */

        } /* End if BeginTransaction */
        else { /* BeginTransaction failed; */

            DynamicCommand[0] = '/'; /* force logical to */ /* @001 */
            DynamicCommand[1] = '\0'; /* Describe function */ /* @001 */
        }
    }
}

```

Figure 8-9. Program cex10a: Dynamic Commands of Unknown Format (page 9 of 11)

```

    } /* End if DynamicCommand */

    } while (DynamicCommand[0] != '/'); /* End do */ /* @001 */

} /* End of Describe function */

int NonQuery()
{
    printf("\nA Non Query SQL command was entered.");
    printf("\nExecute");
    EXEC SQL EXECUTE CMD1;
    if (sqlca.sqlcode != OK) {
        SQLStatusCheck();
        EXEC SQL ROLLBACK WORK;
    }
    else {

        printf("\nThe Non-Query Command Executed Successfully.");
        EndTransaction();
    }
} /* End of NonQuery function */

int Query()
{
    short      RowLength;
    short      i;

    printf("\nA Query SQL command was entered.");
    printf("\n");
    printf("\nNumber of columns:  %2d",sqllda.sqld);
    printf("\n");

    EXEC SQL DECLARE CURSOR1 CURSOR FOR CMD1;
    EXEC SQL OPEN CURSOR1;
    if (sqlca.sqlcode != OK) SQLStatusCheck();
    else {

        sqllda.sqlbuflen = sizeof(DataBuffer);
        sqllda.sqlnrow = ((sqllda.sqlbuflen) / (sqllda.sqlrowlen));
        sqllda.sqlrowbuf = DataBuffer;

        while (sqlca.sqlcode == 0) {

            EXEC SQL FETCH CURSOR1 USING DESCRIPTOR SQLDA;
            if (sqlca.sqlcode != OK) {

```

**Figure 8-9. Program cex10a: Dynamic Commands of Unknown Format (page 10 of 11)**

```

        if (sqlca.sqlcode == EndOF) {
            printf("\nRow not found or no more rows!");
        }
        else
            SQLStatusCheck();
    }
    else
        DisplaySelect();
}

/* End of while sqlca.sqlcode = 0 */

EXEC SQL CLOSE CURSOR1;
if (sqlca.sqlcode != OK) SQLStatusCheck();
} /* End of if OPEN CURSOR is OK */

EndTransaction();

int StrMove(n,s1,p1,s2,p2)
int n, p1, p2;
char s1[], s2[];
{
    int i = 1;

    while (i++ <= n)
        s2[p2++] = s1[p1++];
} /* StrMove */

int StrInsert(c, string)
char c;
char *string;
{
    char *temp;

    temp = malloc(MaxStr);
    strcpy(temp, string);
    *string++ = c;
    while ((*string++ = *temp++) != '\0');
} /* StrInsert */

```

Figure 8-9. Program cex10a: Dynamic Commands of Unknown Format (page 11 of 11)



## cex10b: Program Using Dynamic Queries of Known Format

In some applications, you may know the format of a query result in advance, but may still 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 a 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 in which it has been returned.

Program cex10b, whose flow chart is shown in Figure 8-10, whose execution is illustrated in Figure 8-11, and whose source code appears in Figure 8-12, 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 the tables in the 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 schema 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 (9) in host variable CmdLine. 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):

```
printf(CmdLine,"SELECT OWNER, NAME, DBEFILESET, RTYPE FROM SYSTEM.TABLE\  
WHERE TYPE = 0 AND OWNER = '%s';",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 enters ALL in response to the owner name prompt. In this case, the program uses the following form of the dynamic query (13):

```
printf(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, sqlda.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 several sqllda fields (19) as follows:

- The sqllda.sqlnrow field is set to 300, as 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.
- The sqllda.sqlbufen field is set to the size of the data buffer. In this program, the data buffer for the first query is a structure 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 their corresponding SYSTEM.TABLE columns. Note also that each element in the array is declared as the same size as its corresponding column in the system table and not one character larger. It is up to the program to insert the ASCII 0 null character in the correct location to indicate the end of a character string.
- The sqllda.sqlrowbuf field is set to the address of the data buffer.

After initializing the required fields in the sqllda, 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[2] 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 StrCpy function (24) to move CHAR values to char array variables so that other C string functions 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:

```
sprintf(CmdLine,"SELECT COLNAME, LENGTH, TYPECODE, NULLS, PRECISION,\
SCALE FROM SYSTEM.COLUMN WHERE OWNER = '%s' AND TABLENAME = \
'%s';",OwnerName, TableName);
```

These names are obtained from the Owner and Table values in the TableList array (4).

After each version of the second query is dynamically preprocessed (27), the program initializes two sqllda 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 the necessary sqllda values (31):

- The sqllda.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 sqllda.sqlbufen field is set to the size of the data buffer. The data buffer for the second query is a structure array of records named ColumnList (5).
- The sqllda.sqlrowbuf field is set to the address of the data buffer.

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 because 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 (34):

```
CREATE LockMode TABLE OwnerName.TableName,  
  (ColumnList[1].ColName  TypeInfo  NullInfo,  
   ColumnList[2].ColName  TypeInfo  NullInfo,  
   .  
   .  
   .  
   ColumnList[j].ColName  TypeInfo  NullInfo) IN TableList[i].FileSet;
```

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 switch statement (33) based on the value ALLBASE/SQL put in TableList[i].LockMode. The SYSTEM.TABLE view stores the automatic locking mode for tables as an integer from 1 through 3. The switch statement equates these codes with the expressions that must appear in the CREATE TABLE command.
- TypeCode is generated in a switch statement (35) based on the value ALLBASE/SQL put in ColumnList[i].TypeCode. The SYSTEM.COLUMN view stores the data type of each column as an integer from 0 through 5. The switch statement equates these codes with the expressions that must appear in the CREATE TABLE command.
- Nulls is generated from the null indicator ALLBASE/SQL returned to ColumnList[i].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 with a COMMIT WORK RELEASE command (38).

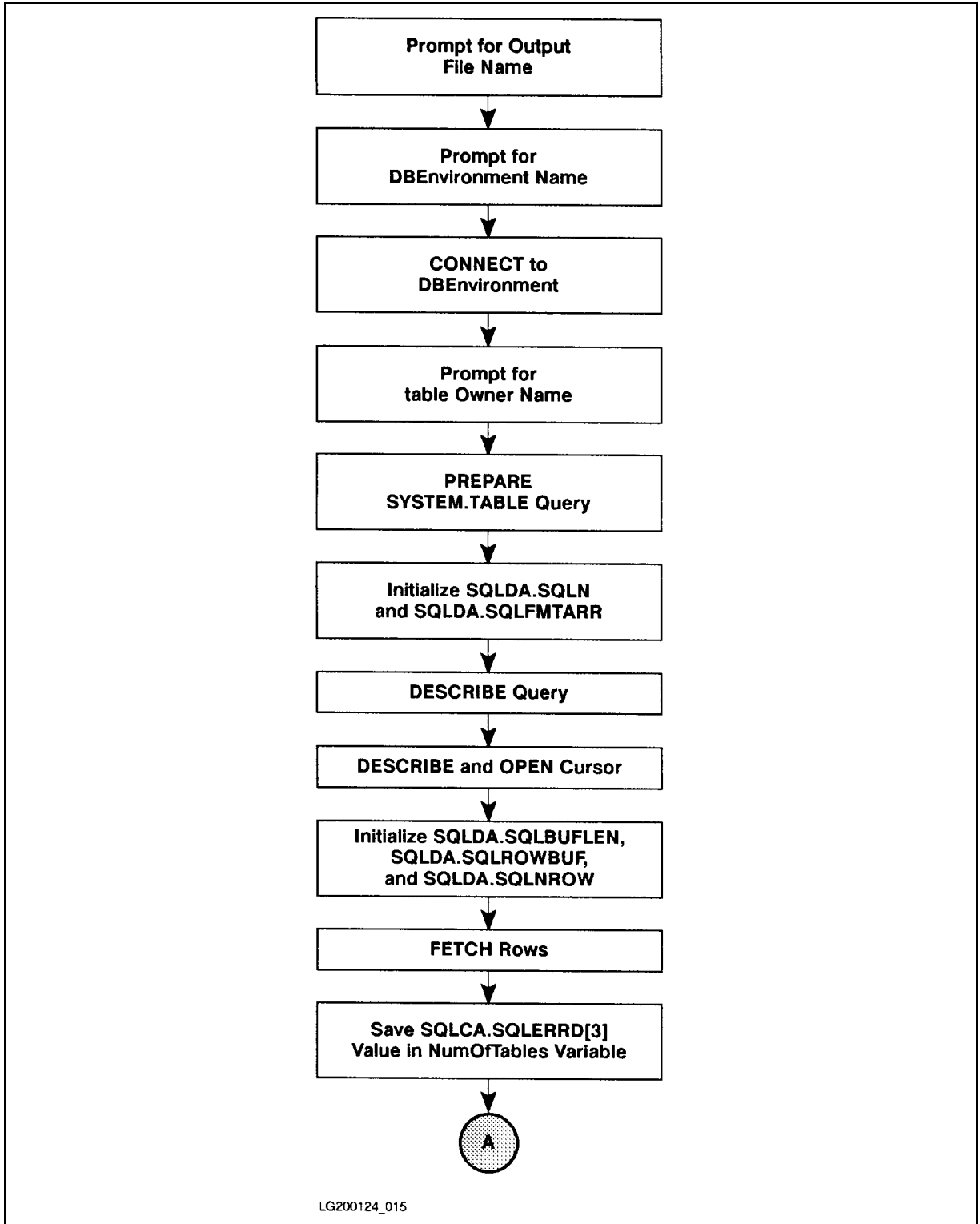


Figure 8-10. Flow Chart of Program cex10b

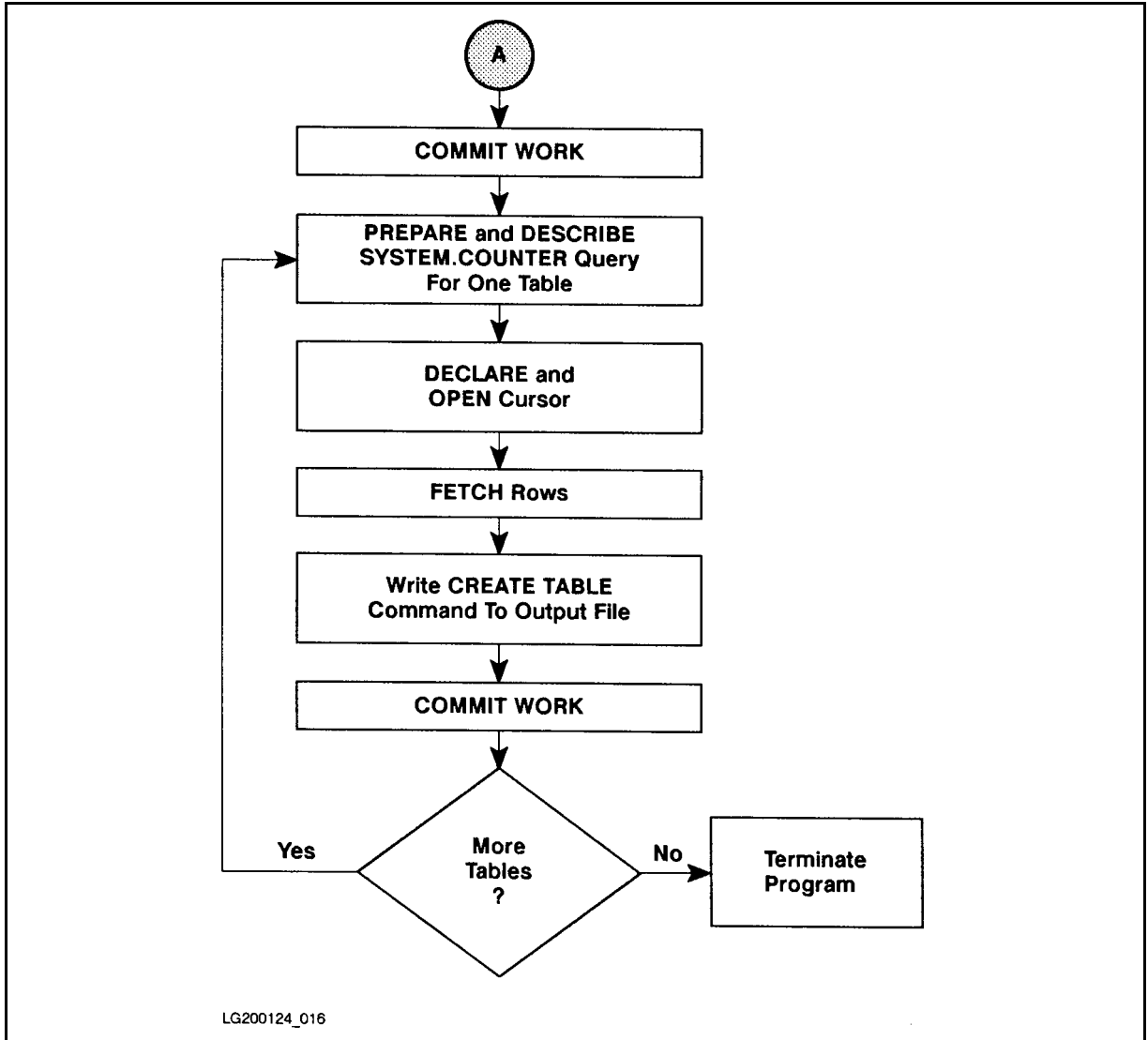


Figure 8-10. Flow Chart of Program cex10b (page 2 of 2)

```
C program illustrating dynamic command processing -- cex10b

ALLBASE/SQL/HP-UX 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
:
```

Figure 8-11. Run Time Dialog of Program cex10b

```

/* Program cex10b */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* This program generates an ISQL command file that will re-create */
/* tables within a particular DBEnvironment. This program must be */
/* preprocessed; however, it does not need to be installed.      */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

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

#define OK 0
#define MaxNbrTables 300
#define MaxNbrColumns 255
#define mode 0700
#define NbrFmtRecords 32

sqlca_type sqlca; /* SQL Communication Area */
sqllda_type sqllda; /* SQL Describe Area */

sqlformat_type sqlfmfts[NbrFmtRecords]; /* declaration of format nodes */

char FileName[15];
char OwnerName[21];
char TableName[21];
char DBEFileSet[21];
char ColumnName[21];
char DBENAME[128];
char OneLine[81];
int FileNum;
short i;
short j;
short NumOfTables;

struct {
    char Owner[20];
    char Table[20];
    char FileSet[20];
    short LockMode;
} TableList[MaxNbrTables];

```

1  
2

4

Figure 8-12. Program cex10b: Dynamic Queries of Known Format

```

struct {
    char          ColName[20];
    int           Length;
    short        TypeCode;
    short        Nulls;
    short        Precision;
    short        Scale;
} ColumnList[MaxNbrColumns];

        /* Begin Host Variable Declarations */
EXEC SQL BEGIN DECLARE SECTION;
char      CmdLine[200];
char      SQLMessage[133];
EXEC SQL END DECLARE SECTION;
        /* End Host Variable Declarations */

main() /* Beginning of Program */
{
printf("\n C program illustrating dynamic command processing -- cex10b");
printf("\n");
printf("\n ALLBASE/SQL/HP-UX SCHEMA Generator for Tables");
printf("\n");
printf("\n Event List:");
printf("\n      Prompt for the name of the schema file to create");
printf("\n      Prompt for the name of the DBEnvironment");
printf("\n      Prompt for the owner name");
printf("\n      Generate schema file");
printf("\n");

printf("\n Enter name of schema file to be generated > ");
scanf("%s",FileName);

FileNum = unlink(FileName);
FileNum = creat(FileName,mode);

printf("\n Enter name of DBEnvironment > ");
scanf("%s",DBENAME);

sprintf(CmdLine,"CONNECT TO '%s';",DBENAME);

EXEC SQL EXECUTE IMMEDIATE :CmdLine;
if (sqlca.sqlcode != OK) {
    printf("\n Could not CONNECT to DBEnvironment!");
    EXEC SQL SQLEXPLAIN :SQLMessage;
    printf("%s\n",SQLMessage);
    goto a9999;
}
}

```

Figure 8-12. Program cex10b: Dynamic Queries of Known Format (page 2 of 7)



```

printf("\n Enter database owner name or ALL for all owners > "); 11
scanf("%s",OwnerName);

    for (i = 0; i <= strlen(OwnerName); i++) {    /* Upshift OwnerName */
        if (islower(OwnerName[i])) {
            OwnerName[i] = toupper(OwnerName[i]); 12
        }
    }

response = "ALL";

if (strcmp(response,OwnerName) == 0) {
sprintf(CmdLine,"SELECT OWNER,NAME,DBEFILESET,RTYPE FROM SYSTEM.TABLE\
WHERE TYPE = 0 AND OWNER <> 'SYSTEM'"); 13
}
else {
sprintf(CmdLine,"SELECT OWNER,NAME,DBEFILESET,RTYPE FROM SYSTEM.TABLE\
WHERE TYPE = 0 AND OWNER = '%s';",OwnerName); 14
}

EXEC SQL PREPARE SelectCmd1 FROM :CmdLine; 15
if (sqlca.sqlcode != OK) {
    printf("\n Problem PREPARING the SELECT #1 command!");
    EXEC SQL SQLEXPLAIN :SQLMessage;
    printf("%s\n",SQLMessage);
    goto a9999;
}

/* set up SQLDA fields */
    sqlda.sqlfmtarr = sqlfmts;    /* pointer to format nodes */ 16
    sqlda.sqln      = 4;          /* number of columns expected */

EXEC SQL DESCRIBE SelectCmd1 INTO SQLDA; 17
if (sqlca.sqlcode != OK) {
    printf("\n Problem describing SelectCmd1!");
    EXEC SQL SQLEXPLAIN :SQLMessage;
    printf("%s\n",SQLMessage);
    goto a9999;
}

```

Figure 8-12. Program cex10b: Dynamic Queries of Known Format (page 3 of 7)

```

EXEC SQL DECLARE TableList CURSOR for SelectCmd1;                                18
if (sqlca.sqlcode != OK) {
    printf("\n Problem declaring TableList cursor!");
    EXEC SQL SQLEXPLAIN :SQLMessage;
    printf("%s\n",SQLMessage);
    goto a9999;
}

EXEC SQL OPEN TableList;                                                         18
if (sqlca.sqlcode != OK) {
    printf("\n Problem opening TableList cursor!");
    EXEC SQL SQLEXPLAIN :SQLMessage;
    printf("%s\n",SQLMessage);
    goto a9999;
}

/* set up SQLDA fields */
sqlda.sqlnrow = MaxNbrTables;                                                  19
sqlda.sqlbuflen = sizeof(TableList);
sqlda.sqlrowbuf = TableList;

/* Get Table List from SYSTEM.TABLE */
EXEC SQL FETCH TableList USING DESCRIPTOR SQLDA;                               20
if (sqlca.sqlcode == 100) {
    printf("\n No tables qualified!");
    goto a9999;
}
else {
    if (sqlca.sqlcode != OK) {
        printf("\n Problem encountered when reading SYSTEM.TABLE!");
        EXEC SQL SQLEXPLAIN :SQLMessage;
        printf("%s\n",SQLMessage);
        goto a9999;
    }
}

NumOfTables = sqlca.sqlerrd[2];                                               21
EXEC SQL COMMIT WORK;                                                         22

/* do loop for i */

for (i = 0; i < NumOfTables; i ++) {                                         23
    TableList[i].Owner[19] = '\0';
    TableList[i].Table[19] = '\0';
    TableList[i].FileSet[19] = '\0';
}

```

Figure 8-12. Program cex10b: Dynamic Queries of Known Format (page 4 of 7)

```

    sscanf(TableList[i].Owner,"%s",OwnerName);
    sscanf(TableList[i].Table,"%s",TableName);
    sscanf(TableList[i].FileSet,"%s",DBEFileSet);

    printf("\n Generating SQL command to CREATE TABLE ");
    printf("%s.%s",OwnerName,TableName);

sprintf(CmdLine,"SELECT COLNAME,LENGTH,TYPECODE,NULLS,PRECISION,\
SCALE FROM SYSTEM.COLUMN WHERE OWNER = '%s' AND TABLENAME =\
'%s';",OwnerName,TableName);

EXEC SQL PREPARE SelectCmd2 FROM :CmdLine;
if (sqlca.sqlcode != OK) {
    printf("\n Problem PREPARING the SELECT #2 command!");
    EXEC SQL SQLEXPLAIN :SQLMessage;
    printf("%s\n",SQLMessage);
    goto a9999;
}

/* set up SQLDA fields */
sqlda.sqlfmtarr = sqlfmts; /* pointer to format nodes */
sqlda.sqln      = 6;      /* number of columns expected */

EXEC SQL DESCRIBE SelectCmd2 INTO SQLDA;
if (sqlca.sqlcode != OK) {
    printf("\n Problem describing SelectCmd2!");
    EXEC SQL SQLEXPLAIN :SQLMessage;
    printf("%s\n",SQLMessage);
    goto a9999;
}

EXEC SQL DECLARE ColumnList CURSOR for SelectCmd2;
if (sqlca.sqlcode != OK) {
    printf("\n Problem declaring ColumnList Cursor!");
    EXEC SQL SQLEXPLAIN :SQLMessage;
    printf("%s\n",SQLMessage);
    goto a9999;
}

EXEC SQL OPEN ColumnList;
if (sqlca.sqlcode != OK) {
    printf("\n Problem opening ColumnList cursor!");
    EXEC SQL SQLEXPLAIN :SQLMessage;
    printf("%s\n",SQLMessage);
    goto a9999;
}

```

Figure 8-12. Program cex10b: Dynamic Queries of Known Format (page 5 of 7)

```

/* set up SQLDA fields */
sqlda.sqlnrow   = MaxNbrColumns;           31
sqlda.sqlbuflen = sizeof(ColumnList);
sqlda.sqlrowbuf = ColumnList;

/* Get Column List from SYSTEM.COLUMN */

EXEC SQL FETCH ColumnList USING DESCRIPTOR SQLDA;           32
if (sqlca.sqlcode != OK) {
    printf("\n Problem encountered when reading SYSTEM.COLUMN!");
    EXEC SQL SQLEXPLAIN :SQLMessage;
    printf("%s\n",SQLMessage);
    goto a9999;
}

switch (TableList[i].LockMode) {           33
case 1:  sprintf(OneLine,"\nCREATE PUBLICREAD ");
        break;
case 2:  sprintf(OneLine,"\nCREATE PRIVATE ");
        break;
case 3:  sprintf(OneLine,"\nCREATE PUBLIC ");
        break;
default: sprintf(OneLine,"\nUnrecognized Lock Mode ");
        break;
} /* end switch */

sprintf(OneLine+strlen(OneLine),"TABLE %s.%s ",OwnerName,TableName);

write(FileNum,OneLine,strlen(OneLine));           34

for (j = 0; j < sqlca.sqlerrd[2]; j++) {
    ColumnList[j].ColName[19] = '\0';
    strcpy(ColumnName,ColumnList[j].ColName);
    if (j==0) sprintf(OneLine,"\n (%s ",ColumnName);
    else sprintf(OneLine,"\n (%s ",ColumnName);

    switch (ColumnList[j].TypeCode) {           35
        case 0:  if (ColumnList[j].Length == 4) {
                    sprintf(OneLine+strlen(OneLine),"INTEGER           ");
                }
                else
                    sprintf(OneLine+strlen(OneLine),"SMALLINT           ");
                break;
        case 2:
        case 8:  sprintf(OneLine+strlen(OneLine),"CHAR( %2d )",\
                    ColumnList[j].Length);
                break;
    }
}

```

Figure 8-12. Program cex10b: Dynamic Queries of Known Format (page 6 of 7)

```

    case 3:
    case 9:  sprintf(OneLine+strlen(OneLine),"VARCHAR( %2d )",\
                ColumnList[j].Length);
            break;
    case 4:  sprintf(OneLine+strlen(OneLine),"FLOAT          ");
            break;
    case 5:  sprintf(OneLine+strlen(OneLine),"DECIMAL( %2d,%2d )",\
                ColumList[j].Precision,ColumnList[j].Scale);
            break;
    default: sprintf(OneLine+strlen(OneLine)," **** ");
            break;
} /* end switch */

if (ColumnList[j].Nulls == 0) {
    sprintf(OneLine+strlen(OneLine)," NOT NULL");
}

if (j != sqlca.sqlerrd[2]-1) {
    sprintf(OneLine+strlen(OneLine),"");
}
else {
    sprintf(OneLine+strlen(OneLine)," ) IN %s;\n",TableList[i].FileSet
}

write(FileNum,OneLine,strlen(OneLine));

} /* for j = 1 to sqlca.sqlerrd[2] */

EXEC SQL COMMIT WORK;

} /* for i = 1 to NumOfTables */

a9999:

EXEC SQL COMMIT WORK RELEASE;
printf("\n");
}

```

Figure 8-12. Program cex10b: Dynamic Queries of Known Format (page 7 of 7)



## 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

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 in nature. 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 9-1. 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.



**Table 9-2. 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.	

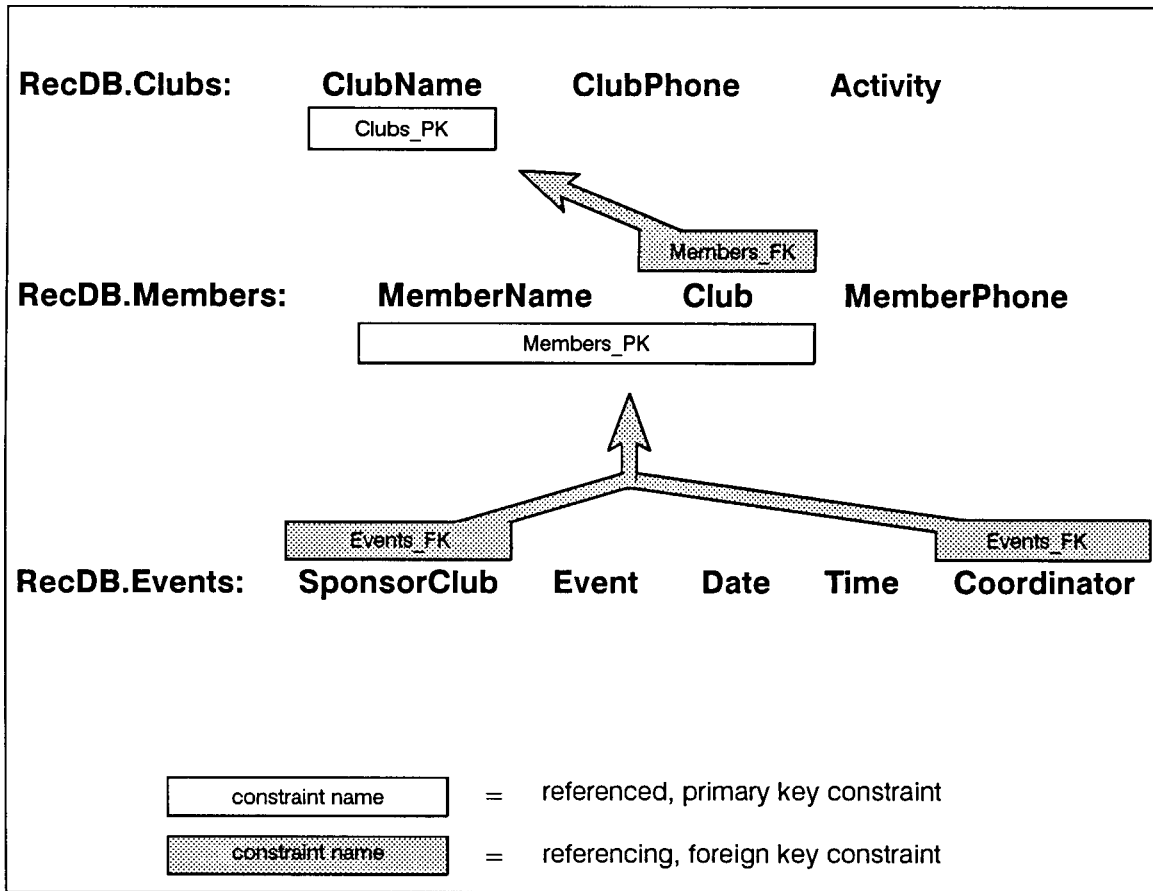
---

## 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 9-1 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 9-1. 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 informational 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.")

**9-4 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 subroutines 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.*



## 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 CREATE TABLE or ALTER TABLE.
- Defining Input and Output with the LONG Column I/O String.
- Putting Data into a LONG Column with [BULK] INSERT.
- Changing a LONG Column with UPDATE [WHERE CURRENT].
- Retrieving LONG Column Data with [BULK] SELECT, FETCH, or REFETCH.
- 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 document, 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 10-1. Commands You Can Use with LONG Columns**

| DDL Operations | DML Operations         |
|----------------|------------------------|
| ALTER TABLE    | [BULK] INSERT          |
| CREATE TABLE   | UPDATE [WHERE CURRENT] |
|                | [BULK] 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 processed by ALLBASE/SQL. 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.

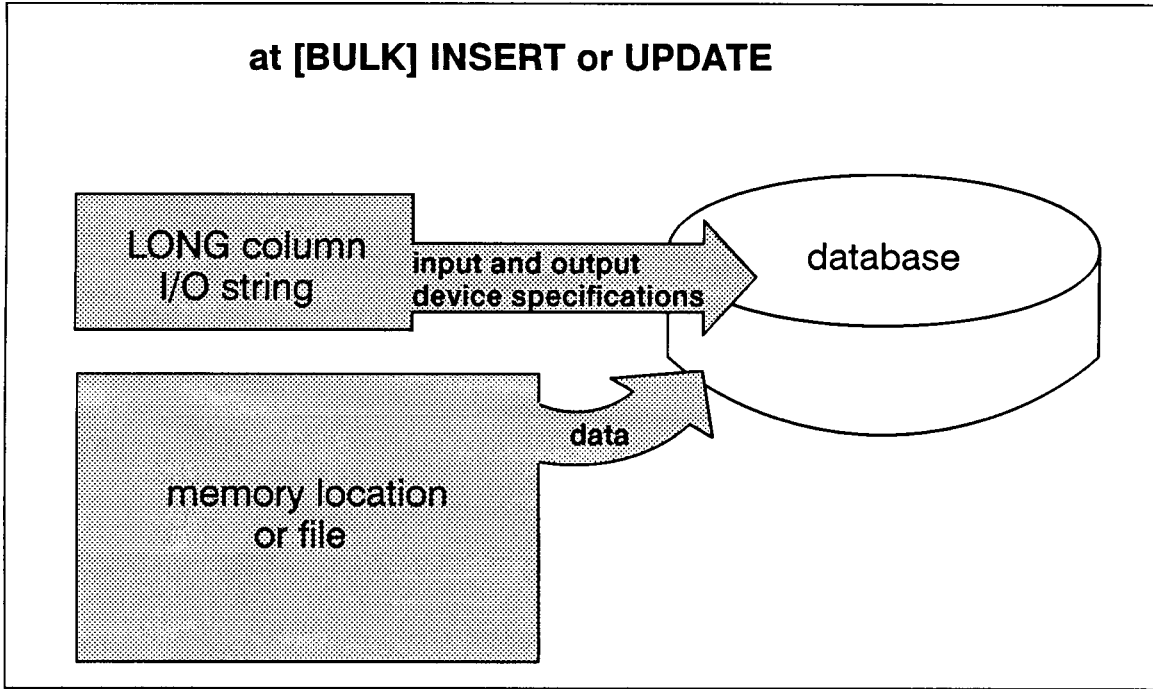
The LONG specification is used with a given ALLBASE/SQL data type when you create the LONG column. Currently, LONG BINARY and LONG VARBINARY are available. Refer to the chapter on “Host Variables” for the details of BINARY and VARBINARY data types.

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.

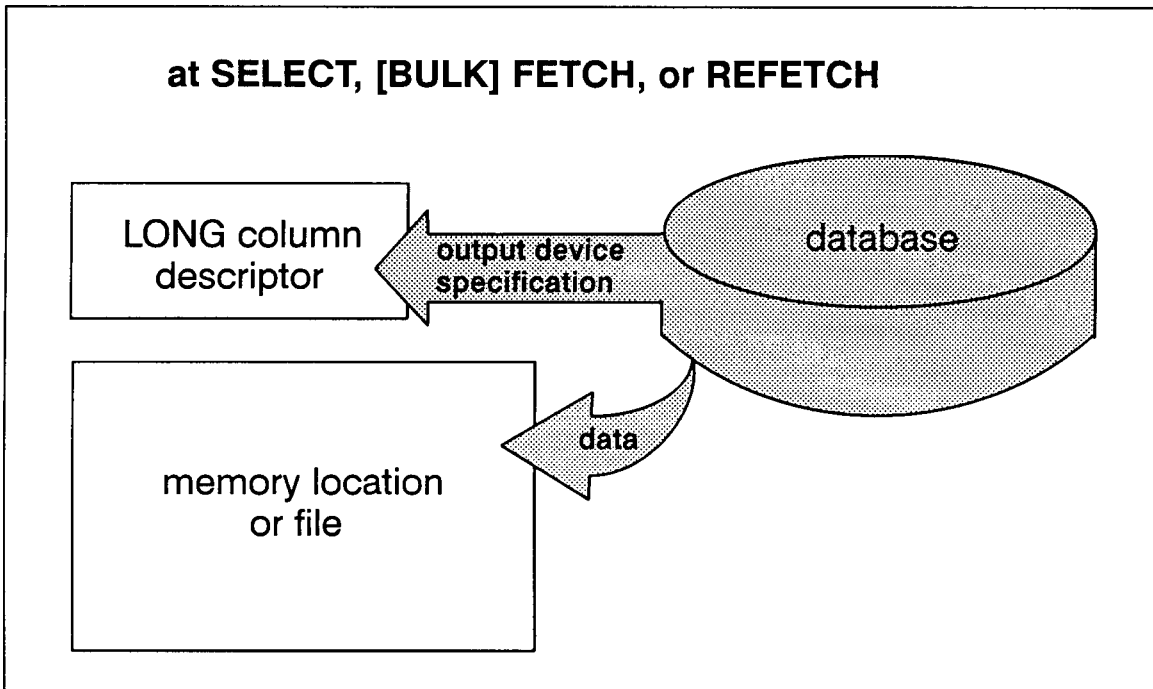
A **LONG column descriptor** (rather than the data itself) is selected or fetched into a host variable. Figure 10-1 and Figure 10-2 illustrate these concepts.





LG200145\_005

Figure 10-1. Flow of LONG Column Data and Related Information to the Database



LG200145\_006

Figure 10-2. 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. The syntax is as follows:

$$(ColumnName \text{ LONG } \left\{ \begin{array}{l} \text{BINARY} \\ \text{VARBINARY} \end{array} \right\} (ByteSize) [ \text{IN } DBEFileSet ] [ \text{NOT NULL} ] [ , \dots ]$$

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’s 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 [BULK] 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 :PartModuleIO, are shown in the last two fields of a hypothetical record in the section, “Example Data File”, that appears later in this chapter. In the above example, the values, bracket and 200, are coded as constants, rather than coming from the data file. The following represents a record could be read into host variables in preparation for the above INSERT statement. The constants, bracket and 200, in the statement could alternately be read into host variables.

```
bracket    200 0'<../tools/bracket >bracket'    0'<../module/mod88 > mod88'    0
```

### Bulk Insert Using Host Variables for LONG Column I/O Strings

The following example illustrates how to define and use an appropriate host variable for a BULK INSERT into PartsTable. Define an entry in your host variable array for each LONG column I/O string.

## Example

```
/* This code segment reads a data file into a host variable array, one line */
/* at a time. It parses the buffer and displays each record as it's         */
/* read and loaded. Then a BULK INSERT to an ALLBASE/SQL database table is  */
/* performed. Maximum number of records per BULK INSERT is 25.             */

#include <stdio.h>

#define MAXSIZE 25

.
.
.

/* PartNum is used to read in the PartNumber as a string.                  */

char PartNum[7];

/* Note that all of the columns allow null values, and an indicator        */
/* variable has been defined for each.                                       */

EXEC SQL BEGIN DECLARE SECTION;
struct {
    char    PartName[11];
    sqlind  PartNameInd;
    int     PartNumber;
    sqlind  PartNumberInd;
    char    PartPictureIO[31]; /* IO string for LONG column PartPicture */
    sqlind  PartPictureInd;
    char    PartModuleIO[31]; /* IO string for LONG column PartModule */
    sqlind  PartModuleInd;
} PartsTableRows[25]; /* INSERT up to 25 rows at a time */
char      SQLMessage[133];
EXEC SQL END DECLARE SECTION;
        /* End Host Variable Declarations */

.
.
.
```

```

int InsertRows()          /* function to insert rows in PartsTable */
{

StartIndex = 1;
NumberOfRows = counter1;

EXEC SQL BULK INSERT INTO PartsTable
        (PartName,
         PartNumber,
         PartPicture,
         PartModule)
        VALUES (:PartsTableRows,
                :StartIndex,
                :NumberOfRows);

if (sqlca.sqlcode != 0) {
    SQLStatusCheck();
}

} /* End of InsertRows Function */

/* Here you could accept data from the user or from a file.      */
/* For this example, a file is used.                             */

int main()               /* function to initialize host variable array from */
                        /* a file */
{

    char s[225];
    int i = 0;

FILE *ptr;
ptr = fopen("data_file","r");

    while (fgets(s,225,ptr) && (i < MAXSIZE)) {
        sscanf (s, "%10c %2d%2c %2d%30c %2d%30
                PartsTableRows[i].PartName, &PartsTableRows[i].PartNameInd,
                PartNum, &PartsTableRows[i].PartNumberInd,
                PartsTableRows[i].PartPictureI0,
                &PartsTableRows[i].PartPictureInd,
                PartsTableRows[i].PartModuleI0, &PartsTableRows[i].PartModuleInd);

```

```

/* We read PartNumber as a string, but assign it as an integer.          */
/* atoi converts the ascii to integer.                                    */
   PartsTableRows[i].PartNumber = atoi(PartNum);

   printf ("%10s %2d %2d %2d %30s %2d %30s %2d\n",
           PartsTableRows[i].PartName, PartsTableRows[i].PartNameInd,
           PartsTableRows[i].PartNumber, PartsTableRows[i].PartNumberInd,
           PartsTableRows[i].PartPictureI0,
           PartsTableRows[i].PartPictureInd,
           PartsTableRows[i].PartModuleI0, PartsTableRows[i].PartModuleInd);
   i++;
}
} /* end of initialize function */

```

.  
.  
.

**Example Data File**

The file, in this case named data\_file, that your program reads might look something like this. Note that it is limited to 80 characters per record to facilitate documentation.

```

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 [BULK] 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. The syntax is as follows:

$$\begin{array}{l} \text{[BULK]SELECT [ALL] } \left\{ \begin{array}{l} * \\ \text{[ Owner. ] Table.*} \\ \text{CorrelationName.*} \\ \text{CorrelationName.ColumnName} \end{array} \right\} \text{[ , ... ]} \\ \text{[ INTO HostVariableDeclaration ] FROM } \{ \text{[ Owner. ] FromTableName [ CorrelationName ] } \\ \text{[ , ... ]} \end{array}$$

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 10-2 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 10-2. 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<br>1 = overwrite<br>2 = append<br>3 = wildcard<br>4 = overwrite and wildcard<br>5 = append and wildcard | 45            |
| Output Device Type                | 0 = no device specified<br>1 = file<br>2 = shared memory                                                                        | 46            |
| Input Device Type                 | 0 = no device specified<br>1 = file<br>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 |

**Example LONG Column Descriptor Declaration**

```

/* Use this when you don't need to break down the descriptor. */

EXEC SQL BEGIN DECLARE SECTION;
char longdescriptor[96];
:
EXEC SQL END DECLARE SECTION;

/* If you want to access a portion of the descriptor, copy the */
/* longdescriptor host variable to the structure below. */

struct {
    char outputdevlocation[44];
    char outputdevop;
    char outputdevtype;
    char inputdevtype;
    char unused_A;
    int bytelength;
    char unused_B[44];
} longstruct

```

## Using LONG Columns with a BULK 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 in the array. 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 sqltallen 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 another 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. The actual shared memory address will be placed in the LONG column descriptor. Refer to this address when using SELECT and UPDATE commands.

## Programming with ALLBASE/SQL 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.”

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.

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 “Expressions” and “Data Types” chapters.

---

**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 11-1. Where to Use Date/Time Functions**

| DML Operation                     | Clause                            |
|-----------------------------------|-----------------------------------|
| [BULK]INSERT <sup>1</sup>         | VALUES<br>WHERE                   |
| UPDATE or<br>UPDATE WHERE CURRENT | SET<br>WHERE                      |
| DELETE or<br>DELETE WHERE CURRENT | WHERE                             |
| [BULK]SELECT                      | Select list <sup>2</sup><br>WHERE |
| DECLARE                           | Select list <sup>2</sup><br>WHERE |

<sup>1</sup> In the case of a [BULK] INSERT, 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.

<sup>2</sup> 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 about defining a host variable to be compatible with a specific ALLBASE/SQL data type. 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 11-2 shows host variable data type compatibility for date/time functions.

**Table 11-2. Host Variable Data Type Compatibility for Date/Time Functions**

| Date/Time Function                                    | Input Format Specification | Output Format Specification | Input Data | Output Data            |
|-------------------------------------------------------|----------------------------|-----------------------------|------------|------------------------|
| TO_DATE<br>TO_TIME<br>TO_DATETIME<br>TO_INTERVAL      | (VAR)CHAR                  |                             | (VAR)CHAR  | (VAR)CHAR <sup>1</sup> |
| TO_CHAR                                               |                            | (VAR)CHAR                   |            | (VAR)CHAR              |
| TO_INTEGER                                            |                            | (VAR)CHAR                   |            | INTEGER                |
| ADD_MONTHS                                            | INTEGER                    |                             |            | (VAR)CHAR <sup>1</sup> |
| <sup>1</sup> Applies only when used in a select list. |                            |                             |            |                        |

## Using Date/Time Input Functions

The new 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 11-1. 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 11-3.

**Table 11-3. 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 <sup>1</sup>        | 8                                                  |
| INTERVAL            | DDDDDD HH:MM:SS              | 16                                                 |

<sup>1</sup> 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 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 11-2 with one exception. In the case of a [BULK] 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:

$$\left\{ \begin{array}{l} \text{TO\_CHAR} ( \text{ColumnName} [ , \text{FormatSpecification} ] ) \\ \text{TO\_INTEGER} ( \text{ColumnName}, \text{FormatSpecification} ) \end{array} \right\}$$

### 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.*

*In the ReportBuffer array, 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) in the ReportBuffer array.*

```
END DECLARE SECTION
```

```
.  
. .  
. .
```

```
DECLARE ReportInfo CURSOR FOR
```

```
        SELECT BatchStamp,  
               TO_INTEGER(TestDate, :QuarterlyFormat),  
               LabTime,  
               PassQty,  
               TestQty  
        FROM ManufDB.TestData
```

```
.  
. .  
. .
```

```
BULK FETCH ReportInfo  
        INTO ReportBuffer
```

---

## 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; refer back to Table 11-2.

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 earlier in this chapter in Table 11-1, 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 example program shown in Figure 11-1 is based on the manufacturing database and the purchasing database that are a part of the sample database environment, PartsDBE. (Reference the *ALLBASE/SQL Reference Manual*, appendix C.)

The program shows how to convert a column data type from CHAR to DATE. Informative comments and explanations are present throughout the listing.



```

struct  {
    int      NewOrderNumber;
    int      NewVendorNumber;
    sqlind   NewVendorNumInd;
    char     NewOrderDate[11]; /*Add a byte for end of char array.*/
    sqlind   NewOrderDateInd;
} NewOrders[25];
short    StartIndex2;
short    NumberOfRows2;

char     SQLMessage[133];      /*Add a byte for end of char array.*/

EXEC SQL END DECLARE SECTION;
      /* End Host Variable Declarations */

/*****
/*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.
*****/

boolean DeclareCursor(){

    EXEC SQL DECLARE OrdersCursor
            CURSOR FOR
            SELECT *
            FROM PurchDB.Orders;
}

/*****
/*Function to rollback the transaction.
*****/

int RollBackWork(){

printf("Rollback Work\n");
EXEC SQL ROLLBACK WORK;
    if (sqlca.sqlcode != 0K){
        SQLStatusCheck();
        TerminateProgram();
    }
}

} /* End RollBackWork Function */

```

**Figure 11-1. Program cex9a: Using Date/Time Functions (2 of 7)**



```

/*****
/*                               Beginning of program.                               */
*****/
main() {

printf("Program to convert date from CHAR to DATE data type.\n");
printf("Event List:\n");
printf("  Connect to PartsDBE\n");
printf("  BULK FETCH all rows from Orders Table.\n");
printf("  Convert the date.\n");
printf("  BULK INSERT all fetched rows into NewOrders Table \n");
printf("  with converted date.\n");
printf("  Release PartsDBE\n\n");

if (ConnectDBE()) {

    DoneConvert = FALSE;
    OrdersOK = TRUE;

    BeginTransaction();

    EXEC SQL OPEN OrdersCursor KEEP CURSOR WITH LOCKS;

    if (sqlca.sqlcode != OK) {
        SQLStatusCheck();
        RollBackWork();
        OrdersOK = FALSE;
        DoneConvert = TRUE;
    }

    do {
        FetchOld();
    } while (! 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)                /* If there were no errors in processing, data */
    CommitWork();            /* is committed to the database.      */

TerminateProgram();

} /* END if */

} /* End of Main Program */

```

**Figure 11-1. Program cex9a: Using Date/Time Functions (3 of 7)**

```

/*****
/*
Function to release PartsDBE.
*/
*****/

int TerminateProgram() /* Function to Release PartsDBE */
{

EXEC SQL RELEASE;

} /* End TerminateProgram Function */

/*****
/*Function to display error messages and terminate the program when the */
/*transaction has been rolled back by ALLBASE/SQL.
*/
*****/

int SQLStatusCheck() /* Function to Display Error Messages */
{

Abort = FALSE;

if (sqlca.sqlcode <= DeadLock) Abort = TRUE;
if (sqlca.sqlcode = NoMemory) Abort = TRUE;

do {
EXEC SQL SQLEXPLAIN :SQLMessage;
printf(SQLMessage);
} while (sqlca.sqlcode != 0);
if (Abort) TerminateProgram();

} /* End SQLStatusCheck Function */

/*****
/*Function to connect to the sample database environment, PartsDBE.
*/
*****/

boolean ConnectDBE(){
boolean rv; /* return value */

printf("Connect to PartsDBE\n");

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

rv = TRUE;
if (sqlca.sqlcode != OK){
rv = FALSE;
}
}

```

Figure 11-1. Program cex9a: Using Date/Time Functions (4 of 7)

```

        SQLStatusCheck();

    } /* End if */
return(rv);
} /* End of ConnectDBE Function */
/*****
/*Function to begin the transaction with cursor stability specified.  */
*****/

int BeginTransaction(){
EXEC SQL BEGIN WORK CS;

    if (sqlca.sqlcode != OK){
        SQLStatusCheck();
        TerminateProgram();
    }

} /* End BeginTransaction Function */

/*****
/*Function to commit work to the database OR save the cursor position.  */
*****/

int CommitWork(){
printf("Commit Work\n");
EXEC SQL COMMIT WORK;
    if (sqlca.sqlcode != OK){
        SQLStatusCheck();
        TerminateProgram();
    }

} /* End CommitWork Function */

/*****
/*Function to BULK INSERT into PurchDB.NewOrders table.  */
*****/

int InsertNew(){
NumberOfRows2 = counter1;
StartIndex2   = 0;

printf("BULK INSERT INTO  PurchDB.NewOrders\n");

EXEC SQL BULK INSERT INTO  PurchDB.NewOrders
        VALUES (:NewOrders,
                :StartIndex2,
                :NumberOfRows2);

```

Figure 11-1. Program cex9a: Using Date/Time Functions (5 of 7)

```

switch (sqlca.sqlcode){
    case    OK:        break;

        default:      SQLStatusCheck();
                      RollBackWork();
                      OrdersOK = FALSE;
                      DoneConvert = TRUE;
    }    /* switch */
}

/* End of Function InsertNew */

/*****
/*Function to convert OrderDate from CHAR to DATE data type and transfer */
/*data to an array in preparation for BULK INSERT into a new table.    */
*****/

int TransferData()
{
    int i,j;

    NumberOfRows = counter1;

    for (i = 0; i <= NumberOfRows; i++){
        NewOrders[i].NewOrderNumber = Orders[i].OrderNumber;
        NewOrders[i].NewVendorNumber = Orders[i].VendorNumber;
    }

        /* Convert Date */

    for (i = 0; i <= NumberOfRows; i++){
        for (j = 0; j < 4; j++){
            NewOrders[i].NewOrderDate[j] = Orders[i].OrderDate[j];
        }
        NewOrders[i].NewOrderDate[4] = '-';
        for (j = 5; j < 7; j++){
            NewOrders[i].NewOrderDate[j] = Orders[i].OrderDate[j-1];
            NewOrders[i].NewOrderDate[7] = '-';
        }
        for (j = 8; j < 10; j++){
            NewOrders[i].NewOrderDate[j] = Orders[i].OrderDate[j-2];
        }
    }

} /* End of Function TransferData */

```

Figure 11-1. Program cex9a: Using Date/Time Functions (6 of 7)

```

/*****
/*Function to BULK FETCH Orders table data 25 rows at a time into an array*/
*****/

int FetchOld()
{
    NumberOfRows = 25;
    StartIndex = 0;

    printf("BULK FETCH PurchDB.Orders\n");

    EXEC SQL BULK FETCH OrdersCursor
        INTO :Orders, :StartIndex, :NumberOfRows;

    counter1 = sqlca.sqlerrd[2];    /* Set counter1 to number of rows fetched.*/

    switch (sqlca.sqlcode){
        case OK:    CommitWork();          /* SAVE THE CURSOR POSITION */
                    break;                /* Used in conjunction with */
                                           /* cursor stability.      */

        case NotFound:    CommitWork();
                    printf("\nThere are no Orders Table rows to FETCH.\n");
                    DoneConvert = TRUE;
                    break;

        default:    SQLStatusCheck();
                    RollBackWork();
                    OrdersOK = FALSE;
                    DoneConvert = TRUE;
    }    /* switch */

    if (! DoneConvert)
        TransferData();

    if (! DoneConvert)
        InsertNew();

}    /* End of Function FetchOld */

```

Figure 11-1. Program cex9a: Using Date/Time Functions (7 of 7)

---

## 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 atomically.

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 variable array, VendorsArray.

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)
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, or a constant. 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:

```
sqltid      tidvarname;
```

### 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 11-4. 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.
- Insuring 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:

```
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 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, 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. ISQL displays, “Number of rows selected is 0” for a SELECT statement and “Number of rows processed is 0” for an UPDATE or DELETE statement.

Status checking is discussed in detail in the ALLBASE/SQL application programming guides. Refer to the guide for the language you are using.

## 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 11-2 uses the RC isolation level with a BULK SELECT statement to obtain data and the 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 11-2. Using RC and RR Transactions with BULK SELECT, SELECT, and UPDATE**

*When all user changes have been entered, use a loop to compare the previously fetched rows (in OrdersArray) with the same rows as they now exist in the database.*

*Begin your transaction with the RR isolation level. No other transaction can access the locked data until this transaction ends, providing maximum data integrity.*

```
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 11-2. Using RC and RR Transactions with BULK SELECT, SELECT, and UPDATE (2 of 2)**

## Reducing Commit Overhead for Multiple Updates with TID Access

Figure 11-3 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 an RC isolation 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 a *RC* isolation level. This ensures maximum concurrency while assuring that only committed 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 *NewPartsArray*. Include the appropriate *TID* value with each *NewPartsArray* entry.

For each row in the *NewPartsArray*

```
    VerifyAndUpdate (8)
```

End For

Figure 11-3. 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 when 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 11-3. Using TID Access to Reduce Commit Overhead (2 of 2)**



# Index

---

## 3

- 300/400 differences
  - compiler command, 2-10
  - debugging, 1-26
  - link command, 2-11
  - preprocessor banner, 2-16
  - shared memory, 1-24

## A

- absolute pathname, 1-22
- active set
  - re-access, 6-8
- ADD\_MONTHS function
  - example with BULK SELECT, 11-9
  - syntax, 11-9
- aggregate function
  - simple data manipulation, 5-2
- ALTER TABLE command
  - syntax for LONG columns, 10-4
- ANSI SQL1 level 2
  - specifying a default value, 3-17
- ANSI SQL86 level 2
  - floating point data, 3-10
- ANSI standards
  - sqlcode, 4-4
- arrays
  - BULK SELECT, 3-5
  - character data, 3-8
  - declarations of, 3-25
  - in sqlda declaration, 8-15
  - referencing, 7-2
- atomic operation
  - defined, 4-2
- authorization
  - changing, 1-21
  - dynamic preprocessing, 8-2
  - granting, 1-18
  - program development, 1-15
  - program maintenance, 1-20
- automatic rollback, 4-12

## B

- basic SQL statements, 1-7
- BEGIN DECLARE SECTION
  - declaring host variables, 3-6

- delimiting the declare section, 2-12
- BEGIN WORK
  - in transaction management, 5-7
  - issued by preprocessor, 2-3
- binary data
  - compatibility, 3-11
  - host variable definition, 3-11
  - using the LONG phrase with, 3-11
- BULK FETCH
  - basic uses of, 7-8
- BULK FETCH command
  - used in example program, 11-11
  - with TO\_INTEGER function, 11-8
- BULK INSERT
  - basic uses of, 7-10
- BULK INSERT command
  - used in example program, 11-11
  - used with LONG columns, 10-6
  - using host variables for LONG column I/O strings, 10-6
  - with LONG columns:example data file, 10-9
  - with LONG columns:example program, 10-7
- BULK option
  - not used for dynamic FETCH, 8-10
- bulk processing
  - INTO clause, 3-5
- bulk processing variables, 3-5
- BULK SELECT
  - basic uses, 7-4
- [BULK] SELECT command
  - used with LONG columns, 10-10
- BULK SELECT command
  - used with LONG columns, 10-12
  - with ADD\_MONTHS function, 11-9
- BULK table processing
  - BULK FETCH, 7-8
  - BULK INSERT, 7-10
  - BULK SELECT, 7-4
  - commands, 7-4
  - sample program, 7-13
  - techniques, 7-1

## C

- case shifting
  - installable module name, 2-23
- cc -c

- compiler command, 2-8
- cex10a, 8-23
- cex10b, 8-41
- cex2, 2-25
  - source code, 1-31
- cex2.c
  - preprocessor modified source code, 2-25
- cex5, 4-18
- cex7, 5-18
- cex8, 6-25
- cex9, 7-20
- CHAR data declaration, 3-8
- CHECKPOINT
  - before preprocessing, 2-3
- CLOSE
  - after BULK FETCH, 7-8
  - before ending a transaction, 6-8, 6-9
  - freeing buffer space with, 6-8
  - to re-access the active set, 6-8
  - with COMMIT WORK, 6-11
  - with KEEP CURSOR, 6-11
- coding considerations
  - for date/time functions, 11-9
  - for LONG columns, 10-13, 10-14
- column specifications for floating point data, 3-10
- comments in SQL commands, 1-10
- COMMIT WORK
  - and revalidation of sections, 2-22
  - in transaction management, 5-7
  - issued by preprocessor, 2-3, 2-8
  - with CLOSE, 6-11
  - with KEEP CURSOR, 6-11
- compatibility of variables, 3-19
- compiler
  - and linker, 2-8
  - compilable section, 2-8
- compiler command
  - 300/400 differences, 2-10
- compiler options
  - current language, 1-28
  - lcl, 2-8
  - lportnls, 2-8
  - lsql, 2-8
- concurrency, 5-7
- CONNECT
  - in application programs, 1-18
  - issued by preprocessor, 2-3
  - to start a DBE session, 1-15
- CONNECT authority
  - and preprocessing, 1-15
- constant
  - as default data value, 3-17
- constraint test matrix for integrity constraints, 9-3

- conversion
  - numeric, 3-24
  - string data, 3-23
- CREATE TABLE command
  - syntax for LONG columns, 10-4
- CURRENT\_DATE function result
  - used as default data value, 3-17
- CURRENT\_DATETIME function result
  - used as default data value, 3-17
- current language, 1-28
- current row
  - DELETE WHERE CURRENT, 6-7
- CURRENT\_TIME function result
  - used as default data value, 3-17
- cursor
  - and BULK FETCH, 7-8
  - and dynamic queries, 8-17
  - and sections, 2-20
- cursor processing
  - CLOSE, 6-8
  - commands, 6-1
  - DECLARE CURSOR, 6-2
  - definition, 6-1
  - DELETE WHERE CURRENT, 6-7
  - FETCH, 6-3
  - OPEN, 6-3
  - sample program, 6-25
  - techniques, 6-1
  - transaction management, 6-9
  - UPDATE and FETCH, 6-6
  - UPDATE WHERE CURRENT, 6-5

## D

- DataBase Environment Configuration (DBECon), 1-22
- data buffer
  - declaration, 3-26
  - layout, 8-15
  - null indicator suffix, 8-15
  - parsing, 8-19
  - rows to retrieve, 8-15
  - varchar prefix, 8-15
- data compatibility
  - binary, 3-11
  - floating point, 3-10
  - for date/time function parameters, 11-2, 11-3
  - for default data values, 3-18
  - LONG binary, 3-11
  - LONG varbinary, 3-11
- data consistency, 4-2
  - in sample database, 4-2
- data input using date/time functions, 11-3
- data integrity
  - changes to error checking, 9-1
  - introduction to, 9-1



- number of rows processed , 9-1
- row level versus statement level, 9-1
- using sqlerrd[2], 9-1
- data retrieval using date/time functions, 11-7
- data structures
  - for dynamic query, 8-8
- data type
  - compatibility, 3-19
  - conversion, 3-23, 3-24
  - declarations, 3-8
  - equivalency, 3-19
- data types
  - floating point, 3-10
  - used with LONG columns, 10-2
- date/time ADD\_MONTHS function
  - overview, 11-9
  - where to use, 11-9
- date/time data conversion
  - example program, 11-11
- date/time functions
  - coding considerations, 11-9
  - data compatibility, 11-2, 11-3
  - example programs, 11-10
  - examples using ManufDB database, 11-4, 11-7, 11-9
  - example using default format specifications, 11-5
  - how used, 11-2
  - introduction to, 11-1
  - leading zeros required for input functions, 11-9
  - parameters for, 11-2
  - unspecified format elements default filled, 11-9
  - used to add a number of months, 11-9
  - used when inputting data, 11-3
  - used when retrieving data, 11-7
  - using host variables for format specifications, 11-2
  - using host variables for input and output data, 11-2
  - using host variables with, 11-2
  - where to use ADD\_MONTHS, 11-9
  - where to use input functions, 11-3
  - where to use output functions, 11-7
  - where to use TO\_CHAR, 11-7
  - where to use TO\_DATE, 11-3
  - where to use TO\_DATETIME, 11-3
  - where to use TO\_INTEGER, 11-7
  - where to use TO\_INTERVAL, 11-3
  - where to use TO\_TIME, 11-3
  - where used, 11-2
- date/time input functions
  - examples, 11-4
  - not intended for use in select list, 11-3
  - overview, 11-3
  - where to use, 11-3
- date/time output functions
  - examples, 11-7, 11-8
  - overview, 11-7
  - where to use, 11-7, 11-9
- DBA authority
  - and preprocessing, 1-15
  - and START DBE, 1-15
- DBECon file, 1-22
- DBEnvironment
  - user mode, 2-3
- DBE session
  - in preprocessing, 2-3
- DCL defined, 1-2
- DDL defined, 1-2
- DDL operations
  - used with integrity constraints, 9-2
  - used with LONG columns, 10-1
- deadlock
  - and error recovery, 4-2
  - status checking, 4-30
- debugging, 1-26
  - setting flags, 1-26
  - signals, 1-26
- DECIMAL data declaration, 3-16
- decimal type compatibility, 3-24
- declaration of data
  - CHAR, 3-8
  - DATE, 3-16
  - DATETIME, 3-16
  - DECIMAL, 3-16
  - FLOAT, 3-10
  - INTEGER, 3-10
  - INTERVAL, 3-16
  - SMALLINT, 3-10
  - TIME, 3-16
  - VARCHAR, 3-8
- declaration part
  - host variable, 3-6
- DECLARE CURSOR
  - FOR UPDATE OF, 6-2
  - preprocessor directive, 6-3
  - SELECT, 6-2
  - syntax, 6-2
  - used with BULK FETCH, 7-8
- declare section
  - BEGIN and END DECLARE SECTION
    - commands, 2-12
  - defined, 3-6
- default data values
  - constant, 3-17
  - data compatibility, 3-18
  - for columns allowing nulls, 3-17
  - in addition to null, 3-17

- not used with LONG BINARY data, 3-18
- not used with LONG columns, 3-18
- not used with LONG VARBINARY data, 3-18
- NULL, 3-17
- result of CURRENT\_DATE function, 3-17
- result of CURRENT\_DATETIME function, 3-17
- result of CURRENT\_TIME function, 3-17
- USER, 3-17
- default format specification example
  - date/time functions, 11-5
- defining integrity constraints, 9-2
- defining LONG columns
  - in a table, 10-4
  - input and output specification, 10-5
  - with the LONG column I/O string, 10-5
- definitions
  - fast ipc, 1-25
  - input device specification, 10-5
  - LONG column I/O string, 10-5
  - output device specification, 10-5
  - row level integrity, 9-1
  - signal ipc, 1-25
- DELETE, 5-7
- DELETE command
  - used with LONG columns, 10-13
  - with TO\_DATETIME function, 11-6
- DELETE WHERE CURRENT
  - current row, 6-7
  - restrictions, 6-7
  - syntax, 6-7
- DELETE WHERE CURRENT command
  - used with LONG columns, 10-13
- delimiting SQL commands, 1-9
- DESCRIBE
  - dynamic non-query, 8-21
  - dynamic query, 8-21
- designing an application using statement level integrity, 9-3
- detecting end of scan, 4-35
- DML defined, 1-2
- DML operations
  - used with date/time functions, 11-2
  - used with integrity constraints, 9-2
  - used with LONG columns, 10-1
- d option
  - full preprocessing mode, 2-1
- DROP MODULE, 1-20, 1-21
- dynamically deleting data
  - DELETE WHERE CURRENT command
    - cannot be prepared, 8-10
    - error checking strategy, 8-10
- dynamically updating data
  - error checking strategy, 8-10

- UPDATE WHERE CURRENT command
  - cannot be prepared, 8-10
  - using SELECT command with FOR UPDATE OF clause, 8-10
- dynamic command, 8-1
  - passing to ALLBASE/SQL, 8-5
  - query with known query result format, 8-41
  - query with unknown query result format, 8-23
- dynamic FETCH
  - BULK option not used, 8-10
- dynamic FETCH command
  - used with LONG columns, 10-12
- dynamic operations
  - dynamic commands, 8-1
  - handling non-queries, 8-6
  - queries vs. non-queries, 8-21
  - sample programs, 8-23, 8-41
  - techniques, 8-1
- dynamic preprocessing, 8-1
  - authorization for, 8-2
- dynamic query data structures, 8-8

## E

- END DECLARE SECTION
  - declaring host variables, 3-6
  - delimiting the declare section, 2-12
- environment variables
  - HPSQLfast\_ipc, 1-26
- error checking
  - changes for this release, 9-1
  - using sqlerrd[2], 9-1
  - when dynamically deleting data, 8-10
  - when dynamically updating data, 8-10
  - with row level integrity, 9-1
  - with statement level integrity, 9-1
- example
  - BULK FETCH command with TO\_INTEGER function, 11-8
  - BULK SELECT command with ADD\_MONTHS function, 11-9
  - DELETE command with TO\_DATETIME function, 11-6
  - INSERT command with TO\_DATE function, 11-4
  - INSERT command with TO\_DATETIME function, 11-4
  - INSERT command with TO\_INTERVAL function, 11-4
  - INSERT command with TO\_TIME function, 11-4
  - SELECT command with TO\_CHAR function, 11-7
  - SELECT command with TO\_DATETIME function, 11-6, 11-7

- SELECT command with TO\_INTERVAL
  - function, 11-6
- UPDATE command with TO\_DATE function, 11-5
- UPDATE command with TO\_DATETIME
  - function, 11-5
- example application design
  - using integrity constraints, 9-3
- example data file
  - BULK INSERT command with LONG
    - columns, 10-9
- example program
  - BULK INSERT command with LONG
    - columns, 10-7
  - date/time data conversion, 11-11
  - LONG column descriptor declaration, 10-11
- examples of date/time input functions, 11-4
- examples of date/time output functions, 11-7, 11-8
- executable program
  - creating, 2-8
- EXECUTE
  - non-dynamic queries, 8-8
- executing programs, 1-19
- explicit status checking
  - defined, 4-1
  - introduction, 4-13
  - uses of, 4-24
- externals include file
  - sample, 2-33
- F**
- fast ipc
  - definition, 1-25
  - how to set, 1-25
  - when to use, 1-25
- FETCH, 6-3
  - current row, 6-3
- FETCH command
  - used dynamically with LONG columns, 10-12
  - used with LONG columns, 10-10
- file
  - DataBase Environment Configuration, 1-22
  - DBECon, 1-22
- file IO
  - KEEP CURSOR, 6-15
- FLOAT data declaration, 3-10
- floating point data
  - 4-byte, 3-10
  - 8-byte, 3-10
  - column specifications, 3-10
  - compatibility, 3-10
  - REAL keyword, 3-10
- format array
  - declaration, 3-26
  - fields, 8-13
  - mandatory declaration for dynamic query, 8-17
- FOR UPDATE OF
  - UPDATE WHERE CURRENT, 6-2, 6-5
- full preprocessing mode
  - preprocessor command, 2-1
- G**
- general rules
  - skeleton program, 1-8
- GOTO vs. GO TO, 4-14
- GRANT
  - for required authorities, 1-18
- H**
- host variable
  - bulk processing, 3-5
  - declaration, 3-6
  - declaration part, 3-6
  - declaration summary, 3-10
  - declaring for ALLBASE/SQL messages, 3-28
  - declaring for DBEnvironment names, 3-29
  - declaring for savepoints, 3-28
  - indicator, 3-3
  - initialization, 3-3
  - input, 3-3
  - names, 3-2
  - output, 3-3
  - purpose, 3-1
  - scope, 3-6
  - uses, 3-1
- host variables
  - used for binary data, 3-11
  - used for LONG column I/O strings, 10-6
  - used with date/time functions, 11-2
- host variable scope
  - at preprocessing time, 3-6
  - at run time, 3-6
- HPSQLfast\_ipc environment variable
  - to set interprocess communication, 1-26
- I**
- implicit status checking
  - defined, 4-1
  - usage, 4-13
- include files
  - as input files, 2-15
  - contents, 2-31
  - created by preprocessor, 1-11
  - samples, 2-31
  - user specified, 2-12
- indicator variables, 3-3, 3-27
  - location of, 3-3

- null, 3-3
  - null values, 6-3
  - truncation, 3-3
- INFO command
  - and null indicator variables, 3-20
  - and null values, 3-20
  - and type compatibility, 3-20
- input device specification
  - definition, 10-5
- INSERT
  - and simple data manipulation, 5-5
- INSERT command
  - used with LONG columns, 10-6
  - using host variables for LONG column I/O strings, 10-6
  - with TO\_DATE function, 11-4
  - with TO\_DATE/TIME function, 11-4
  - with TO\_INTERVAL function, 11-4
  - with TO\_TIME function, 11-4
- INSTALL, 1-20, 2-10
- installable module file, 2-23
- INTEGER data declaration, 3-10
- integrity constraint definition, 9-2
- integrity constraints
  - and statement level integrity, 9-3
  - commands used with, 9-2
  - constraint test matrix, 9-3
  - designing an application, 9-3
  - example application using RecDB database, 9-3
  - in RecDB database, 9-3
  - introduction to, 9-1
  - restrictions, 9-2
  - unique and referential, 9-2
- interprocess communication options
  - fast ipc, 1-25
  - how to set, 1-25
  - HPSQLfast\_ipc environment variable, 1-26
  - introduction to, 1-25
  - signal ipc, 1-25
  - when to use, 1-25
- i option
  - preprocessor syntax checking mode, 2-6

## K

- KEEP CURSOR
  - file IO, 6-15
  - terminal IO, 6-15
- KEEP CURSOR WITH NOLOCKS command
  - use with OPEN command , 6-3

## L

- LANG DBEnvironment option
  - current language, 1-28
- LANG environment variable
  - setting and resetting, 1-28
- language
  - current language, 1-28
  - native language support, 1-27
  - setting and resetting, 1-28
- lcl
  - pascal library, 2-8
- link command
  - 300/400 differences, 2-11
  - options, 2-11
- linker
  - and compiler, 2-8
  - to create executable program, 2-8
- link options
  - lcl, 2-8
  - lportnls, 2-8
  - lsql, 2-8
- log file space
  - use during preprocessing, 2-3
- LONG binary data
  - compatibility, 3-11
  - definition, 3-11
  - how stored, 3-11
- LONG binary versus LONG varbinary data
  - usage, 3-11
- LONG column definition
  - in a table, 10-4
  - input and output specification , 10-5
  - with the LONG column I/O string, 10-5
- LONG column descriptor
  - contents of, 10-10
  - example declaration, 10-11
  - general concept, 10-2
  - how used, 10-10
  - introduction to, 10-5
- LONG column I/O string
  - general concept, 10-2
  - how used , 10-5
  - input device specification, 10-5
  - output device specification, 10-5
  - shared memory input and output, 10-6
  - used with [BULK] INSERT command, 10-6
  - used with host variable, 10-6
  - used with INSERT command, 10-6
- LONG columns
  - changing data, 10-12
  - coding considerations, 10-13
  - commands used with, 10-1
  - considering multiple users, 10-14
  - data types used with, 10-2

- deciding on space allocation, 10-14
- deleting data, 10-13
- file usage from an application, 10-5
- general concepts, 10-2
- input options, 10-5
- introduction to, 10-1
- maximum per table definition, 10-4
- output options, 10-5
- performance, 10-4
- putting data in, 10-6
- restrictions, 10-4
- retrieving data from, 10-10
- size maximum, 10-2
- specifying a DBEFileSet, 10-4
- storage, 10-4
- storing and retrieving data, 10-2
- used with [BULK] INSERT command, 10-6
- used with [BULK] SELECT command, 10-10
- used with DELETE [WHERE CURRENT] command, 10-13
- used with dynamic FETCH command, 10-12
- used with FETCH or REFETCH commands, 10-10
- used with UPDATE [WHERE CURRENT] command, 10-12
- using file naming conventions, 10-13
- using file versus shared memory, 10-13
- using the LONG column descriptor, 10-10
- LONG phrase
  - used with binary data, 3-11
  - used with varbinary data, 3-11
- LONG varbinary data
  - compatibility, 3-11
  - definition, 3-11
  - how stored, 3-11
- lportnls
  - native language library, 2-8
- lsql
  - sql library, 2-8
- lsql option, 2-10

## M

- maintaining ALLBASE/SQL programs, 1-19
- ManufDB database
  - examples using date/time functions, 11-4, 11-7, 11-9
- message catalog, 2-15
  - default, 1-15, 2-12
  - native language, 1-15, 2-12
  - preprocessor input, 2-12
- message catalog number
  - related to sqlcode, 4-6
- message file, 2-15
- messages from SQLEXPLAIN
  - when produced, 4-7

- modified source file
  - contents, 2-14
  - created by preprocessor, 1-11
  - creation, 2-14
  - inserted constructs, 2-12
  - name, 2-2, 2-6
- ModifiedSourceFileName.o, 2-8
- module
  - definition, 1-11
  - dropping, 2-2
  - installable, 2-23
  - installation, 2-10
  - name, 2-2, 2-8, 2-12, 2-20
  - owner, 2-2, 2-20
  - OWNER authority for, 1-15
  - ownership, 1-18
  - revoking RUN, 2-2
  - storage, 2-20, 8-2
  - updating, 1-20
  - validation, 1-13
- m option
  - full preprocessing mode, 2-1
- multiple rows
  - not allowed in simple data manipulation, 5-2
- multiple rows qualify
  - runtime error, 5-2
- multiple users of LONG columns, 10-14
- multiple warnings
  - SQLEXPLAIN, 4-10

## N

- naming conventions for LONG column files, 10-13
- native language
  - as current language, 1-28
  - current language, 1-28
  - defaults, 1-27
  - message catalog, 1-15
- native language data
  - verifying column definition, 3-23
- native language support
  - overview, 1-27
- n-computer
  - defined, 1-27
- non-dynamic commands, 8-1
- NULL
  - as default data value, 3-17
- null indicator suffix
  - data buffer, 8-15
- null indicator variable
  - in dynamic command, 8-15
- null indicator variables
  - and the INFO command, 3-20
- NULL result of a dynamic fetch of a LONG column, 10-12

- null value
  - in key column of unique index, 5-3
- null values, 8-16
  - and the INFO command, 3-20
  - and unnamed columns in an INSERT, 5-5
  - in a structure declaration, 8-16
  - indicator variables mandatory, 6-3
  - properties of, 3-4
  - runtime errors, 3-4
  - using indicator variables with, 5-6
  - with FETCH, 3-4, 6-3
  - with SELECT, 3-4
- number of rows processed
  - data integrity, 9-1
- number of rows variable, 7-2

## O

- OLTP defined, 1-2
- o option
  - full preprocessing mode, 2-1
- OPEN
  - before BULK FETCH, 7-8
  - cursor processing, 6-3
- OPEN command
  - use with KEEP CURSOR WITH NOLOCKS command, 6-3
- optimization, 1-12
- output device specification
  - definition, 10-5
- overflow
  - of numeric values, 3-24
- OWNER authority
  - and program development, 1-15
  - granting, 1-18
  - o preprocessor option, 2-2

## P

- pathname
  - absolute, 1-22
  - relative, 1-22
- performance
  - integrity constraints, 9-1
  - LONG columns, 10-4
- permanent section
  - and DBEnvironment, 8-2
- p option
  - preprocessor syntax checking mode, 2-6
- PREPARE
  - non-dynamic queries, 8-8
- preprocessor
  - authorization, 1-15
  - DBE sessions, 2-8
  - effect of mode on modified source, 2-14
  - effect on source code, 2-12
  - errors, 2-23

- events, 1-11
- full preprocessing mode, 2-2
- identifying input, 2-12
- include files, 2-31
- invocation, 2-1
- messages, 2-16
- modes, 2-1
- modes and invocation, 2-6
- options, 2-1, 2-6
- parsing, 2-12
- running in background, 2-8
- syntax checking mode, 2-6
- preprocessor banner
  - 300/400 differences, 2-16
- preprocessor directive
  - DECLARE CURSOR, 6-3
- program
  - executable, 2-8
  - execution, 1-19, 2-10
  - maintenance, 1-19
  - methods of linking, 1-13, 1-15
  - name, 2-12
  - obsolescence, 1-21
  - starting DBE session, 2-8
  - steps in compiling, 1-13, 1-15
  - steps in creating, 1-5
  - user authorization, 1-18
- program example
  - date/time data conversion, 11-11
- program structure
  - example, 1-8
- psqlc
  - preprocessor command, 2-8

## Q

- query
  - dynamic data structures, 8-8

## R

- REAL keyword
  - floating point data, 3-10
- RecDB database application design
  - example maintenance menu, 9-4
  - example of deleting data, 9-7
  - example of error checking, 9-4
  - example of inserting data, 9-5
  - example of updating data, 9-6
  - integrity constraints defined, 9-3
- REFETCH command
  - used with LONG columns, 10-10
- relative pathname, 1-22
- restrictions
  - integrity constraints, 9-2
  - LONG columns, 10-4

- retrieving LONG column data
  - with [BULK] SELECT, FETCH, or REFETCH commands, 10-10

- REVOKE, 1-21

- robust program

  - defined, 4-2

- ROLLBACK WORK

  - issued by preprocessor, 2-3

  - to ensure data consistency, 5-9

- rollforward logging

  - and preprocessing, 2-3

- r option

  - full preprocessing mode, 2-1

- row level integrity

  - definition, 9-1

- rows to retrieve

  - data buffer, 8-15

- RUN authority

  - and program development, 1-15

- runtime authorization, 1-19

- runtime errors, 4-2

  - bulk processing, 3-5

  - multiple rows qualify, 5-2

  - null values, 3-4

- runtime events, 1-15

- runtime library, 2-10

- runtime status checking

  - possible errors, 4-1

  - status codes, 4-1

- runtime warnings, 4-2

## S

- sample database

  - authorities, 2-4

  - data consistency, 4-2

  - views, 2-22

- sample program

  - bulk processing, 7-20

  - cex10a, 8-30

  - cex10b, 8-41

  - cex2, 1-31

  - cex5, 4-18

  - cex7, 5-18

  - cex8, 6-30

  - cex9, 7-20

  - cursor processing, 6-25

  - dynamic queries, known format, 8-41

  - dynamic queries, unknown format, 8-23

  - modified source file, 2-25

  - simple data manipulation, 5-10

  - status checking, 4-18

- section

  - and system catalog, 2-19

  - creation, 1-12

  - definition, 1-11, 2-19

  - dynamic vs. non-dynamic, 8-2

  - permanently stored, 8-2

  - purpose, 1-12

  - temporarily stored, 8-2

  - temporary, 8-8

  - types, 2-19

  - validity, 1-12, 2-19

- SELECT

  - and simple data manipulation, 5-1

  - DECLARE CURSOR, 6-2

  - use of COMMIT WORK with, 2-22

- SELECT command

  - used with LONG columns, 10-10

  - with TO\_CHAR function, 11-7

  - with TO\_DATETIME function, 11-6, 11-7

  - with TO\_INTERVAL function, 11-6

- SELECT with CURSOR

  - input host variables only, 6-2

- separate compilable section, 2-8

- shared memory

  - series 300, 1-24

  - use of, 1-23

- shared memory input and output, 10-6

- shared memory problem

  - status checking, 4-30

- signal ipc

  - definition, 1-25

  - how to set, 1-25

  - when to use, 1-25

- signals

  - setting for debugging, 1-26

  - to avoid in applications, 1-26

  - used by ALLBASE/SQL, 1-26

- simple data manipulation

  - commands, 5-1

  - DELETE, 5-7

  - INSERT, 5-5

  - multiple rows not allowed, 5-2

  - sample program, 5-10

  - SELECT, 5-1

  - techniques, 5-1

  - transaction management, 5-7

  - UPDATE, 5-6

- size maximum

  - LONG columns, 10-2

- skeleton program, 1-8

- SMALLINT data declaration, 3-10

- source file, 2-12

- source file;, 2-12

- source file name, 2-2, 2-6

- space allocation for LONG column data, 10-14

- SQL

  - basic statements, 1-7

- sqlca

  - elements of, 4-4

- purpose, 4-4
- SQL case shifting, 2-23
- sqlca.sqlcode
  - introduction, 4-4
  - usage, 4-6
- sqlca.sqlerrd[2]
  - introduction, 4-4
  - usage, 4-8
- sqlca.sqlwarn[0]
  - introduction, 4-4
  - usage, 4-9
- sqlca.sqlwarn[1]
  - introduction, 4-4
  - usage, 4-10
- sqlca.sqlwarn[2]
  - introduction, 4-4
  - usage, 4-11
- sqlca.sqlwarn[3]
  - introduction, 4-4
  - usage, 4-11
- sqlca.sqlwarn[6]
  - introduction, 4-4
  - usage, 4-12
- sqlcode
  - and sqlwarn[6], 4-6
  - a negative number, 4-6
  - ANSI standards, 4-4
  - deadlock detected, 4-30
  - detecting end of scan, 4-35
  - multiple messages, 4-6
  - multiple sqlcodes, 4-6
  - of 0, 4-6
  - of 100, 4-6, 4-35
  - of -10002, 4-36
  - of -14024, 4-12, 4-30
  - of -4008, 4-12
  - related to message catalog number, 4-6
  - SQLLEXPLAIN, 4-7
  - usage, 4-6
  - used with sqlerrd[2], 7-6
- sqlcode of 100, 7-8
- SQL commands
  - comments within, 1-10
  - delimiters for, 1-8
  - location in the code, 1-8
  - syntax, 1-10
- sqllda
  - declaring, 3-25
  - fields, 8-11
  - when fields are set, 8-11
- sqlerrd[2]
  - as counter in display routine, 7-6
  - error checking, 9-1
  - usage, 4-8
  - used with sqlcode, 7-6

- uses for, 4-30
- SQLLEXPLAIN
  - and message catalogs, 1-15
  - introduction, 4-1
  - multiple messages, 4-1
  - multiple warnings, 4-10
  - no message for sqlcode=100, 4-7
  - simultaneous warning and error, 4-9
  - sqlcode, 4-7
  - sqlwarn[0], 4-9
  - using, 4-7
  - when messages are available, 4-13
- sqlmsg, 2-15
- sqlwarn[0]
  - SQLLEXPLAIN, 4-9
  - usage, 4-9
- sqlwarn[1]
  - string truncation, 3-23
  - usage, 4-10
- sqlwarn[2]
  - usage, 4-11
- sqlwarn[3]
  - usage, 4-11
- sqlwarn[6], 4-12
  - transaction rollback, 4-12
  - usage, 4-12
- start a DBE session
  - CONNECT, 1-15
  - START DBE, 1-15
- START DBE
  - and DBA authority, 1-15
  - and the preprocessor, 2-3
  - to start a DBE session, 1-15
- START DBE NEWLOG
  - to increase log space, 2-3
- StartIndex variable
  - defined, 7-2
- statement level integrity
  - and integrity constraints, 9-3
- status checking
  - deadlock, 4-30
  - elements available, 4-4
  - explicit, 4-24
  - explicit defined, 4-1
  - implicit, 4-13
  - implicit defined, 4-1
  - information available, 4-1
  - introduction to explicit, 4-13
  - kinds of, 4-13
  - procedures, 4-17, 4-24
  - purposes of, 4-2
  - runtime techniques, 4-2
  - shared memory problem, 4-30
- status codes
  - runtime status checking, 4-1



- storage
  - LONG columns, 10-4
- syntax checking mode, 2-6
- syntax for date/time functions
  - ADD\_MONTHS, 11-9
  - input functions, 11-3
  - output functions, 11-7
  - TO\_CHAR, 11-7
  - TO\_DATE, 11-3
  - TO\_DATETIME, 11-3
  - TO\_INTEGER, 11-7
  - TO\_INTERVAL, 11-3
  - TO\_TIME, 11-3
- syntax for LONG columns
  - ALTER TABLE command, 10-4
  - CREATE TABLE command, 10-4
  - select list, 10-10
- syntax within embedded SQL, 1-10
- system catalog, 1-12
  - pages locked during preprocessing, 2-3
- T**
- temporary section, 8-8
- terminal IO
  - KEEP CURSOR, 6-15
- TO\_CHAR function
  - example with SELECT command, 11-7
  - syntax, 11-7
- TO\_DATE function
  - example with INSERT command, 11-4
  - example with UPDATE command, 11-5
  - syntax, 11-3
- TO\_DATETIME function
  - example with DELETE command, 11-6
  - example with INSERT command, 11-4
  - example with SELECT command, 11-6, 11-7
  - example with UPDATE command, 11-5
  - syntax, 11-3
- TO\_INTEGER function
  - example with BULK FETCH command, 11-8
  - syntax, 11-7
- TO\_INTERVAL function
  - example with INSERT command, 11-4
  - example with SELECT command, 11-6
  - syntax, 11-3
- TO\_TIME function
  - example with INSERT command, 11-4
  - syntax, 11-3
- transaction management, 4-12
  - cursor processing, 6-9
  - simple data manipulation, 5-7
- truncation
  - detecting in strings, 3-4
  - in string operations, 3-23
  - of numeric values, 3-24

- of UPDATE or DELETE strings, 3-23
- type compatibility, 3-19
  - and the INFO command, 3-20
  - decimal, 3-24
- type conversion, 3-19
- type include file
  - sample, 2-31
- type precedence
  - in numeric conversion, 3-24

## U

- unique index
  - WHERE clause, 5-2
- UPDATE
  - and simple data manipulation, 5-6
- UPDATE and FETCH
  - cursor processing, 6-6
- UPDATE command
  - used with LONG columns, 10-12
  - used with TO\_DATE function, 11-5
  - used with TO\_DATETIME function, 11-5
- UPDATE STATISTICS
  - before preprocessing, 2-3
  - invalidating sections, 2-22
- UPDATE WHERE CURRENT
  - FOR UPDATE OF, 6-2, 6-5
  - restrictions, 6-5
  - syntax, 6-5
- UPDATE WHERE CURRENT command
  - used with LONG columns, 10-12
- updating application programs, 1-20
- USER
  - as default data value, 3-17
- using default data values
  - introduction to, 3-17
- using indicator variables
  - assigning null values, 5-6

## V

- validation, 1-12
  - module, 1-13
- varbinary data
  - using the LONG phrase with, 3-11
- VARCHAR
  - data declaration, 3-8
  - data storage, 3-23
  - dynamic command declaration, 8-16
  - varchar prefix in the data buffer, 8-15
- variable include file
  - sample, 2-33
- views
  - and sections, 2-20

## **W**

warning message  
  and sqlcode, 4-10  
  and sqlwarn[0], 4-10  
warnings

  runtime handling, 4-2

## **WHENEVER**

  components of, 4-13  
  duration of command, 4-7  
  for different conditions, 4-14  
  transaction roll back, 4-14