900 Series HP 3000 Computer Systems

ALLBASE/SQL FORTRAN Application Programming Guide

HEWLETT PACKARD

HP Part No. 36216-90030
Printed in U.S.A. 1992

Third Edition
E0692

Printing History

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

Edition	**Date**	**Software Version**
First Edition	October 1988	36216-02A.12.00
Second Edition	January 1991	36216-02A.20.00
Third Edition	June 1992	36216-02A.E1.00

ALLBASE/SQL MPE XL Documents

General Reference

- Up and Running with ALLBASE/SQL (36389-90011)
- ALLBASE/SQL Reference Manual (36216-90001)
- ALLBASE/ISQL Reference Manual (36216-90004)
- ALLBASE/SQL Quick Reference Guide (36216-90038)
- HP ALLBASE/QUERY User's Guide (92534-90001)
- ALLBASE/SQL Message Manual (36216-90009)

Database and Network Administration

- ALLBASE/SQL Database Administration Guide (36216-90005)
- ALLBASE/NET User's Guide (36216-90031)
- ALLBASE/Turbo CONNECT Administrator's Guide (36385-90001)
- ALLBASE/SQL Performance Guidelines (36389-90001)
- ALLBASE/DB2 CONNECT User's Guide (30700-90001)
- HP ALLBASE/SQL PC API Installation and Administration Guide (B2463-90001)
- HP ALLBASE/SQL PC API User's Guide (B2463-90003)

Embedded SQL Programming Guides

- ALLBASE/SQL C Application Programming Guide (36216-90023)
- ALLBASE/SQL FORTRAN Application Programming Guide (36216-90030)
- ALLBASE/SQL Pascal Application Programming Guide (36216-90007)
- ALLBASE/SQL COBOL Application Programming Guide (36216-90006)
- HP ALLBASE/SQL Release F.0 Application Programming Bulletin (36216-90062)

4GL Programming Guides

- HP ALLBASE/4GL Developer Administration Manual (30601-64001)
- HP ALLBASE/4GL Developer Reference Manual (30601-64002)
- HP ALLBASE/4GL Developer Self-Paced Training Guide (30601-64003)
- HP ALLBASE/4GL Run-Time Administration Manual (30602-64001)

LG200145_004b

Preface

ALLBASE/SQL is a relational database management system for use on HP 3000 Series 900 computers. 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 FORTRAN language source code.

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

MPE/iX, Multiprogramming Executive with Integrated POSIX, is the latest in a series of forward-compatible operating systems for the HP 3000 line of computers. In HP documentation and in talking with HP 3000 users, you will encounter references to MPE XL, the direct predecessor of MPE/iX. MPE/iX is a superset of MPE XL. All programs written for MPE XL will run without change under MPE/iX. You can continue to use MPE XL system documentation, although it may not refer to features added to the operating system to support POSIX (for example, hierarchical directories).

This manual contains both basic and in-depth information about embedding ALLBASE/SQL. Code examples are based, for the most part, on the sample database, PartsDBE, which accompanies ALLBASE/SQL. Refer to Appendix C in the *ALLBASE/SQL Reference Manual* for information about the structure of PartsDBE and for listings of the sample database.

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

Chapters 6 through 12 address the various ways to manipulate data in an ALLBASE/SQL FORTRAN program.

- Chapter 6, "Overview of Data Manipulation," is an overview of data manipulation and the techniques for executing data manipulation commands.
- Chapter 7, "Simple Data Manipulation," explains how to process data one row at a time.
- Chapter 8, "Processing with Cursors," explains the use of a cursor to process a multiple row query result one row at a time.
- Chapter 9, "Using Dynamic Operations," covers the use of ALLBASE/SQL commands that are preprocessed at runtime.
- Chapter 10, "Programming with Constraints," describes ALLBASE/SQL data integrity features and describes how to use them in programs.
- Chapter 11, "Programming with LONG Columns," describes how to program with LONG columns.
- Chapter 12, "Programming with ALLBASE/SQL Functions," contains descriptions of SQL, including date/time functions and the TID function.

Chapters 2, 3, 5, and 7 through 9 contain sample programs for use with the sample database.

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*
bold italics	In a syntax statement, a word in bold italics represents a parameter 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`: $\texttt{COMMAND} \left\{ \begin{matrix} \texttt{ON} \\ \texttt{OFF} \end{matrix} \right\}$
`[  ]`	In a syntax statement, brackets enclose optional elements. In the following example, `OPTION` can be omitted: `COMMAND` *filename* `[OPTION]` When several elements are stacked within brackets, you can select one or none of the elements. In the following example, you can select `OPTION` or *parameter* or neither. The elements cannot be repeated. $\texttt{COMMAND}\ \textit{filename} \left[\begin{matrix} \texttt{OPTION} \\ \textit{parameter} \end{matrix} \right]$

Conventions (continued)

`[ ... ]`	In a syntax statement, horizontal ellipses enclosed in brackets indicate that you can repeatedly select the element(s) that appear within the immediately preceding pair of brackets or braces. In the example below, you can select *parameter* zero or more times. Each instance of *parameter* must be preceded by a comma:
	
	`[,`*parameter*`] [...]`
	
	In the example below, you only use the comma as a delimiter if *parameter* is repeated; no comma is used before the first occurrence of *parameter*:
	
	`[`*parameter*`] [,...]`

`| ... |`	In a syntax statement, horizontal ellipses enclosed in vertical bars indicate that you can select more than one element within the immediately preceding pair of brackets or braces. However, each particular element can only be selected once. In the following example, you must select **`A, AB, BA`**, or **`B`**. The elements cannot be repeated. $\left\{\begin{matrix}\mathtt{A}\\\mathtt{B}\end{matrix}\right\}$ `| ... |`
...	In an example, horizontal or vertical ellipses indicate where portions of an example have been omitted.
`Δ`	In a syntax statement, the space symbol `Δ` shows a required blank. In the following example, *parameter* and *parameter* must be separated with a blank: `(`*parameter*`)Δ(`*parameter*`)`
( )	The symbol ( ) indicates a key on the keyboard. For example, (RETURN) represents the carriage return key or (Shift) represents the shift key.
(CTRL)*character*	(CTRL)*character* indicates a control character. For example, (CTRL)`Y` means that you press the control key and the Y key simultaneously.

Contents

Figures

Tables

1

Getting Started with ALLBASE/SQL Programming in FORTRAN

The steps in creating a FORTRAN application program that accesses an ALLBASE/SQL **DBEnvironment** are summarized in Figure 1-1.

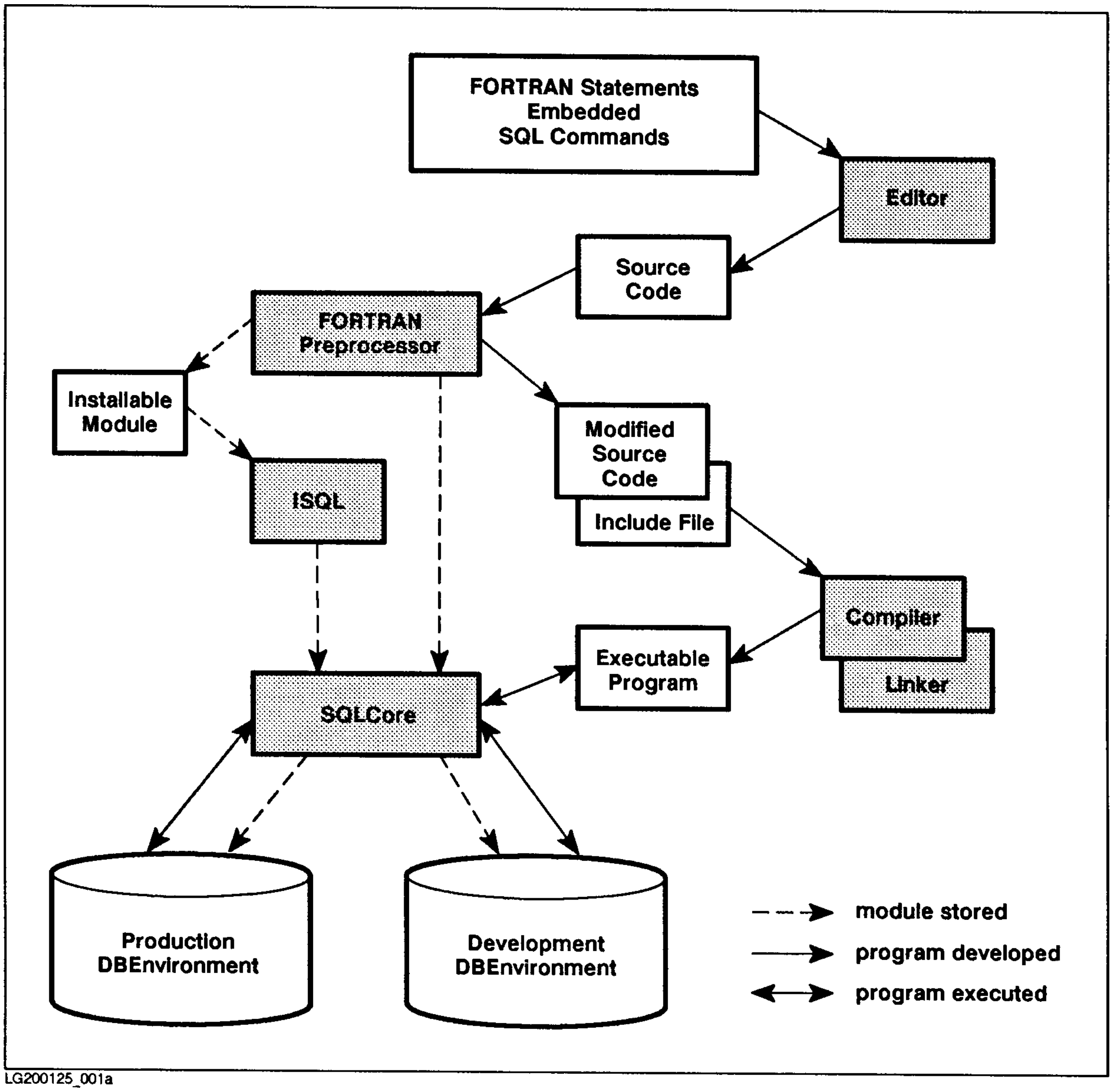


Figure 1-1. Creating an ALLBASE/SQL FORTRAN Application Program

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

Before compiling the source code, it must be *preprocessed* using the ALLBASE/SQL **FORTRAN preprocessor**. The FORTRAN preprocessor:

- Checks the syntax of the SQL commands.
- Stores a **module** in the system catalog of the DBEnvironment to be accessed at runtime. A module consists of ALLBASE/SQL instructions for executing SQL commands in your program.
- Creates an **installable module file**. This file contains a copy of the module stored in the DBEnvironment at preprocessing time. This file can be used to install the module into another DBEnvironment so that the application program can be run in that DBEnvironment.
- Generates FORTRAN statements for executing the SQL commands and comments out the SQL commands. Non-SQL statements are ignored. This modified version of your source code is placed into a file created by the preprocessor, referred to as a **modified source code file**.
- Creates one **include file**, which contains type declarations and initialization of variables used by the preprocessor generated FORTRAN statements.

The **FORTRAN compiler** and **system linker** are used to create the **executable program** from the modified source code file and the include file. The executable program automatically makes the appropriate database accesses at runtime in DBEnvironments where the related module is stored.

ALLBASE/SQL FORTRAN Programs

To write a FORTRAN application that accesses an ALLBASE/SQL database, SQL commands are be embedded in the FORTRAN source to:

- Start or terminate a DBEnvironment session, either in single-user mode or multiuser mode.
- Start or terminate a transaction.
- Retrieve rows from or change data in tables in a database.
- Create or drop objects, such as indexes or views.

Special SQL commands known as **preprocessor directives** may also be embedded in the FORTRAN source. The FORTRAN preprocessor uses these directives to:

- Identify FORTRAN variables referenced in SQL commands, known as **host variables**.
- Set up a common block, known as the **SQL Communications Area (SQLCA)**, for communicating the status of executed SQL commands to your program.
- Optionally automate program flow based on SQLCA information.
- Identify cursor declarations.

Program Structure

The following skeleton program illustrates the relationship between FORTRAN statements and embedded SQL commands in an application program. SQL commands may appear in a program at locations indicated by boldface notations. The SQLCA Common Block Declaration may appear either before or after the FORTRAN type declaration section. The SQLCA Common Block Declaration must appear, however, before the host variable declaration section which must be the last of the type declarations in the program unit. Refer to Chapter 3 for further clarification of program structure.

```
PROGRAM ProgramName
FORTRAN Statements
.
.
.
SQLCA Declaration
FORTRAN Type Declarations
Host Variable Declarations
.
.
.
FORTRAN Statements, some containing SQL Commands
.
.
.
END
SUBROUTINE SubroutineName
FORTRAN Statements
.
.
.
SQLCA Declaration
FORTRAN Type Declarations
Host Variable Declarations
.
.
.
FORTRAN Statements, some containing SQL Commands
.
.
RETURN
END
```

To delimit SQL commands for the preprocessor, each SQL command is prefixed by **EXEC SQL**:

```
EXEC SQL BEGIN WORK
```

SQL commands may appear in the main program or any subprogram unit where you establish DBEnvironment access and manipulate data in a database.

DBEnvironment Access

You must always specify a DBEnvironment at preprocessing time. The preprocessor needs to access the DBEnvironment you specify in the INFO string. It does so in order to store a module containing permanent sections used by your application program at runtime. In this example, the environment is SomeDBE which is in the group and account SomeGroup.SomeAcct.

```
: RUN PSQLFOR.PUB.SYS; INFO = "SomeDBE.SomeGroup.SomeAcct"
```

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

```
SUBPROGRAM Unit
.
.
.
EXEC SQL CONNECT TO 'SomeDBE.SomeGroup.SomeAcct'
.
.
.
EXEC SQL RELEASE
RETURN
END
```

At runtime, the program starts a DBE session in *SomeDBE.SomeGroup.SomeAcct*, where a module for the program has been stored.

A program can accept a DBEnvironment name from the program user and dynamically preprocess the SQL command that starts a DBEnvironment session. Refer to Chapter 9 for more information on dynamically connecting to a database and refer to Chapter 4 for more information on using a host variable to connect to a database.

No matter how you access a DBEnvironment (dynamic or stored sections), you must always specify a DBEnvironment name when you preprocess.

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

An alternative method of accessing more than one DBEnvironment from the same program would be to separate the program into separate compilable files. Each source file would access a DBEnvironment. In each file you start and terminate a DBE session for the DBEnvironment accessed. You then preprocess and compile each file separately. When you invoke the preprocessor, you identify the DBEnvironment accessed by the source file being preprocessed.

After 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, you link them to create the executable program.

Note that a program which accesses more than one DBEnvironment must do so in sequence. Such program design may adversely affect performance and requires special consideration.

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

Authorization

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

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

To **run** a program accessing an ALLBASE/SQL DBEnvironment, you need the following authorities:

- If the program uses a CONNECT command to start a DBE session, you need both CONNECT authority and either 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 runtime, any SQL command in the program, except for the command used to start the DBE session, is executed only if the OWNER of the module has the authorization to execute the command at runtime. However, any dynamic command is executed only if the individual running the program has the authority to execute the command at run time. A **dynamic command** is an SQL command entered by the user at runtime.

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

File Referencing

When you create a DBEnvironment, a Database Environment Configuration (DBECon) file is created. The file name of this DBECon file is stored in the DBECon file itself. In all subsequent references to files, you may use either a fully qualified file name or a file name relative to that of the DBECon file.

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

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

and the user is currently in the SQL group of the DBSUPPORT account, the file name PARTSDBE.SQL.DBSUPPORT is stored in the DBECon file. If the user subsequently creates a DBEFile with the command:

```
   CREATE DBEFILE ORDERS WITH PAGES=50, NAME='ORDERSFS'
```

the ORDERSFS file is created in the same group and account as the DBECon file with the name ORDERSFS.SQL.DBSUPPORT. If however, the user creates a DBEFile with the command:

```
   CREATE DBEFILE ORDERS WITH PAGES=50, NAME='DBSUPPORT'
```

the name stored in the DBECon file is ignored while creating this file. The user then needs to fully qualify this file name each time the file is referenced. Remember, a file, group, or account name can contain a maximum of 8 bytes. Fully qualified file names, enclosed in quotes, are restricted to a maximum length of 36 bytes.

In addition, if the DBEnvironment you want the preprocessor to access resides in a group and account other than your current group and account, you will have to qualify the name of the DBEnvironment.

For example, if the DBEnvironment you want the preprocessor to access resides in the SQL group of account DBSUPPORT, you would invoke the preprocessor as follows:

```
RUN PSQLFOR.PUB.SYS;INFO = 'SOMEDBE.SQL.DBSUPPORT'
```

Native Language Support

ALLBASE/SQL lets you manipulate databases in a wide variety of native languages in addition to the default language, known as **NATIVE-3000**. 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.

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.

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.

Note that MPE XL does not support native language file names nor DBEnvironment names.

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 MPE job control word NLUSERLANG to the number (*LangNum*) of the native language you wish to use. Use the following MPE XL command:

   `SETJCW NLUSERLANG =` *LangNum*

   This language then becomes the **current language**. (If NLUSERLANG is not set, the current language is NATIVE-3000.)
3. Use the `LANG =` *LanguageName* option of the START DBE NEW command to specify the language when you create a DBEnvironment.

Run the MPE XL utility program NLUTIL.PUB.SYS to determine which native languages are supported on your system. Here is a list of some supported languages, preceded by the *LangNum* for each:

0	NATIVE-3000	9	ITALIAN	52	ARABICW
1	AMERICAN	10	NORWEGIAN	61	GREEK
2	C-FRENCH	11	PORTUGUESE	71	HEBREW
3	DANISH	12	SPANISH	81	TURKISH
4	DUTCH	13	SWEDISH	201	CHINESE-S
5	ENGLISH	14	ICELANDIC	211	CHINESE-T
6	FINNISH	41	KATAKANAC	221	JAPANESE
7	FRENCH	51	ARABIC	231	KOREAN
8	GERMAN				

Any one user would not be using all of these languages at one time, of course. The languages you want need to be initialized on the MPE XL system and the MPE XL system rebooted. Once that is done, the NLUTIL.PUB.SYS program might display the following, assuming that the languages listed are those that have been initialized on MPE XL:

```
Lang   Lang                        Char        Char
  ID   Name                          ID        Name
----   ----                        ----        ----
   0   NATIVE-3000                    0        USASCII
   1   AMERICAN                       1        ROMAN8
   5   ENGLISH                        1        ROMAN8
   7   FRENCH                         1        ROMAN8
   8   GERMAN                         1        ROMAN8
```

Note that ENGLISH is British English and AMERICAN is American English.

Resetting NLUSERLANG while you are connected to a DBEnvironment has no effect on the current DBE session.

The ALLBASE/SQL FORTRAN Preprocessor

The FORTRAN preprocessor that is part of ALLBASE/SQL is specifically for FORTRAN 77 programs. Although the preprocessor ignores FORTRAN statements in your source code, it generates FORTRAN statements, based on embedded SQL commands.

Figure 1-2 summarizes the four main preprocess-time events:

- Syntax checking of SQL commands and host variable declarations.
- Creation of compilable files: one modified source code file and one include file.
- Creation of an installable module.
- Storage of a module in the system catalog.

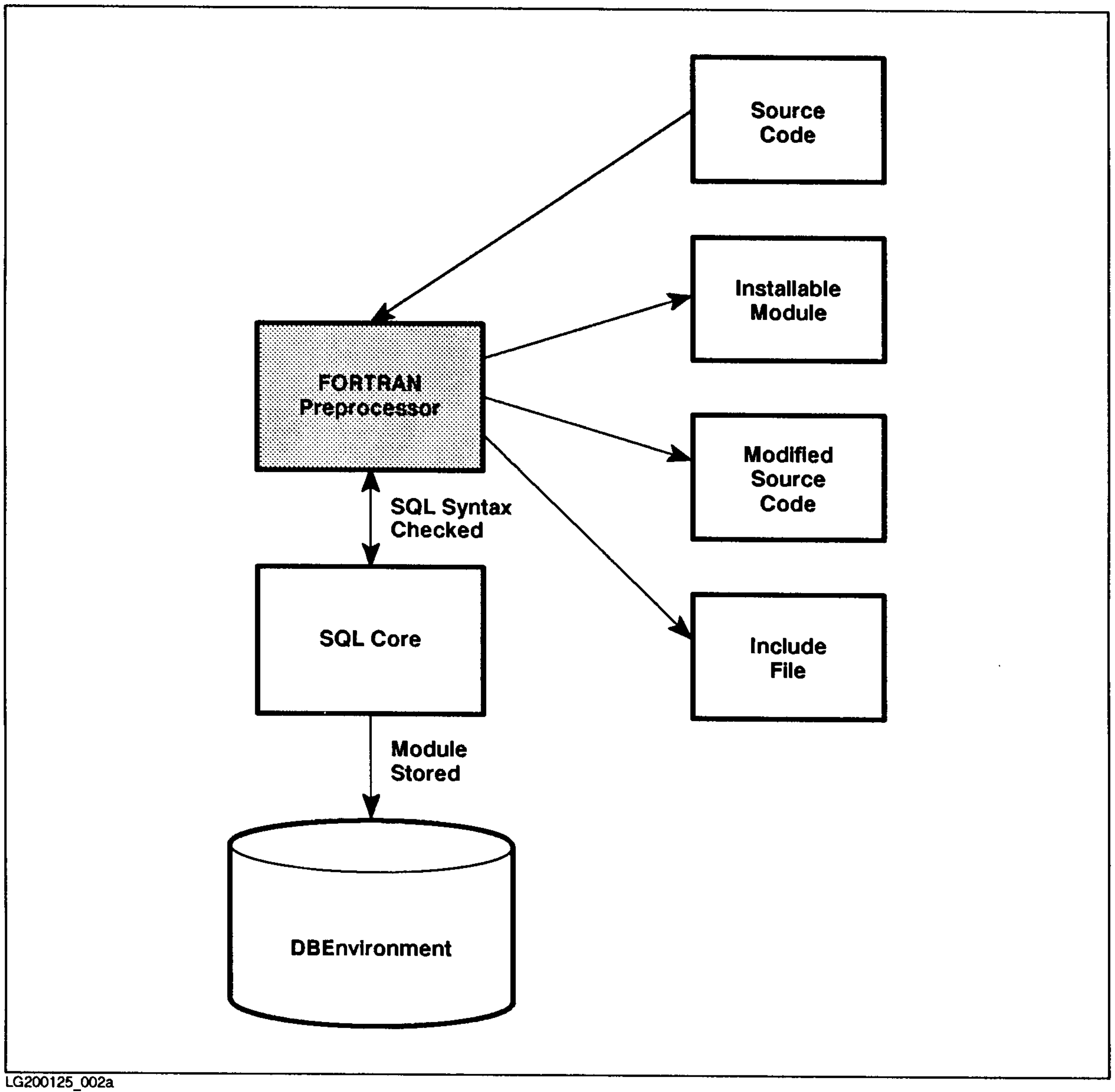


Figure 1-2. Preprocess-Time Events

Effect of Preprocessing on Source Code

The FORTRAN preprocessor scans the source code for SQL commands. If the syntax of an SQL command is valid, the preprocessor converts the command to compilable FORTRAN statements that call ALLBASE/SQL external procedures at runtime. During preprocessing, for example, the SQL command:

```
      EXEC SQL SELECT  PartNumber, PartName, SalesPrice
     1          INTO :PartNumber,
     2               :PartName,
     3               :SalesPrice
     4          FROM  Purchdb.Parts
     5         WHERE  PartNumber = :PartNumber
```

is converted into the following modified source code statements:

```
C**** Start SQL Preprocessor ****
C           EXEC SQL SELECT  PartNumber, PartName, SalesPrice
C    1                 INTO :PartNumber,
C    2                       :PartName,
C    3                       :SalesPrice :SalesPriceInd
C    4                 FROM  PurchDB.Parts
C    5                WHERE  PartNumber = :PartNumber
C
C**** Start Inserted Statements ****
      WRITE(SQLTMP,'(A16)')PartNumber
      CALL SQLXFE(SQLCAID,SQLOWN,SQLMDN,1,SQLTMP,16,56,1)
      IF (SQLCODE .EQ. 0) THEN
      READ(SQLTMP,'(A16,A30,A8,A2)')PartNumber,PartName,SalesPrice,Sales
     1PriceInd
      ELSE
      END IF
C**** End SQL Preprocessor   ****
```

The embedded SELECT command has been converted into a FORTRAN comment, and FORTRAN statements that enable ALLBASE/SQL to execute the SELECT command at runtime have been inserted. The names that appear in the inserted FORTRAN code (italicized in the above example) identify variables used by the ALLBASE/SQL external procedures; in this example, the names identify variables used by the SQLXFE external procedure. Some of these variables are derived from **host variables**. As shown in the embedded SELECT command above, you precede a host variable with a colon when you use it in SQL commands:

```
:PartNumber
```

Type declarations used by preprocessor generated code are defined and initialized in the include file the preprocessor creates. The preprocessor inserts INCLUDE statements that reference this file in each program unit of the modified source code after the host variable declarations. Even if you do not declare host variables you must still include the EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION commands in order for the preprocessor to create and insert this include file:

```
INCLUDE 'SQLVAR'
```

Caution Never modify the statements inserted by the preprocessor in the modified source code file, or the include file the preprocessor creates. Changes to preprocessor generated information could damage your DBEnvironment or your system.

Effect of Preprocessing on DBEnvironments

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

When the preprocessor encounters a syntactically correct SQL command, it creates a *section* and stores that section in the system catalog of the DBEnvironment being accessed. An ALLBASE/SQL section is a group of stored ALLBASE/SQL instructions for executing one SQL command.

All sections created during a preprocessing session constitute a **module**. The preprocessor derives the name of the module from the PROGRAM statement or subroutine name unless you supply a different name when you invoke the preprocessor:

```
:RUN PSQLFOR.PUB.SYS;INFO = "DBEnvironmentName (MODULE(ModuleName))"
```

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

The Stored Section

A section consists of ALLBASE/SQL instructions for executing an SQL command. The preprocessor creates a section and assigns a unique section number for all embedded ALLBASE/SQL commands in a module except:

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

Purpose of Sections

A section serves two purposes:

- Access validation: Before executing a stored section at runtime, ALLBASE/SQL ensures that any objects referenced exist and that runtime authorization criteria are satisfied.
- Access optimization: If ALLBASE/SQL has more than one way to access data, it determines the most efficient method and creates the section based on that method. Indexes, for example, can expedite the performance of some queries.

By creating and storing sections at preprocessing time rather than at runtime, you improve runtime performance.

Section Validity

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

- Changes in authorities of the module's owner.
- Alterations to tables accessed by the program.
- Deletions or creations of indexes.
- Updating a table's statistics.

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

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

The Compiler and the Linker

Figure 1-3 summarizes the steps in creating an executable ALLBASE/SQL FORTRAN program from the files created by the FORTRAN preprocessor.

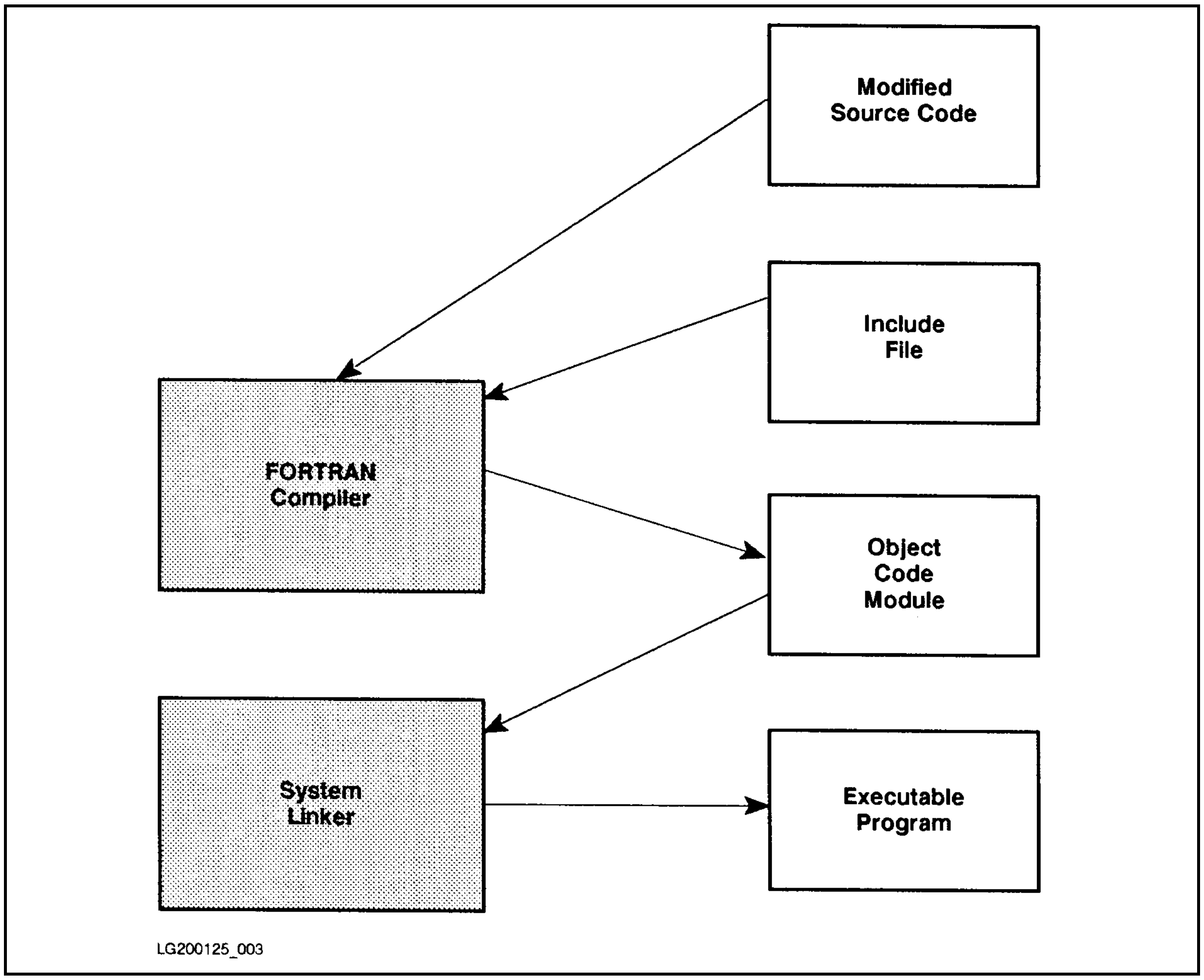


Figure 1-3. Compile-Time and Link-Time Events

You must use native mode to compile and link your program.

You submit to the FORTRAN compiler a modified source code file and the related include file(s) created by the preprocessor. The compiler then generates an object code file. To convert one or more object code files into an executable program, you link them by invoking the linker. This step creates an executable program file. Refer to the Chapter "Using the ALLBASE/SQL FORTRAN Preprocessor" for more information on compiling and linking.

The following example illustrates the preprocessor storing a module in a DBEnvironment located in a group other than that in which preprocessing was initiated. An executable program named SOMEPROG is created in the GROUP1 group after a module named SOMEMOD is stored by the FORTRAN preprocessor in a DBEnvironment named SOMEDBE.GROUP2.ACCTDB. Note that PGMR1 must have appropriate ALLBASE/SQL and operating system file authorities as well.

```
:HELLO PGMR1.ACCTDB.GROUP1
 .
 .
 .
:RUN PSQLFOR.PUB.SYS; INFO = 'SOMEDBE.GROUP2 (MODULE(SOMEMOD))'
 .
 .
 .
:CCXLLK ModifiedSourceFile,SOMEPROG,$NULL
```

The Executable 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 PGMR1.

```
:RUN SOMEPROG.GROUP1.ACCTDB
```

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

- Load the executable program file onto the machine where the production DBEnvironment resides.
- Install any related module(s) in the production DBEnvironment.
- Ensure necessary module owner authorities exist.
- Grant required authorities to program users.

Installing the Program Module

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

```
isql=> CONNECT TO 'SOMEDBE.GROUP3.ACCTDB;
isql=> INSTALL SOMEMOD.GROUP2.ACCTDB;

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

isql=> COMMIT WORK;
```

ISQL copies the module from the installable module file named SOMEMOD.GROUP2.ACCTDB into a DBEnvironment named SOMEDBE.GROUP3.ACCTDB. During installation, ALLBASE/SQL marks each section in the module valid or invalid, depending on the current objects and authorities in SOMEDBE.GROUP3.ACCTDB.

To use the INSTALL command, you need to be able to start a DBE session in the DBEnvironment that is to contain the new module. If you are replacing a module with a new one of the same name, make sure no other users are accessing the module. To avoid problems, install modules while connected to the DBEnvironment in single-user mode.

Granting Required Owner Authorization

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

If module PGMR1@ACCTDB.SOMEMOD contains a SELECT command for table PURCHDB.PARTS, the following grant would ensure valid owner authorization:

```
isql=>  GRANT SELECT on PURCHDB.PARTS TO PGMR1@ACCTDB;
```

If PGMR1@ACCTDB had DBA authority, he could have assigned ownership of the module to another owner by using the OWNER parameter.

```
:RUN PSQLFOR.PUB.SYS;INFO='SOMEDBE.GROUPDB.ACCTDB &
(MODULE(SOMEMOD) OWNER(PURCHDB))'
```

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

```
isql=>  GRANT SELECT ON PURCHDB.PARTS TO PURCHDB;
```

Granting Program User Authorization

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

```
RUN
module OWNER
DBA
```

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

```
isql=> CONNECT TO 'SOMEDBE.GROUP3.ACCTDB';
isql=> GRANT CONNECT TO SOMEUSER@SOMEACCT;
isql=> COMMIT WORK;
```

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

```
isql=> CONNECT TO 'SOMEDB.GROUP3.ACCTDB';
isql=> GRANT RUN ON PGMR1@ACCTDB.SOMEMOD TO SOMEUSER@SOMEACCT;
isql=> COMMIT WORK;
```

Now SOMEUSER@SOMEACCT can run program SOMEPROG.GROUP1.ACCTDB which accesses module PGMR1@ACCTDB.SOMEMOD.

```
:HELLO SOMEUSER.SOMEACCT
 .
 .
 .
:RUN SOMEPROG.GROUP1.ACCTDB
```

Note that if a program contains more than one module (multiple application source files), you need to GRANT RUN authority to the user for each module.

Running the Program

At runtime, two file equations may be required, one for the ALLBASE/SQL message catalog and one for the DBEnvironment to be accessed by the program.

If the program contains the SQLEXPLAIN command, the ALLBASE/SQL message catalog must be available at runtime. SQLEXPLAIN obtains warning and error messages from the message catalog (SQLCT*xxx*.PUB.SYS). If SQLCT*xxx* is installed in a different group or account on your system, you must use a file equation to specify its location. See Chapter 2 for further information on the ALLBASE/SQL message catalog.

If the program contains a CONNECT or START DBE command that uses a back referenced DBEnvironmentName, submit a FILE command to identify the DBEnvironment to be accessed by the program at runtime:

```
EXEC SQL CONNECT TO '*DBE'
```

This command initiates a DBE session in the DBEnvironment identified at runtime as follows:

```
:FILE DBE = SOMEDBE.SOMEGRP.SOMEACCT
```

Once you identify the ALLBASE/SQL message catalog and appropriate DBEnvironment, you can run the program:

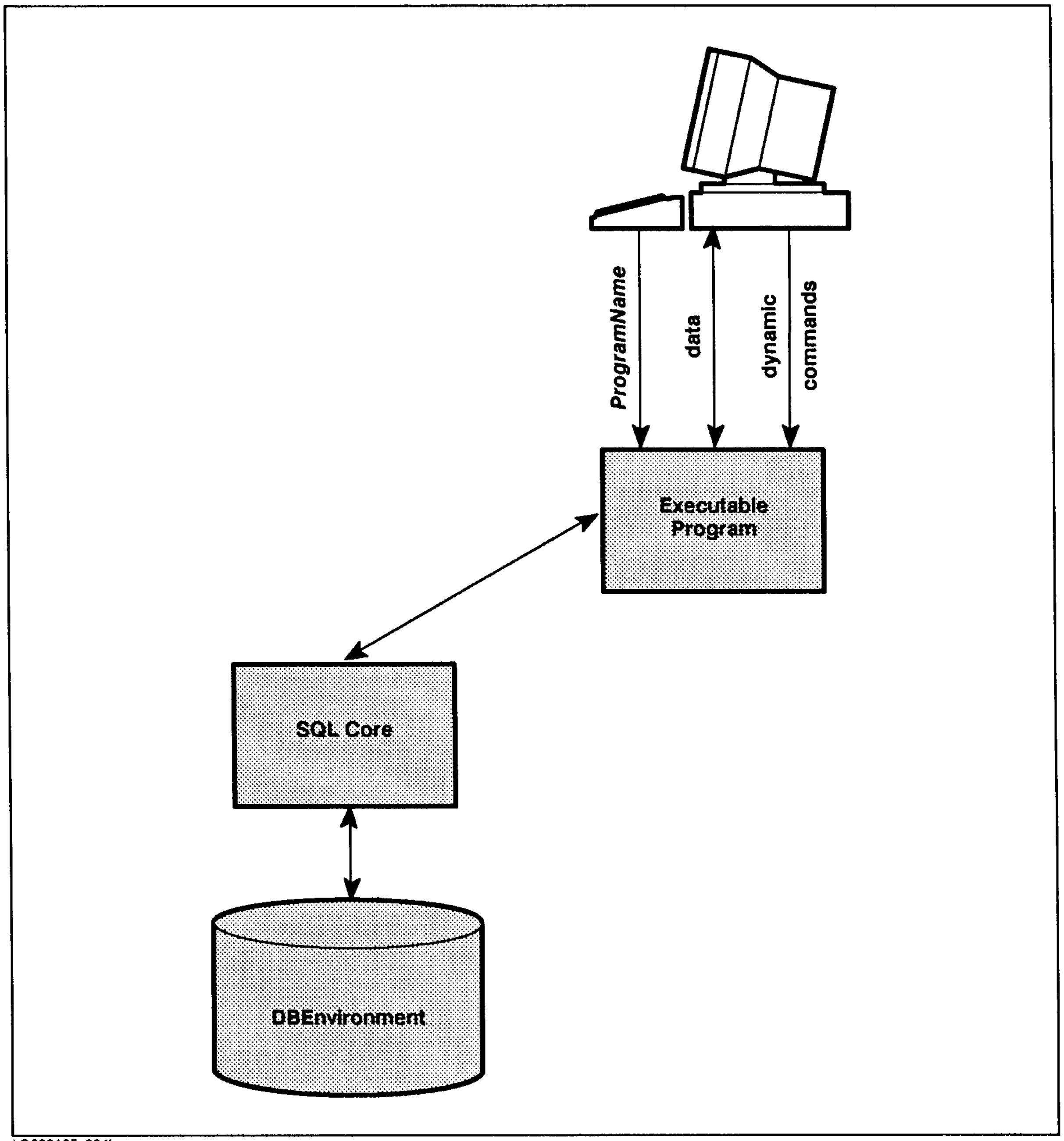
```
:RUN SOMEPROG.GROUP1.ACCTDB
```

You must specify the name of an executable program file as SOMEPROG. Do not specify a module name in the RUN command.

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

All the FORTRAN 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 runtime or can be validated by ALLBASE/SQL at runtime.

Dynamic commands are those not known until runtime. Such commands can be entered by the user at runtime. ALLBASE/SQL converts these commands into executable ALLBASE/SQL instructions at runtime rather than at preprocessing time. Sections and other instructions created for dynamic data manipulation commands are deleted at the end of the transaction. Dynamic commands are described in more detail in the chapter "Using Dynamic Operations."



LG200125_004b

Figure 1-4. Runtime Events

Maintaining ALLBASE/SQL Programs

After FORTRAN programs are in production use, changes in applications, personnel, or databases may necessitate:

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

Updating Application Programs

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

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

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

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

While in ISQL, to delete a module and any existing RUN authorities for it simply omit the PRESERVE option.

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

```
   :RUN PSQLFOR.PUB.SYS;INFO="SomeDBE (MODULE(MYMOD) DROP REVOKE)"
```

This invocation line drops the module named MYMOD, and revokes any related RUN authorities. To revoke the RUN authorities, you would specify the REVOKE option along with the DROP option in the command string.

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

Changing Program-Related Authorization

Once a program is in production use, granting and revoking RUN and CONNECT authority may be necessary as program users change.

Revoking CONNECT authority requires DBA authorization:

```
   isql=> REVOKE CONNECT FROM OLD@USER;
```

Revoking RUN authority requires either module OWNER or DBA authority:

```
   isql=> REVOKE RUN ON PGMR1@GROUPC.SOMEMOD FROM OLD@USER;
```

Obsoleting Programs

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

```
   isql=> DROP MODULE MYMOD
```

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

2

Using The ALLBASE/SQL FORTRAN Preprocessor

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

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

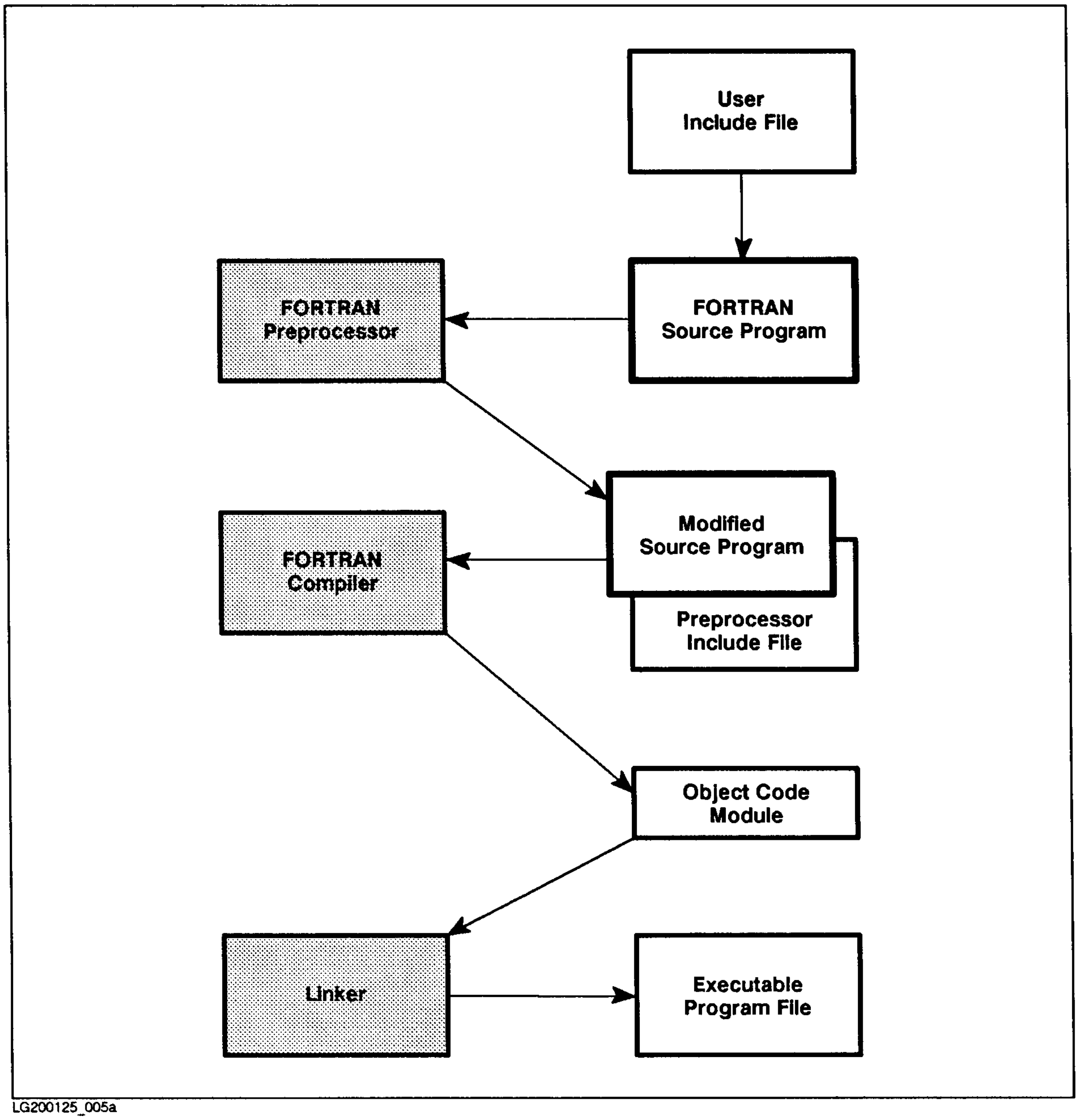
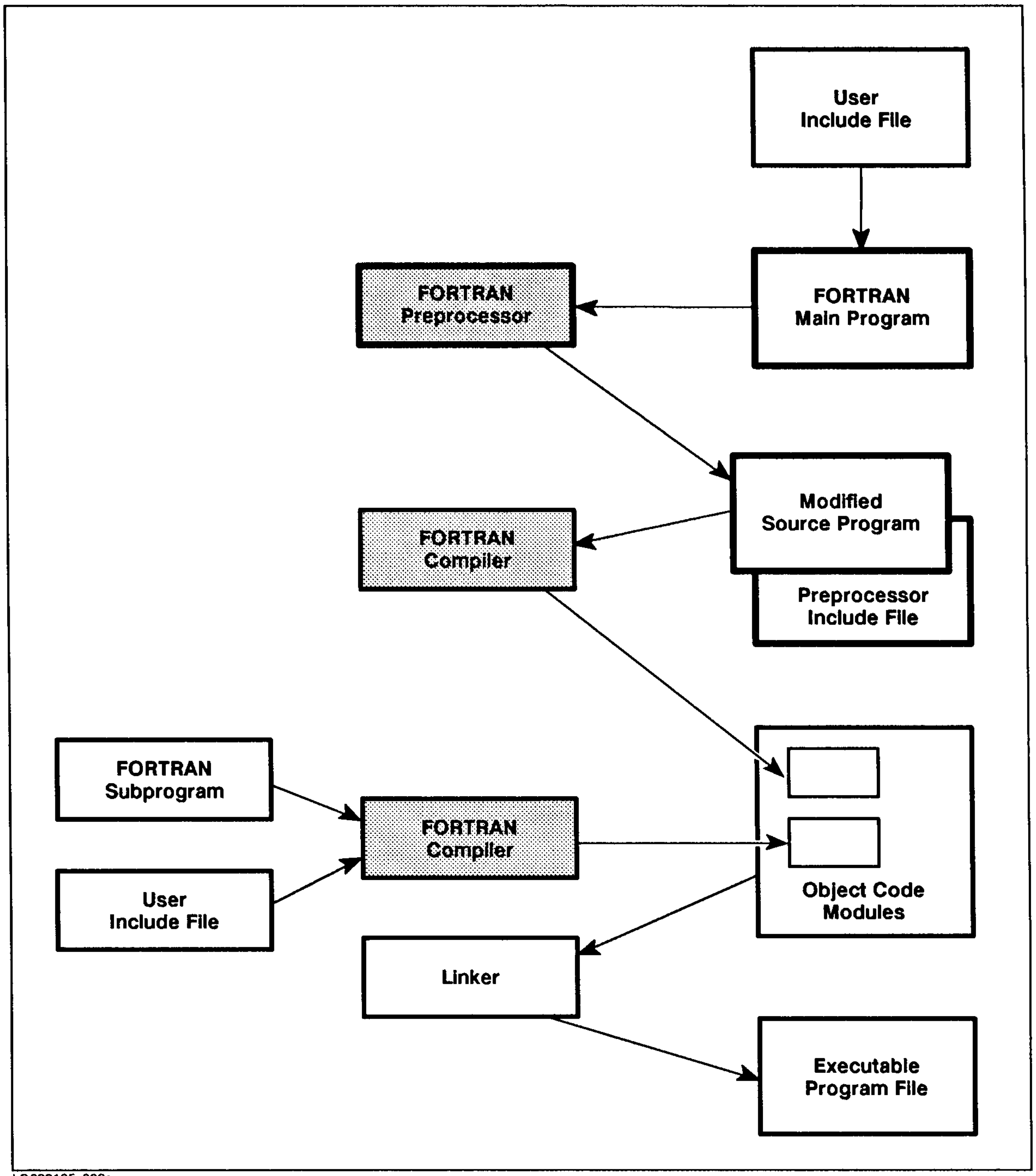


Figure 2-1. Developing a FORTRAN Program

In other cases, the ALLBASE/SQL application program might consist of a main program unit and one or more subprogram units in separate files. In these cases, only source files containing embedded SQL code need to be preprocessed, as illustrated in Figure 2-2. However, each program unit which contains SQL commands must be preprocessed and compiled before the next program unit is preprocessed. Separately preprocessed program units that are not immediately compiled will write over each other's preprocessor created include file and consequently create an error when compiled. You invoke the FORTRAN preprocessor and compiler as many times as necessary to create the desired number of object code modules.



LG200125_006a

Figure 2-2. Developing a FORTRAN Program with Subprograms

During preprocessing, the FORTRAN preprocessor actually accesses the same DBEnvironment to be used by your main program or subprogram unit at runtime. The preprocessor stores a module in the DBEnvironment which is executed at runtime. The module is used at runtime to optimize and validate DBEnvironment operations.

FORTRAN Preprocessor

Compiler Directives

The FORTRAN preprocessor supports the following compiler directives:

- `$`[*option*]`SET` (*identifierlist*)
- `$`[*option*]`IF` (*conditionlist*)
- `$`[*option*]`ELSE`
- `$`[*option*]`ENDIF`
- `$`[*option*]`INCLUDE` '*filename*'

The $IF directive does not support logical operators. (Note that the compiler and preprocessor may not support the same things.) Only simple TRUE and FALSE logical variables are supported. If the $INCLUDE directive is used and the user include file contains embedded SQL commands, there cannot be duplicate host variable type declaration sections. The program unit being preprocessed and the file that is included cannot both contain BEGIN DECLARE SECTION and END DECLARE SECTION commands. However, one or the other may contain a host variable type declaration section.

The $ must be the first character in the line, and must be followed by any compiler directives. For the Series 900, refer to the *HP FORTRAN 77/XL Reference Manual* for MPE XL on HP 3000 Series 900 Computers (Part Number 31501-90010) for more information on FORTRAN compiler directives.

Preprocessor Modes

You can use the preprocessor in two modes:

1. Syntax checking mode, which *only* checks your SQL syntax.
2. Full preprocessing mode, which includes SQL syntax checking, creating compilable output, storing a module in a DBEnvironment, and creating a file that contains an installable copy of the stored module.

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

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

Preprocessor Input and Output

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

- **source file**: a file containing the source code of the FORTRAN program with embedded SQL commands for one or more DBEnvironments. The default input filename is:

  SQLIN

  An alternative name can be specified by using a file equation as shown later in this chapter.

- **ALLBASE/SQL message catalog**: a file containing preprocessor messages and ALLBASE/SQL error and warning messages. The formal file designator for the message catalog is as follows, with *xxx* being the numeric representation for the current native language:

  SQLCT*xxx*.PUB.SYS

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

  As Figure 2-3 points out, the FORTRAN preprocessor creates the following output files:

- **modified source file**: a file containing the modified version of the source code in the source file. The default filename for this file is:

  SQLOUT

  An alternative name can be specified by using a file equation.

- **variable include file**: the name for this file, which contains variable declarations used by FORTRAN statements that the preprocessor inserts into the modified source file is:

  SQLVAR

  Both SQLOUT and SQLVAR are created as permanent files in order to invoke the FORTRAN compiler, as shown in Figure 2-4.

- **ALLBASE/SQL message file**: a file containing the preprocessor banner, error, and warning messages, and other messages. The file name for this file is:

  SQLMSG

- **installable module file**: a file containing a copy of the module created by the preprocessor. The file name for this file is:

  SQLMOD

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

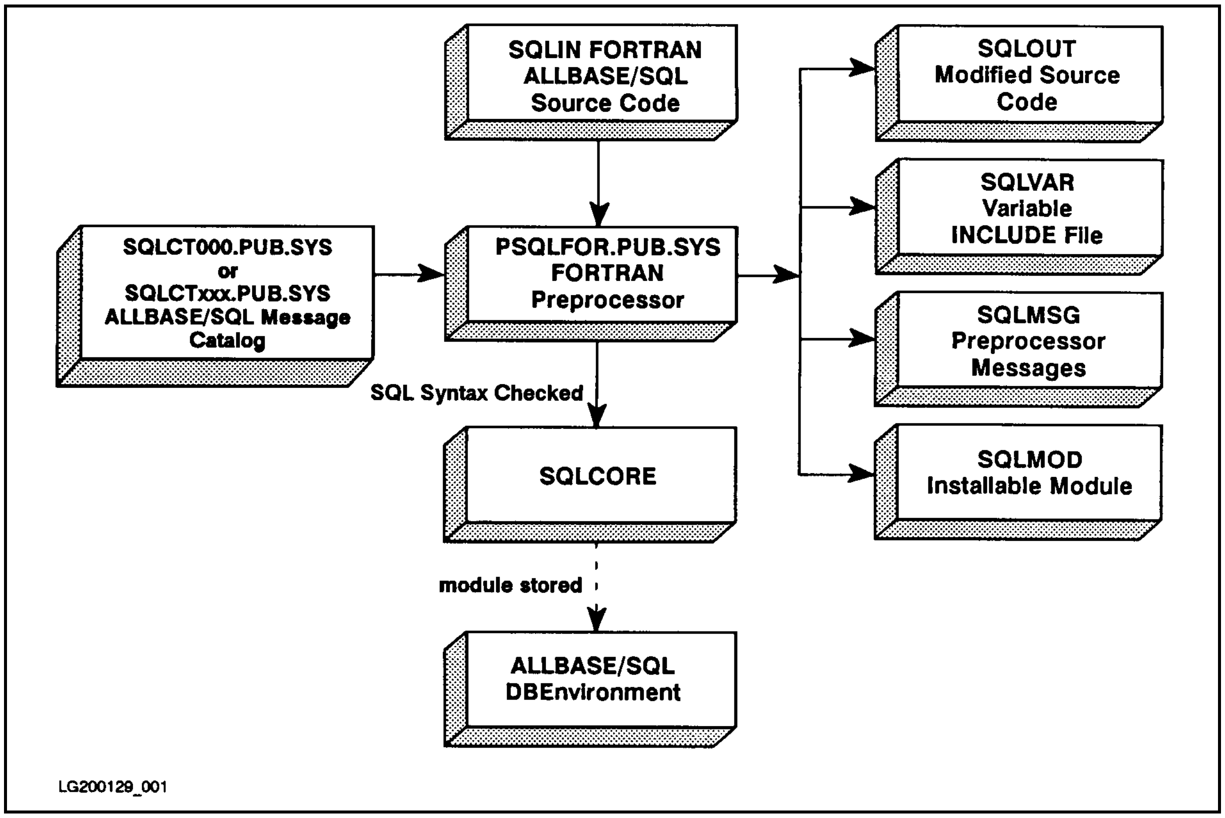

SQLIN FORTRAN ALLBASE/SQL Source Code
SQLCT000.PUB.SYS or SQLCTxxx.PUB.SYS ALLBASE/SQL Message Catalog
PSQLFOR.PUB.SYS FORTRAN Preprocessor
SQLOUT Modified Source Code
SQLVAR Variable INCLUDE File
SQLMSG Preprocessor Messages
SQLMOD Installable Module
SQL Syntax Checked
SQLCORE
module stored
ALLBASE/SQL DBEnvironment
LG200129_001

Figure 2-3. FORTRAN Preprocessor Input and Output

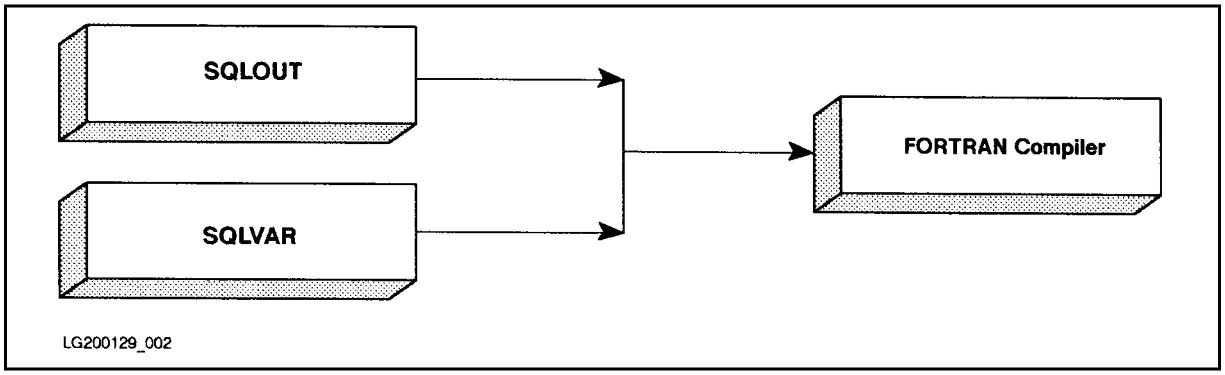


Figure 2-4. FORTRAN Compiler Input

If you want to preprocess several ALLBASE/SQL application programs in the same group and account and compile and link the programs later, or you plan to compile a preprocessed program during a future session, you should do the following for each program:

- Before running the preprocessor, equate SQLIN to the name of the file containing the application you want to preprocess:

```
    :FILE SQLIN = InFile
```

- After running the preprocessor, save and rename the output files if you do not want them overwritten. For example:

```
    :SAVE SQLOUT
    :RENAME SQLOUT, OutFile
    :SAVE SQLMOD
    :RENAME SQLMOD, ModFile
    :SAVE SQLVAR
    :RENAME SQLVAR, VarFile
```

- When you are ready to compile the program, you must equate the include file name to its standard ALLBASE/SQL name (SQLVAR).

Source File

The preprocessor source file must contain at a minimum the following statements:

```
PROGRAM Statement
```
AnyStatement
```
END
```

When parsing the source file, the FORTRAN preprocessor ignores all FORTRAN statements and any FORTRAN compiler directives that are not supported. Only the following information is parsed by the FORTRAN preprocessor:

- The PROGRAM Statement or SUBROUTINE name. Unless you specify a module name in the preprocessor invocation line, the preprocessor uses the PROGRAM Statement or the SUBROUTINE name to name the module it stores. A module name can contain as many as 20 bytes and must follow the rules governing ALLBASE/SQL basic names (given in the *ALLBASE/SQL Reference Manual*).
- Statements found after the prefix `EXEC SQL`. Follow the rules given in Chapter 3 for how and where to embed these statements.
- Statements found between the BEGIN DECLARE SECTION and END DECLARE SECTION commands. These commands delimit a **declare section**, which contains FORTRAN data description entries for the host variables used in that program or subprogram unit. All program units (both main and subprogram) that contain SQL commands, regardless of whether or not they contain host variables, must include the BEGIN DECLARE SECTION and the END DECLARE SECTION commands in order to create the variable include file. Host variables are described in Chapter 4.
- The FORTRAN compiler directives $SET, $IF, $ELSE, $ENDIF, and $INCLUDE are supported by the FORTRAN preprocessor. All other compiler directives are ignored.

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

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

As the following interactive sample dialog illustrates, the program begins a DBE session for PartsDBE, the sample DBEnvironment. It prompts the user for a part number, then displays information about the part from the table PurchDB.Parts. Warning and error conditions are handled with WHENEVER and SQLEXPLAIN commands. The program continues to prompt for a part number until the user enters a slash (/) or until a serious error is encountered:

```
Program to SELECT specified rows from the Parts table -- forex2
Event List:
  CONNECT TO PartsDBE
  BEGIN WORK
  SELECT specified row from the Parts table until user enters a "/"
  COMMIT WORK
  RELEASE PartsDBE

CONNECT TO PartsDBE

Enter PartNumber from Parts table or / to STOP > 1123-P-01
BEGIN WORK
SELECT PartNumber, PartName, SalesPrice
  Part Number:  1123-P-01
  Part Name:    Central Processor
  Sales Price:      500.00
Was retrieved from the PurchDB.Parts table!
COMMIT WORK

Enter PartNumber from 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
Was retrieved from the PurchDB.Parts table!
COMMIT WORK

Enter PartNumber from Parts table or / to STOP > 1954-LP-01
BEGIN WORK
SELECT PartNumber, PartName, SalesPrice
Row not found!
COMMIT WORK

Enter PartNumber from Parts Table or / to STOP > 1823-PT-01
BEGIN WORK
SELECT PartNumber, PartName, SalesPrice
  Part Number:  1823-PT-01
  Part Name:    Graphics Printer
  Sales Price:      450.00
Was retrieved from the PurchDB.Parts table!
COMMIT WORK

Enter PartNumber from Parts table or / to STOP > /
RELEASE PartsDBE
END OF PROGRAM
```

Figure 2-5. Runtime Dialog of Program forex2

```
      PROGRAM forex2
C
C     ********************************************************
C     *  This program illustrates the use of SQL's SELECT     *
C     *  command to retrieve one row or tuple of data at      *
C     *  a time. This program executes a BEGIN WORK command   *
C     *  before the SELECT command, and a COMMIT WORK command *
C     *  after executing the SELECT command. An indicator     *
C     *  variable is also used for SalesPrice.                *
C     ********************************************************
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
           CHARACTER           Done
           CHARACTER           Abort
           INTEGER             MultipleRows
           INTEGER             Deadlock
           CHARACTER*16        Response
C
C     ***************************************************
C     *  Data Type Conversions :                         *
C     *    Character         = SQL Char(1)               *
C     *    Character*n       = SQL Char(n)               *
C     *    Character*n       = SQL VarChar               *
C     *    Double Precision  = SQL Float                 *
C     *    Double Precision  = SQL Decimal               *
C     *    Integer           = SQL Integer               *
C     *    Integer*2         = SQL SmallInt              *
C     ***************************************************
C
C             (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*16         PartNumber
      CHARACTER*30         PartName
      DOUBLE PRECISION     SalesPrice
      SQLIND               SalesPriceInd
      CHARACTER*80         SQLMessage
      EXEC SQL END DECLARE SECTION
C
C            (* End Host Variable Declarations *)
C
C
C
C
```

Figure 2-6. Program forex2

```
C
C            (* Beginning of the Main Program *)
C
      WRITE (*,*) CHAR(27), 'U'
      WRITE (*,*) 'Program to SELECT specified rows from the Parts Table
     1 -- forex2'
      WRITE (*,*) ' '
      WRITE (*,*) 'Event List:'
      WRITE (*,*) '  CONNECT TO PartsDBE'
      WRITE (*,*) '  CONNECT TO ../sampledb/PartsDBE'
      WRITE (*,*) '  BEGIN WORK'
      WRITE (*,*) '  SELECT specified row from the Parts table until use
     1r enters a "/"'
      WRITE (*,*) '  COMMIT WORK'
      WRITE (*,*) '  RELEASE PartsDBE'
C
      CALL ConnectDBE
      CALL QueryTable
      CALL ReleaseDBE
C
      STOP
      END
C
C     (* Beginning of the Sub-Routines *)
C
      SUBROUTINE ConnectDBE
C            (* Subroutine to Connect to PartsDBE *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION
C
      EXEC SQL WHENEVER SQLERROR GOTO 500
C
      WRITE (*,*) ' '
      WRITE (*,*) 'CONNECT TO PartsDBE'
      EXEC SQL CONNECT TO 'PartsDBE'
      GOTO 600
500   CALL SQLStatusCheck
      CALL EndTransaction
      CALL ReleaseDBE
C
```

Figure 2-6. Program forex2 (page 2 of 8)

```
600   RETURN
      EXEC SQL WHENEVER SQLERROR CONTINUE
      END
C     (* End of ConnectDBE Subroutine *)
C
      SUBROUTINE BeginTransaction
C            (* Subroutine to Begin Work *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION
C
      EXEC SQL WHENEVER SQLERROR GOTO 500
C
      WRITE (*,*) 'BEGIN WORK'
      EXEC SQL BEGIN WORK
      GOTO 600
500   CALL SQLStatusCheck
      CALL EndTransaction
      CALL ReleaseDBE
600   RETURN
      EXEC SQL WHENEVER SQLERROR CONTINUE
      END
C     (* End BeginTransaction Subroutine *)
C
      SUBROUTINE EndTransaction
C     (* Subroutine to Commit Work *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION
C
      EXEC SQL WHENEVER SQLERROR GOTO 500
      WRITE (*,*) 'COMMIT WORK'
```

Figure 2-6. Program forex2 (page 3 of 8)

```
      EXEC SQL COMMIT WORK
      GOTO 600
500   CALL SQLStatusCheck
      CALL ReleaseDBE
C
600   RETURN
      EXEC SQL WHENEVER SQLERROR CONTINUE
      END
C     (* End EndTransaction Subroutine *)
C
      SUBROUTINE ReleaseDBE
C     (* Subroutine to Release PartsDBE *)
C
      EXEC SQL INCLUDE SQLCA

C             (* Begin SQL Communication Area *)
C
C             (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION
C
      EXEC SQL WHENEVER SQLERROR GOTO 500
C
      WRITE (*,*) 'RELEASE PartsDBE'
      EXEC SQL RELEASE
      GOTO 600
500   CALL SQLStatusCheck
      CALL EndTransaction
C
600   RETURN
      EXEC SQL WHENEVER SQLERROR CONTINUE
      END
C     (* End ReleaseDBE Subroutine *)
C
C
C
C
C
C
C
C
C
C
C
C
```

Figure 2-6. Program forex2 (page 4 of 8)

```
C
      SUBROUTINE DisplayRow (PartNumber,PartName,SalesPrice,
     1SalesPriceInd)
C     (* Subroutine to Display a Selected Row *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*16     PartNumber
      CHARACTER*30     PartName
      DOUBLE PRECISION SalesPrice
      SQLIND           SalesPriceInd
      CHARACTER*80     SQLMessage
      EXEC SQL END DECLARE SECTION
C
      WRITE(*,100) PartNumber
      WRITE(*,110) PartName
      IF (SalesPriceInd .LT. 0) THEN
      WRITE (*,*) 'Sales Price is NULL'
      ELSE
      WRITE(*,120) SalesPrice
      ENDIF
      WRITE (*,*) 'Was retrieved from the PurchDB.Parts table!'
100   FORMAT('   Part Number:    ',A16)
110   FORMAT('   Part Name:      ',A30)
120   FORMAT('   SalesPrice:     ',F10.2)
C
      RETURN
      END
C     (* End DisplayRow Subroutine *)
C
      SUBROUTINE SQLStatusCheck
C     (* Subroutine to Check the Status of DeadLocks *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
      LOGICAL             Abort
      INTEGER             DeadLock
C
```

Figure 2-6. Program forex2 (page 5 of 8)

```
C
C
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*80        SQLMessage
      EXEC SQL END DECLARE SECTION
C
C            (* End Host Variable Declarations *)
C
      DeadLock = -14024
      Abort = .TRUE.
      WRITE (*,*) Abort
      IF (SQLCode .LT. DeadLock) THEN
 Abort = .TRUE.
      ELSE
 Abort = .FALSE.
      ENDIF
      DO WHILE (SQLCode .NE. 0)
      EXEC SQL SQLExplain :SQLMessage
      WRITE (*,*) SQLMessage
      END DO
      IF (Abort) THEN
 CALL EndTransaction
 CALL ReleaseDBE
      ENDIF
      RETURN
      END
C     (* End of SQLStatusCheck Subroutine *)
C
      SUBROUTINE QueryTable
C     (* Subroutine to Query the Parts table *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
      INTEGER             DeadLock
      INTEGER             MultipleRows
      INTEGER             NotFound
      INTEGER             OK
C
C
C
```

Figure 2-6. Program forex2 (page 6 of 8)

```
C
C             (* Begin Host Variable Declarations *)
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*16         PartNumber
      CHARACTER*30         PartName
      DOUBLE PRECISION     SalesPrice
      SQLIND               SalesPriceInd
      CHARACTER*80         SQLMessage
      EXEC SQL END DECLARE SECTION
C
C            (* End Host Variable Declarations *)
C
      MultipleRows = -10002
      DeadLock = -14024
      NotFound = 100
      OK = 0
C
      DO WHILE (PartNumber .NE. '/')
  WRITE(*,100)
100      FORMAT(/$,' Enter PartNumber from Parts table or / to STOP > ')
  READ(*,110) PartNumber
110      FORMAT (A16)
C
  IF (PartNumber .NE. '/' ) THEN
C
     CALL BeginTransaction
     WRITE(*,*) 'SELECT PartNumber, PartName, SalesPrice'
C
     EXEC SQL SELECT  PartNumber, PartName, SalesPrice
          1                 INTO :PartNumber,
          2                      :PartName,
          3                      :SalesPrice :SalesPriceInd
          4                 FROM  PurchDB.Parts
          5                WHERE  PartNumber = :PartNumber
C
     IF ((SQLWarn(3) .EQ. 'w') .OR. (SQLWarn(3) .EQ. 'W')) THEN
        WRITE (*,*) 'SQL WARNING has occurred. The following row'
        WRITE (*,*) 'of data may not be valid!'
        CALL DisplayRow (PartNumber,PartName,SalesPrice,
     1             SalesPriceInd)
     ENDIF
C
C
C
C
C
C
```

Figure 2-6. Program forex2 (page 7 of 8)

```
C
      IF (SQLCode .EQ. OK) THEN
         CALL DisplayRow (PartNumber, PartName, SalesPrice,
     1SalesPriceInd)
      ELSEIF (SQLCode .EQ. NotFound) THEN
         WRITE (*,*) 'Row not found!'
      ELSEIF (SQLCode .EQ. MultipleRows) THEN
         WRITE(*,*) 'WARNING: More than one row qualifies!'
      ELSE
         CALL SQLStatusCheck
      ENDIF
      CALL EndTransaction
   ENDIF
       END DO
       RETURN
       END
C      (* End QueryTable Subroutine *)
```

Figure 2-6. Program forex2 (page 8 of 8)

Output File Attributes

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

```
:listftemp,2

  TEMPORARY FILES FOR SOMEUSER.SOMEACCT,SOMEGRP

  ACCOUNT=  SOMEACCT    GROUP=  SOMEGRP

  FILENAME  CODE  ----------LOGICAL RECORD---------  ----SPACE----
                  SIZE  TYP       EOF     LIMIT R/B  SECTORS #X MX

  SQLMOD          250W  FB          3      1023  1       208  1 10 (TEMP)
  SQLMSG          254B  VA         14      1023  1       128  1  8 (TEMP)
  SQLOUT           80B  FA        646     10000  16      384  3 32 (TEMP)
  SQLVAR           80B  FA          8      2048  16      128  1 26 (TEMP)
```

Modified Source File

As the FORTRAN preprocessor parses the source file, it copies lines from the source file into the modified source file, comments out embedded SQL commands, and inserts information around each embedded SQL command. Figure 2-7 illustrates the modified source file generated for the source file pictured in Figure 2-6. The shaded lines contain information generated by the FORTRAN preprocessor.

In *both* preprocessing modes, the FORTRAN preprocessor:

- Inserts a C in column 1 on each line containing an embedded SQL command to comment out the SQL command for the FORTRAN compiler.
- Inserts one include FORTRAN compiler directive after the Type Declaration Section. This directive references the preprocessor generated include file (variable include file) during compilation.
- Inserts a "Start SQL Preprocessor" comment before, and an "End SQL Preprocessor" comment after code that it modifies.

In **full preprocessing mode**, the preprocessor also:

- Generates a FORTRAN COMMON BLOCK declaration of SQLCA following the EXEC SQL INCLUDE SQLCA command.
- Generates FORTRAN statements providing conditional instructions following SQL commands encountered after one of the following SQL commands: WHENEVER SQLERROR, WHENEVER SQLWARNING, and WHENEVER NOT FOUND.
- Generates FORTRAN statements that call ALLBASE/SQL external procedures at runtime. These calls reference the module stored by the preprocessor in the DBEnvironment for execution at runtime. Variables used by these external calls are defined in the variable declaration include file.
- Inserts a "Start Inserted Statements" comment before generated information.

Caution Although you can access the modified source file and the variable declaration file with an editor, you should *never* change the information generated by the FORTRAN preprocessor. Your DBEnvironment or other files on the system could be damaged at runtime if preprocessor generated statements are altered.

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

```
C**** Start SQL Preprocessor ****
$ALIAS SQLXCNHF = 'SQLXCNHF' PASCAL     \
$      (%REF,%REF,%VAL,%VAL)
$ALIAS SQLXCO   = 'SQLXCO' PASCAL       \
$      (%REF,%VAL,%REF)
$ALIAS SQLXEXIF = 'SQLXEXIF' PASCAL     \
$      (%REF,%REF,%VAL)
$ALIAS SQLXEXUF = 'SQLXEXUF' PASCAL     \
$      (%REF,%REF,%VAL,%REF,%VAL,%VAL,%REF,%VAL)
$ALIAS SQLXFE   = 'SQLXFE' PASCAL       \
$      (%REF,%REF,%REF,%VAL,%REF,%VAL,%VAL,%VAL)
$ALIAS SQLXID   = 'SQLXID' PASCAL       \
$      (%REF,%REF,%REF,%VAL,%REF,%VAL,%VAL)
$ALIAS SQLXOPKF = 'SQLXOPKF' PASCAL     \
$      (%REF,%REF,%REF,%VAL,%REF,%VAL,%VAL)
$ALIAS SQLXPLNF = 'SQLXPLNF' PASCAL     \
$      (%REF,%REF,%VAL,%VAL)
$ALIAS SQLXPREF = 'SQLXPREF' PASCAL     \
$      (%REF,%REF,%VAL,%REF,%VAL)
$ALIAS SQLXSECF = 'SQLXSECF' PASCAL     \
$      (%REF,%REF,%REF,%VAL)
$ALIAS SQLXST   = 'SQLXST' PASCAL       \
$      (%REF)
$ALIAS SQLXSVPF = 'SQLXSVPF' PASCAL     \
$      (%REF,%VAL,%REF,%REF)
C**** End SQL Preprocessor   ****
      PROGRAM forex2
C     ******************************************************
C     *  This program illustrates the use of SQL's SELECT     *
C     *  command to retrieve one row or tuple of data at      *
C     *  a time. This program executes a BEGIN WORK command   *
C     *  before the SELECT command, and a COMMIT WORK command *
C     *  after executing the SELECT command. An indicator     *
C     *  variable is also used for SalesPrice.                *
C     ******************************************************
C**** Start SQL Preprocessor ****
C     EXEC SQL INCLUDE SQLCA
C            (* Begin SQL Communication Area *)
C**** Start Inserted Statements ****
      CHARACTER SQLCAID*8
      INTEGER   SQLCABC
      INTEGER   SQLCODE
      INTEGER   SQLERRL
      CHARACTER SQLERRM*256
      CHARACTER SQLERRP*8
      INTEGER   SQLERRD(6)
      CHARACTER SQLWARN(0:7)
      INTEGER   SQLEXT(2)
```

Figure 2-7. Modified Source File for Program forex2

```
      CHARCTER SQLWARN0,SQLWARN1,SQLWARN2,SQLWARN3,
     1         SQLWARN4,SQLWARN5,SQLWARN6,SQLWARN7
      EQUIVALENCE (SQLWARN0,SQLWARN(0)),
     1            (SQLWARN1,SQLWARN(1)),
     2            (SQLWARN2,SQLWARN(2)),
     3            (SQLWARN3,SQLWARN(3)),
     4            (SQLWARN4,SQLWARN(4)),
     5            (SQLWARN5,SQLWARN(5)),
     6            (SQLWARN6,SQLWARN(6)),
     7            (SQLWARN7,SQLWARN(7))
      COMMON /SQLCA/ SQLCAID,SQLCABC,SQLCODE,SQLERRL,
     1               SQLERRM,SQLERRP,SQLERRD,SQLWARN,SQLEXT
C**** End SQL Preprocessor   ****
           CHARACTER           Done
           CHARACTER           Abort
           INTEGER             MultipleRows
           INTEGER             Deadlock
           CHARACTER*16        Response
C
C     ***************************************************
C     *  Data Type Conversions :                        *
C     *    Character         = SQL Char(1)              *
C     *    Character*n       = SQL Char(n)              *
C     *    Character*n       = SQL VarChar              *
C     *    Double Precision  = SQL Float                *
C     *    Double Precision  = SQL Decimal              *
C     *    Integer           = SQL Integer              *
C     *    Integer*2         = SQL SmallInt             *
C     ***************************************************
C             (* Begin Host Variable Declarations *)
C
C**** Start SQL Preprocessor ****
C     EXEC SQL BEGIN DECLARE SECTION
C**** End SQL Preprocessor   ****
      CHARACTER*16         PartNumber
      CHARACTER*30         PartName
      DOUBLE PRECISION     SalesPrice
      INTEGER*2 SalesPriceInd
C     SQLIND               SalesPriceInd
      CHARACTER*80         SQLMessage
C**** Start SQL Preprocessor ****
C     EXEC SQL END DECLARE SECTION
C
C            (* End Host Variable Declarations *)
C
C            (* Beginning of the Main Program *)
C
C**** End SQL Preprocessor   ****
```

Figure 2-7. Modified Source File for Program forex2 (page 2 of 13)

```
       INCLUDE 'SQLVAR'
       WRITE (*,*) CHAR(27), 'U'
       WRITE (*,*) 'Program to SELECT specified rows from the Parts Table
      1 -- forex2'
       WRITE (*,*) ' '
       WRITE (*,*) 'Event List:'
       WRITE (*,*) '  CONNECT TO PartsDBE'
       WRITE (*,*) '  BEGIN WORK'
       WRITE (*,*) '  SELECT specified row from the Parts table until use
      1r enters a "/"'
       WRITE (*,*) '  COMMIT WORK'
       WRITE (*,*) '  RELEASE PartsDBE'
C
       CALL ConnectDBE
       CALL QueryTable
       CALL ReleaseDBE
C
       STOP
       END
C
C      (* Beginning of the Sub-Routines *)
C
       SUBROUTINE ConnectDBE
C             (* Subroutine to Connect to PartsDBE *)
C
C**** Start SQL Preprocessor ****
C      EXEC SQL INCLUDE SQLCA
C
C             (* Begin SQL Communication Area *)
C             (* Begin Host Variable Declarations *)
C
C**** Start Inserted Statements ****
       CHARACTER SQLCAID*8
       INTEGER   SQLCABC
       INTEGER   SQLCODE
       INTEGER   SQLERRL
       CHARACTER SQLERRM*256
       CHARACTER SQLERRP*8
       INTEGER   SQLERRD(6)
       CHARACTER SQLWARN(0:7)
       INTEGER   SQLEXT(2)
       CHARACTER SQLWARN0,SQLWARN1,SQLWARN2,SQLWARN3,
      1          SQLWARN4,SQLWARN5,SQLWARN6,SQLWARN7
       EQUIVALENCE (SQLWARN0,SQLWARN(0)),
      1            (SQLWARN1,SQLWARN(1)),
      2            (SQLWARN2,SQLWARN(2)),
      3            (SQLWARN3,SQLWARN(3)),
```

Figure 2-7. Modified Source File for Program forex2 (page 3 of 13)

```
     4            (SQLWARN4,SQLWARN(4)),
     5            (SQLWARN5,SQLWARN(5)),
     6            (SQLWARN6,SQLWARN(6)),
     7            (SQLWARN7,SQLWARN(7))
      COMMON /SQLCA/ SQLCAID,SQLCABC,SQLCODE,SQLERRL,
     1               SQLERRM,SQLERRP,SQLERRD,SQLWARN,SQLEXT
C**** End SQL Preprocessor   ****
C**** Start SQL Preprocessor ****
C     EXEC SQL BEGIN DECLARE SECTION
C**** End SQL Preprocessor   ****
C**** Start SQL Preprocessor ****
C     EXEC SQL END DECLARE SECTION
C
C**** End SQL Preprocessor   ****
      INCLUDE 'SQLVAR'
C**** Start SQL Preprocessor ****
C     EXEC SQL WHENEVER SQLERROR GOTO 500
C
C**** Start Inserted Statements ****
C**** End SQL Preprocessor   ****
      WRITE (*,*) ' '
      WRITE (*,*) 'CONNECT TO PartsDBE'
C**** Start SQL Preprocessor ****
C     EXEC SQL CONNECT TO 'PartsDBE'
C**** Start Inserted Statements ****
      CALL SQLXCO(SQLCAID,264,'00AE00005061727473444245202020202020202
     1020202020202020202020202020202020202020202020202020202020202020202
     2020202020202020202020202020202020202020202020202020202020202020202
     3020202020202020202020202020202020202020202020202020202020202020202
     4020202020202020202020202020')
      IF (SQLCODE .LT. 0) THEN
        GO TO 500
      END IF
C**** End SQL Preprocessor   ****
      GOTO 600
500   CALL SQLStatusCheck
      CALL EndTransaction
      CALL ReleaseDBE
C
600   RETURN
C**** Start SQL Preprocessor ****
C     EXEC SQL WHENEVER SQLERROR CONTINUE
C**** Start Inserted Statements ****
C**** End SQL Preprocessor   ****
      END
C     (* End of ConnectDBE Subroutine *)
      SUBROUTINE BeginTransaction
C            (* Subroutine to Begin Work *)
```

Figure 2-7. Modified Source File for Program forex2 (page 4 of 13)

```
C**** Start SQL Preprocessor ****
C     EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
C            (* Begin Host Variable Declarations *)
C
C**** Start Inserted Statements ****
      CHARACTER SQLCAID*8
      INTEGER   SQLCABC
      INTEGER   SQLCODE
      INTEGER   SQLERRL
      CHARACTER SQLERRM*256
      CHARACTER SQLERRP*8
      INTEGER   SQLERRD(6)
      CHARACTER SQLWARN(0:7)
      INTEGER   SQLEXT(2)
      CHARACTER SQLWARN0,SQLWARN1,SQLWARN2,SQLWARN3,
     1          SQLWARN4,SQLWARN5,SQLWARN6,SQLWARN7
      EQUIVALENCE (SQLWARN0,SQLWARN(0)),
     1            (SQLWARN1,SQLWARN(1)),
     2            (SQLWARN2,SQLWARN(2)),
     3            (SQLWARN3,SQLWARN(3)),
     4            (SQLWARN4,SQLWARN(4)),
     5            (SQLWARN5,SQLWARN(5)),
     6            (SQLWARN6,SQLWARN(6)),
     7            (SQLWARN7,SQLWARN(7))
      COMMON /SQLCA/ SQLCAID,SQLCABC,SQLCODE,SQLERRL,
     1               SQLERRM,SQLERRP,SQLERRD,SQLWARN,SQLEXT
C**** End SQL Preprocessor   ****
C**** Start SQL Preprocessor ****
C     EXEC SQL BEGIN DECLARE SECTION
C**** End SQL Preprocessor   ****
C**** Start SQL Preprocessor ****
C     EXEC SQL END DECLARE SECTION
C
C**** End SQL Preprocessor   ****
      INCLUDE 'SQLVAR'
C**** Start SQL Preprocessor ****
C     EXEC SQL WHENEVER SQLERROR GOTO 500
C
C**** Start Inserted Statements ****
C**** End SQL Preprocessor   ****
      WRITE (*,*) 'BEGIN WORK'
C**** Start SQL Preprocessor ****
C     EXEC SQL BEGIN WORK
C**** Start Inserted Statements ****
      CALL SQLXCO(SQLCAID,16,'00A6007F00110061')
```

Figure 2-7. Modified Source File for Program forex2 (page 5 of 13)

```
      IF (SQLCODE .LT. 0) THEN
              GO TO 500
            END IF
      C**** End SQL Preprocessor   ****
      GOTO 600
500   CALL SQLStatusCheck
      CALL EndTransaction
      CALL ReleaseDBE
600   RETURN
C**** Start SQL Preprocessor ****
C     EXEC SQL WHENEVER SQLERROR CONTINUE
C**** Start Inserted Statements ****
C**** End SQL Preprocessor   ****
      END
C     (* End BeginTransaction Subroutine *)
C
      SUBROUTINE EndTransaction
C     (* Subroutine to Commit Work *)
C**** Start SQL Preprocessor ****
C     EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
C            (* Begin Host Variable Declarations *)
C
C**** Start Inserted Statements ****
      CHARACTER SQLCAID*8
      INTEGER   SQLCABC
      INTEGER   SQLCODE
      INTEGER   SQLERRL
      CHARACTER SQLERRM*256
      CHARACTER SQLERRP*8
      INTEGER   SQLERRD(6)
      CHARACTER SQLWARN(0:7)
      INTEGER   SQLEXT(2)
      CHARACTER SQLWARN0,SQLWARN1,SQLWARN2,SQLWARN3,
     1          SQLWARN4,SQLWARN5,SQLWARN6,SQLWARN7
      EQUIVALENCE (SQLWARN0,SQLWARN(0)),
     1            (SQLWARN1,SQLWARN(1)),
     2            (SQLWARN2,SQLWARN(2)),
     3            (SQLWARN3,SQLWARN(3)),
     4            (SQLWARN4,SQLWARN(4)),
     5            (SQLWARN5,SQLWARN(5)),
     6            (SQLWARN6,SQLWARN(6)),
     7            (SQLWARN7,SQLWARN(7))
      COMMON /SQLCA/ SQLCAID,SQLCABC,SQLCODE,SQLERRL,
     1               SQLERRM,SQLERRP,SQLERRD,SQLWARN,SQLEXT
C**** End SQL Preprocessor   ****
```

Figure 2-7. Modified Source File for Program forex2 (page 6 of 13)

```
C**** Start SQL Preprocessor ****
C     EXEC SQL BEGIN DECLARE SECTION
C**** End SQL Preprocessor   ****
C**** Start SQL Preprocessor ****
C     EXEC SQL END DECLARE SECTION
C
C**** End SQL Preprocessor   ****
      INCLUDE 'SQLVAR'
C**** Start SQL Preprocessor ****
C     EXEC SQL WHENEVER SQLERROR GOTO 500
C
C**** Start Inserted Statements ****
C**** End SQL Preprocessor   ****
      WRITE (*,*) 'COMMIT WORK'
C**** Start SQL Preprocessor ****
C     EXEC SQL COMMIT WORK
C**** Start Inserted Statements ****
      CALL SQLXCO(SQLCAID,8,'00A10000')
      IF (SQLCODE .LT. 0) THEN
        GO TO 500
      END IF
C**** End SQL Preprocessor   ****
      GOTO 600
500   CALL SQLStatusCheck
      CALL ReleaseDBE
600   RETURN
C**** Start SQL Preprocessor ****
C     EXEC SQL WHENEVER SQLERROR CONTINUE
C**** Start Inserted Statements ****
C**** End SQL Preprocessor   ****
      END
C     (* End EndTransaction Subroutine *)
C
      SUBROUTINE ReleaseDBE
C     (* Subroutine to Release PartsDBE *)
C**** Start SQL Preprocessor ****
C     EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C            (* Begin Host Variable Declarations *)
C
C**** Start Inserted Statements ****
      CHARACTER SQLCAID*8
      INTEGER   SQLCABC
      INTEGER   SQLCODE
      INTEGER   SQLERRL
      CHARACTER SQLERRM*256
      CHARACTER SQLERRP*8
```

Figure 2-7. Modified Source File for Program forex2 (page 7 of 13)

```
      INTEGER   SQLERRD(6)
      CHARACTER SQLWARN(0:7)
      INTEGER   SQLEXT(2)
      CHARACTER SQLWARN0,SQLWARN1,SQLWARN2,SQLWARN3,
     1          SQLWARN4,SQLWARN5,SQLWARN6,SQLWARN7
      EQUIVALENCE (SQLWARN0,SQLWARN(0)),
     1            (SQLWARN1,SQLWARN(1)),
     2            (SQLWARN2,SQLWARN(2)),
     3            (SQLWARN3,SQLWARN(3)),
     4            (SQLWARN4,SQLWARN(4)),
     5            (SQLWARN5,SQLWARN(5)),
     6            (SQLWARN6,SQLWARN(6)),
     7            (SQLWARN7,SQLWARN(7))
      COMMON /SQLCA/ SQLCAID,SQLCABC,SQLCODE,SQLERRL,
     1               SQLERRM,SQLERRP,SQLERRD,SQLWARN,SQLEXT
C**** End SQL Preprocessor   ****
C**** Start SQL Preprocessor ****
C     EXEC SQL BEGIN DECLARE SECTION
C**** End SQL Preprocessor   ****
C**** Start SQL Preprocessor ****
C     EXEC SQL END DECLARE SECTION
C
C**** End SQL Preprocessor   ****
      INCLUDE 'SQLVAR'
C**** Start SQL Preprocessor ****
C     EXEC SQL WHENEVER SQLERROR GOTO 500
C**** Start Inserted Statements ****
C**** End SQL Preprocessor   ****
      WRITE (*,*) 'RELEASE PartsDBE'
C**** Start SQL Preprocessor ****
C     EXEC SQL RELEASE
C**** Start Inserted Statements ****
      CALL SQLXCO(SQLCAID,56,'00B20000202020202020202020202020202020
     1202020FFFFFFFF')
      IF (SQLCODE .LT. 0) THEN
        GO TO 500
      END IF
C**** End SQL Preprocessor   ****
      GOTO 600
500   CALL SQLStatusCheck
      CALL EndTransaction
600   RETURN
C**** Start SQL Preprocessor ****
C     EXEC SQL WHENEVER SQLERROR CONTINUE
C**** Start Inserted Statements ****
C**** End SQL Preprocessor   ****
      END
C     (* End ReleaseDBE Subroutine *)
```

Figure 2-7. Modified Source File for Program forex2 (page 8 of 13)

```
      SUBROUTINE DisplayRow (PartNumber,PartName,SalesPrice,
     1SalesPriceInd)
C     (* Subroutine to Display a Selected Row *)
C
C**** Start SQL Preprocessor ****
C     EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C            (* Begin Host Variable Declarations *)
C
C**** Start Inserted Statements ****
      CHARACTER SQLCAID*8
      INTEGER   SQLCABC
      INTEGER   SQLCODE
      INTEGER   SQLERRL
      CHARACTER SQLERRM*256
      CHARACTER SQLERRP*8
      INTEGER   SQLERRD(6)
      CHARACTER SQLWARN(0:7)
      INTEGER   SQLEXT(2)
      CHARACTER SQLWARN0,SQLWARN1,SQLWARN2,SQLWARN3,
     1          SQLWARN4,SQLWARN5,SQLWARN6,SQLWARN7
      EQUIVALENCE (SQLWARN0,SQLWARN(0)),
     1            (SQLWARN1,SQLWARN(1)),
     2            (SQLWARN2,SQLWARN(2)),
     3            (SQLWARN3,SQLWARN(3)),
     4            (SQLWARN4,SQLWARN(4)),
     5            (SQLWARN5,SQLWARN(5)),
     6            (SQLWARN6,SQLWARN(6)),
     7            (SQLWARN7,SQLWARN(7))
      COMMON /SQLCA/ SQLCAID,SQLCABC,SQLCODE,SQLERRL,
     1               SQLERRM,SQLERRP,SQLERRD,SQLWARN,SQLEXT
C**** End SQL Preprocessor   ****
C**** Start SQL Preprocessor ****
C     EXEC SQL BEGIN DECLARE SECTION
C**** End SQL Preprocessor   ****
      CHARACTER*16     PartNumber
      CHARACTER*30     PartName
      DOUBLE PRECISION SalesPrice
      INTEGER*2 SalesPriceInd
C     SQLIND           SalesPriceInd
      CHARACTER*80     SQLMessage
C**** Start SQL Preprocessor ****
C     EXEC SQL END DECLARE SECTION
C
C**** End SQL Preprocessor   ****
      INCLUDE 'SQLVAR'
      WRITE(6,100) PartNumber
```

Figure 2-7. Modified Source File for Program forex2 (page 9 of 13)

```
      WRITE(6,110) PartName
      IF (SalesPriceInd .LT. 0) THEN
      WRITE (*,*) 'Sales Price is NULL'
      ELSE
      WRITE(6,120) SalesPrice
      ENDIF
      WRITE (*,*) 'Was retrieved from the PurchDB.Parts table!'
100   FORMAT('   Part Number:    ',A16)
110   FORMAT('   Part Name:      ',A30)
120   FORMAT('   SalesPrice:     ',F10.2)
C
      RETURN
      END
C     (* End DisplayRow Subroutine *)
C
      SUBROUTINE SQLStatusCheck
C     (* Subroutine to Check the Status of DeadLocks *)
C
C**** Start SQL Preprocessor ****
C     EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
C**** Start Inserted Statements ****
      CHARACTER SQLCAID*8
      INTEGER   SQLCABC
      INTEGER   SQLCODE
      INTEGER   SQLERRL
      CHARACTER SQLERRM*256
      CHARACTER SQLERRP*8
      INTEGER   SQLERRD(6)
      CHARACTER SQLWARN(0:7)
      INTEGER   SQLEXT(2)
      CHARACTER SQLWARN0,SQLWARN1,SQLWARN2,SQLWARN3,
     1          SQLWARN4,SQLWARN5,SQLWARN6,SQLWARN7
      EQUIVALENCE (SQLWARN0,SQLWARN(0)),
     1            (SQLWARN1,SQLWARN(1)),
     2            (SQLWARN2,SQLWARN(2)),
     3            (SQLWARN3,SQLWARN(3)),
     4            (SQLWARN4,SQLWARN(4)),
     5            (SQLWARN5,SQLWARN(5)),
     6            (SQLWARN6,SQLWARN(6)),
     7            (SQLWARN7,SQLWARN(7))
      COMMON /SQLCA/ SQLCAID,SQLCABC,SQLCODE,SQLERRL,
     1               SQLERRM,SQLERRP,SQLERRD,SQLWARN,SQLEXT
C**** End SQL Preprocessor   ****
      LOGICAL             Abort
      INTEGER             DeadLock
```

Figure 2-7. Modified Source File for Program forex2 (page 10 of 13)

```
C            (* Begin Host Variable Declarations *)
C**** Start SQL Preprocessor ****
C     EXEC SQL BEGIN DECLARE SECTION
C**** End SQL Preprocessor   ****
      CHARACTER*80        SQLMessage
C**** Start SQL Preprocessor ****
C     EXEC SQL END DECLARE SECTION
C            (* End Host Variable Declarations *)
C
C**** End SQL Preprocessor   ****
      INCLUDE 'SQLVAR'
      DeadLock = -14024
      Abort = .TRUE.
      WRITE (*,*) Abort
      IF (SQLCode .LT. DeadLock) THEN
        Abort = .TRUE.
      ELSE
        Abort = .FALSE.
      ENDIF
      DO WHILE (SQLCode .NE. 0)
C**** Start SQL Preprocessor ****
C     EXEC SQL SQLExplain :SQLMessage
C**** Start Inserted Statements ****
       CALL SQLXPLNF(SQLCAID,SQLTMP,80,0)
        READ(SQLTMP,'(A80)')SQLMessage
C**** End SQL Preprocessor   ****
      WRITE (*,*) SQLMessage
      END DO
      IF (Abort) THEN
        CALL EndTransaction
        CALL ReleaseDBE
      ENDIF
      RETURN
      END
C     (* End of SQLStatusCheck Subroutine *)
C
      SUBROUTINE QueryTable
C     (* Subroutine to Query the Parts table *)
C
C**** Start SQL Preprocessor ****
C     EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
C**** Start Inserted Statements ****
      CHARACTER SQLCAID*8
      INTEGER   SQLCABC
      INTEGER   SQLCODE
```

Figure 2-7. Modified Source File for Program forex2 (page 11 of 13)

```
       INTEGER   SQLERRL
       CHARACTER SQLERRM*256
       CHARACTER SQLERRP*8
       INTEGER   SQLERRD(6)
       CHARACTER SQLWARN(0:7)
       INTEGER   SQLEXT(2)
       CHARACTER SQLWARN0,SQLWARN1,SQLWARN2,SQLWARN3,
      1          SQLWARN4,SQLWARN5,SQLWARN6,SQLWARN7
       EQUIVALENCE (SQLWARN0,SQLWARN(0)),
      1            (SQLWARN1,SQLWARN(1)),
      2            (SQLWARN2,SQLWARN(2)),
      3            (SQLWARN3,SQLWARN(3)),
      4            (SQLWARN4,SQLWARN(4)),
      5            (SQLWARN5,SQLWARN(5)),
      6            (SQLWARN6,SQLWARN(6)),
      7            (SQLWARN7,SQLWARN(7))
       COMMON /SQLCA/ SQLCAID,SQLCABC,SQLCODE,SQLERRL,
      1               SQLERRM,SQLERRP,SQLERRD,SQLWARN,SQLEXT
C**** End SQL Preprocessor   ****
      INTEGER              DeadLock
      INTEGER              MultipleRows
      INTEGER              NotFound
      INTEGER              OK
C             (* Begin Host Variable Declarations *)
C**** Start SQL Preprocessor ****
C     EXEC SQL BEGIN DECLARE SECTION
C**** End SQL Preprocessor   ****
      CHARACTER*16         PartNumber
      CHARACTER*30         PartName
      DOUBLE PRECISION     SalesPrice
       INTEGER*2 SalesPriceInd
C      SQLIND               SalesPriceInd
      CHARACTER*80         SQLMessage
C**** Start SQL Preprocessor ****
C     EXEC SQL END DECLARE SECTION
C             (* End Host Variable Declarations *)
C
C**** End SQL Preprocessor   ****
       INCLUDE 'SQLVAR'
      MultipleRows = -10002
      DeadLock = -14024
      NotFound = 100
      OK = 0
      DO WHILE (PartNumber .NE. '/')
        WRITE(6,100)
100      FORMAT(/$,' Enter PartNumber from Parts table or / to STOP > ')
         READ(5,110) PartNumber
```

Figure 2-7. Modified Source File for Program forex2 (page 12 of 13)

```
110      FORMAT (A16)
C
         IF (PartNumber .NE. '/' ) THEN
C
            CALL BeginTransaction
            WRITE(*,*) 'SELECT PartNumber, PartName, SalesPrice'
C
C**** Start SQL Preprocessor ****
C           EXEC SQL SELECT  PartNumber, PartName, SalesPrice
C    1                 INTO :PartNumber,
C    2                      :PartName,
C    3                      :SalesPrice :SalesPriceInd
C    4                 FROM  PurchDB.Parts
C    5                WHERE  PartNumber = :PartNumber
C
C**** Start Inserted Statements ****
      WRITE(SQLTMP,'(A16)')PartNumber
      CALL SQLXFE(SQLCAID,SQLOWN,SQLMDN,1,SQLTMP,16,56,1)
      IF (SQLCODE .EQ. 0) THEN
      READ(SQLTMP,'(A16,A30,A8,A2)')PartNumber,PartName,SalesPrice,Sales
     1PriceInd
      ELSE
      END IF
C**** End SQL Preprocessor   ****
            IF ((SQLWarn(3) .EQ. 'w') .OR. (SQLWarn(3) .EQ. 'W')) THEN
               WRITE (*,*) 'SQL WARNING has occured. The following row'
               WRITE (*,*) 'of data may not be valid!'
               CALL DisplayRow (PartNumber,PartName,SalesPrice,
     1            SalesPriceInd)
            ENDIF
C
            IF (SQLCode .EQ. OK) THEN
               CALL DisplayRow (PartNumber, PartName, SalesPrice,
     1           SalesPriceInd)
            ELSEIF (SQLCode .EQ. NotFound) THEN
               WRITE (*,*) 'Row not found!'
            ELSEIF (SQLCode .EQ. MultipleRows) THEN
               WRITE(*,*) 'WARNING: More than one row qualifies!'
            ELSE
               CALL SQLStatusCheck
            ENDIF
            CALL EndTransaction
         ENDIF
      END DO
      RETURN
      END
C     (* End QueryTable Subroutine *)
```

Figure 2-7. Modified Source File for Program forex2 (page 13 of 13)

Variable Declaration Include File

The preprocessor generated include file (SQLVAR), contains declarations for variables referenced in preprocessor generated statements in the modified source file. Figure 2-8 illustrates the variable declaration include file that corresponds to the modified source file in Figure 2-7. Note in Figure 2-7 that just after inserting the EXEC SQL END DECLARE SECTION declaration into the modified source file, the preprocessor inserted the following FORTRAN compiler directive to reference the variable declaration include file:

```
$INCLUDE 'SQLVAR'
```

This directive is always inserted after the Host Variable Type Declaration Section.

When you use file equations to redirect the include files, remember that the preprocessor always inserts the same $INCLUDE directive. Therefore, insure that the applicable file equations are in effect when you preprocess and when you compile. When the preprocessor is invoked, the following file equation must be in effect.

```
:FILE SQLVAR = MYVAR
```

Then when the FORTRAN compiler is invoked, the following file equation must be in effect:

```
:FILE SQLVAR = MYVAR
:FTNC MYSQLPRG, $NEWPASS, $NULL
```

```
C  temporary area
      CHARACTER*112 SQLTMP
C  ownership information
      CHARACTER*20 SQLOWN
      CHARACTER*20 SQLMDN
      DATA SQLOWN /'JOANN@HPSQL         '/
      DATA SQLMDN /'FOREX2              '/
C
```

Figure 2-8. Sample Variable Declaration Include File

ALLBASE/SQL Message File

Messages placed in SQLMSG come from the ALLBASE/SQL message catalog. The default catalog is SQLCTxxx.PUB.SYS. For native language users, the name of the catalog is SQLCT000.PUB.SYS, where NATIVE-3000 is the message catalog.

If the default catalog cannot be opened, ALLBASE/SQL returns an error message indicating that the catalog file is not available. If the native language catalog is available, ALLBASE/SQL returns a warning message, indicating that the default catalog is being used. SQLMSG messages come in four four parts:

1. A banner:

```
        MON, JUL 10, 1991,  4:48 PM
HP36216-02A.E1.16         FORTRAN Preprocessor/3000        ALLBASE/SQL
(C) COPYRIGHT HEWLETT-PACKARD CO.  1982,1983,1984,1985,1986,1987,1988,
1989,1990,1991.   ALL RIGHTS RESERVED
```

   Banners are displayed when ISQL, SQLUtil, or a preprocessor is invoked.

2. A summary of the preprocessor invocation conditions:

```
SQLIN               = FOREX2.SOMEGROUP.SOMEACCT
DBEnvironment       = PartsDBE
Module Name         = FOREX2
```

3. Warnings and errors encountered during preprocessing:

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

****** ALLBASE/SQL errors  (DBERR 10952)
****** in SQL statement ending in line 290
*** Selectlist has 3 items and host variable buffer has 2.  (DBERR 2762)

There are errors.  No sections stored.
```

4. A summary of the results of preprocessing:

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

Both the banner and the preprocessing summary output are echoed to the standard output, the terminal.

As illustrated in Figure 2-9, a line number is often provided in SQLMSG. This line number references the line in the modified source file containing the command in question. A message accompanied by a number may also appear. You can refer to the *ALLBASE/SQL Message Manual* for additional information on the exception condition when these numbered messages appear.

```
:EDITOR
HP32201A.07.20  EDIT/3000  MON, JUL 10, 1990, 4:49 PM
(C) HEWLETT-PACKARD CO. 1990
/T SQLMSG; L ALL UNN
FILE UNNUMBERED

SQLIN                = FOREX2.SOMEGROUP.SOMEACCT
DBEnvironment        = PartsDBE
Module Name          = FOREX2


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

******  ALLBASE/SQL errors (DBERR 10952)
******  in SQL statement ending in line 290
*** Selectlist has 3 items and host variable buffer has 2.  (DBERR 2762)

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

Figure 2-9. Sample SQLMSG Showing Error

As Figure 2-10 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 runtime if it cannot be validated.

```
:EDITOR
HP32201A.07.20  EDIT/3000  MON, JUL 10, 1991, 4:49 PM
(C) HEWLETT-PACKARD CO. 1990
/T SQLMSG; L ALL UNN
FILE UNNUMBERED

       .
       .
       .

SQLIN                = FOREX2.SOMEGROUP.SOMEACCT
DBEnvironment        = PartsDBE
Module Name          = FOREX2

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

****** ALLBASE/SQL warnings. (DBWARN 10602)
****** in SQL statement ending in line 290
*** Column PARNUMBER not found. (DBERR 2211)

  1 Sections stored in DBEnvironment.

 0 ERRORS   1 WARNINGS
END OF PREPROCESSING
```

Figure 2-10. Sample SQLMSG Showing Warning

Installable Module File

When the FORTRAN preprocessor stores a module in the system catalog of a DBEnvironment at preprocessing time, it places a copy of the module in an installable module file. The name of this file is SQLMOD. 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. For example:

```
:RUN PSQLFOR.PUB.SYS;INFO = "DBEnvironmentName&
(MODULE (InstalledModuleName) DROP)"
```

If you want to preserve the SQLMOD file after preprocessing, you must keep it as a permanent file. Rename SQLMOD after making it permanent.

```
:SAVE SQLMOD
:RENAME SQLMOD, MYMOD
```

Before invoking ISQL to install this module file, you may have to transport it and its related program file to the machine containing the target DBEnvironment. After all the files are restored on the target machine, you invoke ISQL on the machine containing the target DBEnvironment.

```
: isql
```

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

```
isql=> CONNECT TO 'PartsDBE.SOMEGROUP.SOMEACCT';
isql=> INSTALL;

File name> MYMOD.SOMEGROUP.SOMEACCT;
Name of module in this file:  JOANN@SOMEACCT.FOREX2
Number of sections installed:  1
COMMIT WORK to save to DBEnvironment.

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

Stored Sections

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

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

The commands listed above either require no authorization to execute or are executed based on information contained in the compilable preprocessor output files.

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 an uncompiled command. The **run tree** is the compiled, executable form of the command.

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

There are three types of sections:

- Sections for executing the SELECT command associated with a DECLARE CURSOR command.
- Sections for executing the SELECT command associated with a CREATE VIEW command.
- Sections for all other commands for which the preprocessor stores a section.

Figure 2-11 illustrates the kind of information in the system catalog that describes each type of stored section. The query result illustrated was extracted from the system view named *SYSTEM.SECTION* by using ISQL. The columns in Figure 2-11 have the following meanings:

- NAME: This column contains the name of the module to which a section belongs. You specify a module name when you invoke the preprocessor; the module name is by default the program name from the PROGRAM Statement. If you are supplying a module name in a language other than NATIVE-3000 (ASCII), be sure it is in the same language as that of the DBEnvironment.
- OWNER: This column identifies the owner of the module. You specify an owner name when you invoke the preprocessor; the owner name is by default the userid associated with the preprocessing session. If you are supplying an owner name in a native language other than NATIVE-3000 (ASCII), be sure it is in the same language as that of the DBEnvironment.

- DBEFILESET: This column indicates the DBEFileSet with which DBEFiles housing the section are associated.
- SECTION: This column gives the section number. Each section associated with a module is assigned a number by the preprocessor as it parses the related SQL command at preprocessing time.
- TYPE: This column identifies the type of section:

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

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

  0 = invalid
  1 = valid

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

---------------------------------------------------------------------------
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
FOREX2              |JOANN@ACCT  |SYSTEM            |        1 |     0|     1
FOREX7              |BILL@SOMEACT|SYSTEM            |        1 |     1|     1
FOREX7              |BILL@SOMEACT|SYSTEM            |        2 |     0|     1
---------------------------------------------------------------------------
Number of rows selected is 16.
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>,or e[nd]>
---------------------------------------------------------------------------
```

Figure 2-11. Information in SYSTEM.SECTION on Stored Sections

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

The remaining rows describe sections associated with two preprocessed programs. FOREX2 contains only one section, for executing the SELECT command in the program illustrated in Figure 2-5. Another program may contain two sections, one for executing the SELECT command associated with a DECLARE CURSOR command and one for executing a FETCH command.

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

```
   isql=> DROP MODULE FOREX2;

          or

   : RUN PSQLFOR.PUB.SYS;INFO = "PartsDBE (MODULE (FOREX2) DROP)
```

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 occur that affect the execution of embedded commands.

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

Invoking The Fortran Preprocessor

The FORTRAN preprocessor can be invoked in either

- Syntax checking mode, or
- Full preprocessing mode

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

Syntax Checking Mode

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

Syntax

```
:RUN PSQLFOR.PUB.SYS;INFO="(SYNTAX)"
```

1. The preprocessor does not access a DBEnvironment when it is run in this mode.
2. When performing only syntax checking, the preprocessor does not convert the SQL commands into FORTRAN constructs. Therefore the modified source code file does not contain any preprocessor generated calls to ALLBASE/SQL external procedures.
3. The include and installable module files are created, but incomplete.

Authorization

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

Example

```
:FILE SQLIN = FOREX2
:RUN PSQLFOR.PUB.SYS; INFO= "(SYNTAX)"

           MON, JUL 10, 1989,  4:48 PM
HP36216-02A.E1.16        FORTRAN Preprocessor/3000         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.

PROGRAM TERMINATED IN AN ERROR STATE.  (CIERR 976)

:EDITOR
HP32201A.07.20  EDIT/3000  TUE, JUN 21, 1991, 2:00 PM
(C) HEWLETT-PACKARD CO. 1990
/T SQLMSG; L ALL UNN
FILE UNNUMBERED
:
SQLIN          = FOREX2.SOMEGROUP.SOMEACCT

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

****** ALLBASE/SQL errors.  (DBERR 10977)
****** in SQL statement ending in line 290
***    Syntax error.  (DBERR 1001)

Syntax checked.

 1 ERRORS   0 WARNINGS
END OF PREPROCESSING
```

The line 290 referenced in SQLMSG is the line in the source file where the erroneous SQL command ends.

Full Preprocessing Mode

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

Syntax

```
:RUN PSQLFOR.PUB.SYS;

                          ┌   ┌ MODULE(ModuleName)                ┐        ┐
                          │   │ OWNER (OwnerName )                │        │
INFO="DBEnvironmentName   │ ( ┤ ┌      ┌ PRESERVE ┐ ┐             ├ | ... |)│ "
                          │   │ ┤ DROP ┤ REVOKE   ┘ ├             │        │
                          └   └ └ NODROP            ┘             ┘        ┘
```

Parameters

DBEnvironmentName	identifies the DBEnvironment in which a module is to be stored. You may use a backreference for a file defined in a file for this parameter.
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, the preprocessor uses the PROGRAM statement name as the module name.
OwnerName	Associates the stored module with a user's log-on name, 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 to be stored. You can also specify a group as owner if you are a member of the group. If not specified, the owner name is your log-on name (*USER@ACCOUNT*). Any object names in the source file not qualified with an owner name are qualified with this *OwnerName*.
DROP	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.
PRESERVE	Is specified when the program being preprocessed already has a stored module and you want to preserve existing RUN authorities for that module. If not specified, PRESERVE is assumed. PRESERVE cannot be specified unless DROP is also specified.
REVOKE	Is specified when the program being preprocessed already has a stored module and you want to revoke existing RUN authorities for that module. REVOKE cannot be specified unless DROP is also specified.
NODROP	Terminates preprocessing if any module currently exists in the DBEnvironment with the *ModuleName* and *OwnerName* specified in the INFO string. If not specified, NODROP is assumed.

Description

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

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

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

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

5. During preprocessing, system catalog pages accessed for embedded commands are locked. In multiuser mode, other DBE sessions accessing the same objects must wait, and the potential for a deadlock exists. Therefore minimize competing transactions when preprocessing an application program. Refer to the *ALLBASE/SQL Database Administration Guide* for information on operations that lock system catalog pages.

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

7. If you specify an OwnerName or ModuleName in a language other than NATIVE-3000 (ASCII), be sure that the language you are using is also the language of the DBEnvironment in which the module will be stored.

Authorization

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

Example

```
:FILE SQLIN=FOREX2
:RUN PSQLFOR.PUB.SYS;INFO=&
"PartsDBE (MODULE(FOREX2) OWNER(OwnerP@SomeAcct) REVOKE DROP)"

        MON, JUL 10, 1991,  4:48 PM
HP36216-02A.E1.16        FORTRAN Preprocessor/3000         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.

END OF PROGRAM
:EDITOR
HP32201A.07.20  EDIT/3000  TUE, JUN 21, 1991, 2:00 PM
(C) HEWLETT-PACKARD CO. 1990
/T SQLMSG; L ALL UNN
FILE UNNUMBERED

       MON, JUL 10, 1989,  5:00 PM
      .
      .
      .
SQLIN                = FOREX2.SOMEGROUP.SOMEACCT
DBEnvironment        = PartsDBE
Module Name          = FOREX2

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

  0 ERRORS   1 WARNINGS
END OF PROCESSING

/
```

Using the Preprocessor UDC's

Two UDC's for invoking the FORTRAN preprocessor are provided with ALLBASE/SQL in the HPSQLUDC.PUB.SYS file:

- **PFOR**, illustrated in Figure 2-12, invokes the preprocessor in full preprocessing mode. You specify the source file name, a DBEnvironment name, and a name for SQLMSG (if you do not want preprocessor messages to go to $STDLIST).

  :PFOR *SourceFileName,DBEnvironment*

  The PFOR UDC uses the following preprocessor INFO string parameters:

  ModuleName is the name of the source file.

  OwnerName is the log-on *User@Account*.

  PRESERVE and DROP are in effect.

- **PPFOR**, illustrated in Figure 2-13, invokes the preprocessor in full preprocessing mode, then invokes the FORTRAN compiler if preprocessing is successful and the linker if compilation is successful.

  To use this UDC, you specify the source file name, a DBEnvironment name, and an executable file name. You can specify a name for SQLMSG if you do not want preprocessor messages to go to $STDLIST:

  :PPFOR *SourceFileName,DBEnvironment,ExecutableFileName*

  This UDC uses the following preprocessor INFO string parameters:

  ModuleName is the source file name.

  OwnerName is the log-on *User@Account*.

  PRESERVE and DROP are in effect.

  If you make your own version of the UDC's, do not modify the record attributes for any of the preprocessor output files. Only modify the file limit (`disc=`*FileLimit*) if required.

Note Because the UDC's purge the preprocessor message file, if messages are sent to $STDLIST an error message appears when you use the UDC's, but preprocessing continues.

```
PFOR srcfile,dbefile,msgfile=$stdlist
continue
setvar _savefence hpmsgfence
setvar hpmsgfence 2
continue
purge !msgfile
purge sqlout
purge sqlmod
purge sqlvar
setvar hpmsgfence _savefence
deletevar _savefence
file sqlin  = !srcfile
file sqlmsg = !msgfile; rec=-80,16,f,ascii
file sqlout;   disc=10000,32;  rec=-80,16,f,ascii
file sqlmod;   disc=1023,10,1; rec=250,,f,binary
file sqlvar;   disc=2048,32;   rec=-80,16,f,ascii
continue
run psqlfor.pub.sys;info="!dbefile (drop)"
reset sqlin
reset sqlmsg
reset sqlout
reset sqlmod
reset sqlvar
```

Figure 2-12. UDC for Preprocessing SQLIN

```
PPFOR srcfile,dbefile,pgmfile,msgfile=$stdlist
continue
setvar _savefence hpmsgfence
setvar hpmsgfence 2
continue
purge !msgfile
purge sqlout
purge sqlmod
purge sqlvar
setvar hpmsgfence _savefence
deletevar _savefence
file sqlin  = !srcfile
file sqlmsg = !msgfile; rec=-80,16,f,ascii
file sqlout;   disc=10000,32;  rec=-80,16,f,ascii
file sqlmod;   disc=1023,10,1; rec=250,,f,binary
file sqlvar;   disc=2048,32;   rec=-80,16,f,ascii
continue
run psqlfor.pub.sys;info="!dbefile (drop)"
if jcw <= warn then
  continue
  ftnxllk sqlout,!pgmfile,$null
endif
reset sqlin
reset sqlmsg
reset sqlout
reset sqlmod
reset sqlvar
```

Figure 2-13. UDC for Preprocessing, Compiling, and Preparing SQLIN

The example in Figure 2-14 illustrates the use of PPFOR on an SQLIN that could be successfully preprocessed, compiled, and linked.

```
:PPFOR FOREX2,PARTSDBE,FOREX2R

           MON, JUL 10, 1989,  3:43
HP36216-02A.03.01  FORTRAN Preprocessor/3000               ALLBASE/SQL
(C) COPYRIGHT HEWLETT-PACKARD CO., 1982,1983,1984,1985,1986,1987,1988,
1989,1990,1991.  ALL RIGHTS RESERVED

 SQLIN               = FOREX2.SOMEGRP.SOMEACCT
 DBEnvironment       = partsdbe

 Module Name         = FOREX2
1 Sections stored in DBEnvironment.

 0 ERRORS     0 WARNINGS
END OF PREPROCESSING.

END OF PROGRAM

⋮

END OF COMPILE

HP Link Editor/XL (HP30315A.04.04) Copyright Hewlett-Packard Co 1986

LinkEd> LINK FROM=$OLDPASS;TO=FOREX2R

END OF PROGRAM
:
```

Figure 2-14. Sample UDC Invocation

If there are compiler errors or warnings, the line number referenced in the compiler output messages is the FORTRAN statement number in the *compiler output listing*. Remember that PPFOR UDC sends the compiler output listing to $null. Thus to identify the line in error, you must reinvoke the compiler, sending the compiler listing to an output file:

```
:BUILD FORLIST;DISC=10000,32;REC=-80,16,F,ASCII
:FTNXL SQLOUT,$OLDPASS,FORLIST
```

Running the Preprocessor in Job Mode

You can preprocess FORTRAN programs in job mode. Figure 2-15 illustrates a job file that uses the PPFOR UDC to preprocess several sample programs.

```
!JOB JOANN,MGR.HPDB,FORTRAN;OUTCLASS=,1
!ppfor forp01,PartsDBE,forp01r
!ppfor forp01a,PartsDBE,forp01ar
!ppfor forp02,PartsDBE,forp02r
     .
     .
!ppfor for50,PartsDBE,for50r
!TELL JOANN,MGR.HPDB; FORTRAN Preprocessing is complete!
!EOJ
```

Figure 2-15. Sample Preprocessing Job File

Preprocessing Errors

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

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

Preprocessor or DBEnvironment Termination

Whenever the FORTRAN 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:

```
Input source file not found.  (DBERR 10921)

 1 ERRORS  0 WARNINGS
END OF PREPROCESSING.
```

If the invocation line names a DBEnvironment that does not exist or contains erroneous syntax, preprocessing terminates as follows:

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

Source File Errors

When the FORTRAN preprocessor encounters errors while parsing the source file, messages are placed in SQLMSG. Refer to the discussion earlier in this chapter under "SQL Message File" for additional information on this category of errors.

DBEnvironment Errors

Some errors can be caused because:

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

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

3

Embedding SQL Commands

In every FORTRAN program, you embed SQL commands in the main program unit and/or in a subprogram unit in order to:

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

The program listing shown in Figure 3-1 illustrates where in a program you can embed SQL commands to accomplish the activities listed above.

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

```
      PROGRAM forex2
C
C     ******************************************************
C     *  This program illustrates the use of SQL's SELECT     *
C     *  command to retrieve one row or tuple of data at      *
C     *  a time. This program executes a BEGIN WORK command   *
C     *  before the SELECT command, and a COMMIT WORK command *
C     *  after executing the SELECT command. An indicator     *
C     *  variable is also used for SalesPrice.                *
C     ******************************************************
C                                                                1
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
           CHARACTER           Done
           CHARACTER           Abort
           INTEGER             MultipleRows
           INTEGER             Deadlock
           CHARACTER*16        Response
C
C     ***************************************************
C     *  Data Type Conversions :                        *
C     *    Character          = SQL Char(1)             *
C     *    Character*n        = SQL Char(n)             *
C     *    Character*n        = SQL VarChar             *
C     *    Double Precision   = SQL Float               *
C     *    Double Precision   = SQL Decimal             *
C     *    Integer            = SQL Integer             *
C     *    Integer*2          = SQL SmallInt            *
C     ***************************************************
C
C             (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION                             2
      CHARACTER*16          PartNumber
      CHARACTER*30          PartName
      DOUBLE PRECISION      SalesPrice
      SQLIND                SalesPriceInd
      CHARACTER*80          SQLMessage
      EXEC SQL END DECLARE SECTION                               2
C
C            (* End Host Variable Declarations *)
C
C
C
C
```

Figure 3-1. Sample Source File

```
C
C            (* Beginning of the Main Program *)
C
      WRITE (*,*) CHAR(27), 'U'
      WRITE (*,*) 'Program to SELECT specified rows from the
     1Parts Table 1 -- forex2'
      WRITE (*,*) ' '
      WRITE (*,*) 'Event List:'
      WRITE (*,*) '  CONNECT TO PartsDBE'
      WRITE (*,*) '  BEGIN WORK'
      WRITE (*,*) '  SELECT specified row from the Parts
     1table until use 1r enters a "/"'
      WRITE (*,*) '  COMMIT WORK'
      WRITE (*,*) '  RELEASE PartsDBE'
C
      CALL ConnectDBE
      CALL QueryTable
      CALL ReleaseDBE
C
      STOP
      END
C
C     (* Beginning of the Sub-Routines *)
C
      SUBROUTINE ConnectDBE
C            (* Subroutine to Connect to PartsDBE *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION
C
      EXEC SQL WHENEVER SQLERROR GOTO 500
C
      WRITE (*,*) ' '
      WRITE (*,*) 'CONNECT TO PartsDBE'
      EXEC SQL CONNECT TO 'PartsDBE'                          3
      GOTO 600
500   CALL SQLStatusCheck
      CALL EndTransaction
      CALL ReleaseDBE
C
C
C
```

Figure 3-1. Sample Source File (page 2 of 8)

```
600   RETURN
      EXEC SQL WHENEVER SQLERROR CONTINUE
      END
C     (* End of ConnectDBE Subroutine *)
C
      SUBROUTINE BeginTransaction
C            (* Subroutine to Begin Work *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION
C
      EXEC SQL WHENEVER SQLERROR GOTO 500
C
      WRITE (*,*) 'BEGIN WORK'
      EXEC SQL BEGIN WORK                            4
      GOTO 600
500   CALL SQLStatusCheck
      CALL EndTransaction
      CALL ReleaseDBE
C
600   RETURN
      EXEC SQL WHENEVER SQLERROR CONTINUE
      END
C     (* End BeginTransaction Subroutine *)
C
      SUBROUTINE EndTransaction
C     (* Subroutine to Commit Work *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION
C
      EXEC SQL WHENEVER SQLERROR GOTO 500
C
      WRITE (*,*) 'COMMIT WORK'
C
C
```

Figure 3-1. Sample Source File (page 3 of 8)

```
      EXEC SQL COMMIT WORK                         5
      GOTO 600
500   CALL SQLStatusCheck
      CALL ReleaseDBE
C
600   RETURN
      EXEC SQL WHENEVER SQLERROR CONTINUE
      END
C     (* End EndTransaction Subroutine *)
C
      SUBROUTINE ReleaseDBE
C     (* Subroutine to Release PartsDBE *)
C
      EXEC SQL INCLUDE SQLCA

C            (* Begin SQL Communication Area *)
C
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION
C
      EXEC SQL WHENEVER SQLERROR GOTO 500          6
C
      WRITE (*,*) 'RELEASE PartsDBE'
      EXEC SQL RELEASE                             7
      GOTO 600
500   CALL SQLStatusCheck
      CALL EndTransaction
C
600   RETURN
      EXEC SQL WHENEVER SQLERROR CONTINUE          6
      END
C     (* End ReleaseDBE Subroutine *)
C
C
C
C
C
C
C
C
C
C
C
C
C
C
```

Figure 3-1. Sample Source File (page 4 of 8)

```
C
C
      SUBROUTINE DisplayRow (PartNumber,PartName,SalesPrice,
     1SalesPriceInd)
C     (* Subroutine to Display a Selected Row *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*16     PartNumber
      CHARACTER*30     PartName
      DOUBLE PRECISION SalesPrice
      SQLIND           SalesPriceInd
      CHARACTER*80     SQLMessage
      EXEC SQL END DECLARE SECTION
C
      WRITE(*,100) PartNumber
      WRITE(*,110) PartName
      IF (SalesPriceInd .LT. 0) THEN
      WRITE (*,*) 'Sales Price is NULL'
      ELSE
      WRITE(*,120) SalesPrice
      ENDIF
      WRITE (*,*) 'Was retrieved from the PurchDB.Parts table!'
100   FORMAT('   Part Number:     ',A16)
110   FORMAT('   Part Name:       ',A30)
120   FORMAT('   SalesPrice:      ',F10.2)
C
      RETURN
      END
C     (* End DisplayRow Subroutine *)
C
      SUBROUTINE SQLStatusCheck
C     (* Subroutine to Check the Status of DeadLocks *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
      LOGICAL            Abort
      INTEGER            DeadLock
C
C
C
```

Figure 3-1. Sample Source File (page 5 of 8)

```
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*80        SQLMessage
      EXEC SQL END DECLARE SECTION
C
C            (* End Host Variable Declarations *)
C
      DeadLock = -14024
      Abort = .TRUE.
      WRITE (*,*) Abort
      IF (SQLCode .LT. DeadLock) THEN                  9
        Abort = .TRUE.
      ELSE
        Abort = .FALSE.
      ENDIF
      DO WHILE (SQLCode .NE. 0)
      EXEC SQL SQLExplain :SQLMessage                  10
      WRITE (*,*) SQLMessage
      END DO
      IF (Abort) THEN
        CALL EndTransaction
        CALL ReleaseDBE
      ENDIF
      RETURN
      END
C     (* End of SQLStatusCheck Subroutine *)
C
      SUBROUTINE QueryTable
C     (* Subroutine to Query the Parts table *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
      INTEGER              DeadLock
      INTEGER              MultipleRows
      INTEGER              NotFound
      INTEGER              OK
C
```

Figure 3-1. Sample Source File (page 6 of 8)

```
C            (* Begin Host Variable Declarations *)
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*16        PartNumber
      CHARACTER*30        PartName
      DOUBLE PRECISION    SalesPrice
      SQLIND              SalesPriceInd
      CHARACTER*80        SQLMessage
      EXEC SQL END DECLARE SECTION
C
C           (* End Host Variable Declarations *)
C
      MultipleRows = -10002
      DeadLock = -14024
      NotFound = 100
      OK = 0
C
      DO WHILE (PartNumber .NE. '/')
         WRITE(*,100)
100      FORMAT(/$,' Enter PartNumber from Parts table
     1or / to STOP > ')
         READ(*,110) PartNumber
110      FORMAT (A16)
C
         IF (PartNumber .NE. '/' ) THEN
C
     CALL BeginTransaction
     WRITE(*,*) 'SELECT PartNumber, PartName, SalesPrice'
C
         EXEC SQL SELECT  PartNumber, PartName, SalesPrice          8
     1                INTO :PartNumber,
     2                     :PartName,
     3                     :SalesPrice :SalesPriceInd
     4                FROM  PurchDB.Parts
     5               WHERE  PartNumber = :PartNumber
C
         IF ((SQLWarn(3) .EQ. 'w') .OR. (SQLWarn(3) .EQ. 'W')) THEN
           WRITE (*,*) 'SQL WARNING has occured. The following row'
           WRITE (*,*) 'of data may not be valid!'
           CALL DisplayRow (PartNumber,PartName,SalesPrice,
     1         SalesPriceInd)
         ENDIF
C
C
C
C
C
C
C
```

Figure 3-1. Sample Source File (page 7 of 8)

```
      IF (SQLCode .EQ. OK) THEN                                  9
         CALL DisplayRow (PartNumber, PartName, SalesPrice)
      ELSEIF (SQLCode .EQ. NotFound) THEN
         WRITE (*,*) 'Row not found!'
      ELSEIF (SQLCode .EQ. MultipleRows) THEN
         WRITE(*,*) 'WARNING: More than one row qualifies!'
      ELSE
         CALL SQLStatusCheck
      ENDIF
      CALL EndTransaction
          ENDIF
       END DO
       RETURN
       END
C      (* End QueryTable Subroutine *)
```

Figure 3-1. Sample Source File (page 8 of 8)

General Rules for Embedding SQL

Embedded SQL commands must appear in certain locations within the FORTRAN program. Each embedded SQL command must be accompanied by the prefix EXEC SQL. Comments may be placed within an embedded command, and an embedded SQL command may continue for several lines.

An embedded SQL command has no *maximum length*. A dynamic SQL command can be no longer than 2048 bytes.

Location of SQL Commands

Put SQL commands, including their prefix, within columns 7 through 72 of either the main program unit or a subprogram unit:

- INCLUDE SQLCA must appear in the Type Declaration Section of the program unit which contains embedded SQL statements. The INCLUDE command must appear before the BEGIN DECLARE SECTION command.
- BEGIN DECLARE SECTION and END DECLARE SECTION must appear as the last declaration in the Type Declaration Section of the program unit which contains embedded SQL statements. No variable type declarations are allowed after the END DECLARE SECTION command.
- All other SQL commands may appear after the END DECLARE SECTION in either the main program unit or a subprogram unit.

Prefix

Precede each SQL command with the prefix EXEC SQL. Minimally the prefix must be specified *on the same line*. For example, the following are legal:

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

      EXEC SQL
     1           SELECT PartName
     2           INTO  :PartName
     3           FROM   PurchDB.Parts
     4           WHERE  PartNumber = :PartNumber
```

However, the following is not legal:

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

FORTRAN Comments

You may insert FORTRAN comment lines within or between embedded SQL commands. Denote comment lines by placing the letter C in column 1 and entering the comment in columns 7 through 72:

```
      EXEC SQL SELECT PartNumber, PartName
C     put the data into the following host variables
     1  INTO :PartNumber, :PartName
C     find the data in the following table
     2  FROM PurchDB.Parts
C     retrieve only data that satisfies this search condition
     3  WHERE PartNumber = :PartNumber
```

SQL Comments

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

```
EXEC SQL SELECT * FROM PurchDB.Parts   -- This code selects Parts Table values.
  WHERE SalesPrice > 500.;
```

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

Continuation Lines

The FORTRAN preprocessor allows you to continue a *non-numeric literal* from one line to the next. Enter any character other than the digit 0 or a blank *in column 6*, and do not enter the character C, *, or $ in column 1. The FORTRAN preprocessor supports up to 19 continuation lines. The the FORTRAN/XL compiler allows you to extend the number of continuation lines, the FORTRAN compiler on other systems where the preprocessor is used does not allow more than 19 contination lines. Therefore, for consistency, the preprocessor only allows 19 contination lines.

```
      IF (PREDEFINEDCOMMENT .EQ. '5') THEN
          EXEC SQL INSERT INTO PURCHDB.VENDORS
     1      (VendorRemarks)
     2      VALUES ('This vendor is bad news.  Definitely place
     3          no orders.')
      ELSE
          EXEC SQL INSERT INTO PURCHDB.VENDORS
     1      (VendorRemarks)
     2      VALUES (:VendorRemarks)
      ENDIF
```

Declaring the SQLCA

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

Every ALLBASE/SQL FORTRAN program unit that contains embedded SQL statements must contain an SQLCA declaration. When a main program unit or a subprogram unit starts a DBE session, the SQLCA declaration must be in its Type Declaration Section. If a subprogram unit is called by a main program and each contains SQL commands to be executed in the same DBE session, the SQLCA declaration must appear in both the main program and the subprogram units.

As shown in Figure 3-1 at $\boxed{1}$, you declare the SQLCA by using the INCLUDE command:

```
        EXEC SQL INCLUDE SQLCA
```

When the FORTRAN preprocessor parses this command in the source file, it generates a complete FORTRAN declaration for this area. Some of the fields in the SQLCA are available for programmers to use:

```
SQLCODE
SQLERRD(3)
SQLWARN(0)
SQLWARN(1)
SQLWARN(2)
SQLWARN(6)
```

Some values ALLBASE/SQL places into these fields indicate warning and error conditions that resulted when the immediately preceding SQL command was executed. Other values simply provide information about normal command execution and are programmatically useful. For example, when you submit an UPDATE command, the number of rows updated is placed in SQLERRD(3). If this value is greater than 1, the program may want to advise the user of that condition and process a ROLLBACK WORK or COMMIT WORK command based on the user's response.

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

Declaring Host Variables

Variables used in SQL commands are known as **host variables**. All host variables used in either a main program unit or a subprogram unit must be declared in the Type Declaration Section of the program unit where the host variable is used. The host variable declarations must be the last declarations in the Type Declaration Section of the program unit which contains the embedded SQL statements and must appear between the two following SQL commands:

```
EXEC SQL BEGIN DECLARE SECTION
  .
  .  Host variables are declared here
  .  in FORTRAN variable type descriptions.
  .
EXEC SQL END DECLARE SECTION
```

In Figure 3-1, host variable declarations start at $\boxed{2}$.

You cannot put more than one such declaration section in a program unit. In addition, you must include the BEGIN DECLARE SECTION and END DECLARE SECTION commands even if there are no host variables used in the embedded SQL commands. This is because the preprocessor includes the variable declaration file after the END DECLARE SECTION command.

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

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

Because these host variables are for data from a DBEnvironment, the variable types must be compatible with the ALLBASE/SQL data stored in the respective columns:

`CHARACTER*16`	*PartNumber*
`CHARACTER*30`	*PartName*
`DOUBLE PRECISION`	*SalesPrice*
`SQLIND`	*SalesPriceInd*

The host variable named *SalesPrice* is accompanied by a second host variable, *SalesPriceInd*. Known as an **indicator variable**, this host variable is used to detect null values. If column *SalesPrice* contains a null value, ALLBASE/SQL returns a negative number to *SalesPriceInd*. The subprogram unit named *DisplayRow* examines the value in this indicator variable to determine whether to display the announcement that a *Sales Price is null* or display the value in *SalesPrice*.

Indicator variables are declared at $\boxed{2}$ in Figure 3-1 as *SQLIND*. SQLIND is a special declaration reserved for indicator variables.

Starting a DBE Session

In most application programs you embed the CONNECT command to start a DBE session in a program:

```
   EXEC SQL CONNECT TO DBEnvironmentName
```

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

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

Place the DBE session initiation command in a subprogram unit of your program such that it executes at runtime before all other SQL commands in your program *except* a WHENEVER command which may be executed before a CONNECT TO command is executed.

If the program uses either a dynamic command or a host variable to connect to a DBEnvironment, the command or variable must have been entered by the program user prior to attempting to execute any other embedded SQL command.

Defining Transactions

You define transactions in a program unit to control what changes get committed to a DBEnvironment and when they get committed.

A transaction consists of all the SQL commands that are executed between a BEGIN WORK command and either a COMMIT WORK command or a ROLLBACK WORK command. When a COMMIT WORK command is successfully executed, all operations performed by the transaction it ends have a permanent effect on the DBEnvironment. The opposite is true for a ROLLBACK WORK command; no operations performed by the transaction it ends have a permanent effect on the DBEnvironment.

The number and duration of transactions in an application program depend on such factors as:

- Concurrency: Concurrent DBE sessions may compete for data and index locks and buffers.
- Update activities: Applications that are update-intensive should issue COMMIT WORK commands more frequently to avoid data re-entry in the event of a failure.
- Data consistency: Program changes to a table that are meaningful only if changes are made to another table should be committed or undone at the same time to ensure the data remains consistent.

The commands at $\boxed{4}$ and $\boxed{5}$ in subroutines BeginTransaction and EndTransaction in Figure 3-1 start and end a transaction that consists of a single execution of the SELECT command at $\boxed{8}$ in subroutine *QueryTable*.

The BEGIN WORK command in subprogram unit *BeginTransaction* is optional but recommended. If you omit a BEGIN WORK command, ALLBASE/SQL automatically issues a BEGIN WORK on your behalf before executing the first SQL command that requires that a transaction be in progress.

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

Implicit Status Checking

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

```
EXEC SQL WHENEVER SQLERROR GOTO 500
        |        |
        |        |
        |        |
        |        the action
        |
        the condition
```

Preprocessor-generated statements for each WHENEVER command are embedded into the preprocessed code after each subsequent SQL command found in the program's source code. Because of this, you must either end each program unit which contains a WHENEVER *condition* GOTO *label* statement with a WHENEVER *condition* CONTINUE statement, or have a label in each subsequent program unit in the preprocessed source code which corresponds to the appropriate WHENEVER *condition* GOTO *label* statement. If no label exists and a WHENEVER *condition* CONTINUE statement is not entered, an error will occur at compile time.

For example, if execution of the SELECT command at $\boxed{8}$ or the SQLEXPLAIN command at $\boxed{10}$, illustrated earlier in this chapter, were to cause an error condition, ALLBASE/SQL would take no special action because the WHENEVER command shown at $\boxed{6}$ *precedes* both the SELECT and the SQLEXPLAIN commands in the source listing.

If, however, the WHENEVER SQLERROR CONTINUE statement was not entered and an error occurred at $\boxed{8}$, there would be a compile time error as there is no label 500 in Subroutine QueryTable. The WHENEVER SQLERROR CONTINUE command at $\boxed{6}$ turns off the implicit status checking of the command that appears earlier in the source listing:

```
EXEC SQL WHENEVER SQLERROR GOTO 500
```

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

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

Terminating a DBE Session

As illustrated at $\boxed{7}$ in Figure 3-1, you can terminate a DBE session with the RELEASE option of the COMMIT WORK command. The program in Figure 3-1 terminates its DBE session whenever:

- The user enters a slash (/) in response to the prompt in subprogram unit *QueryTable*.
- The program encounters an error serious enough to set Abort to *.TRUE.* in subprogram unit *SQLStatusCheck*.
- The program encounters an error when processing the CONNECT, BEGIN WORK, or COMMIT WORK commands.

Defining and Manipulating Data

You embed data definition and data manipulation commands in a subprogram unit.

Data Definition

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

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

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

```
 EXEC SQL CREATE INDEX PartNameINDEX
1                  ON PurchDB.Parts (PartName)
```

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

```
 EXEC SQL SELECT  PartName
1           INTO :PartName
2           FROM  PurchDB.Parts
3          WHERE  PartName LIKE :partialkey
```

Data Manipulation

SQL has four basic data manipulation commands:

- SELECT: retrieves data.
- INSERT: adds rows.
- DELETE: deletes rows.
- UPDATE: changes column values.

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

- Simple data manipulation: operations that retrieve single rows, insert single rows, or delete or update a limited number of rows.
- Processing with cursors: operations that use a cursor to operate on a row at a time within a set of rows. A cursor is a pointer the program advances through the set of selected rows.
- Dynamic operations: operations specified by the user or program at runtime.

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

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

```
      EXEC SQL SELECT  PartNumber, PartName, SalesPrice
     1          INTO :PartNumber,
     2               :PartName
     3               :SalesPrice :SalesPriceInd
     4          FROM  PurchDB.Parts
     5         WHERE  PartNumber = :PartNumber
```

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

```
      SalesPrice = response

      EXEC SQL SELECT  COUNT(PartNumber)
     1          INTO :PART-COUNT
     2          FROM  PurchDB.Parts
     3         WHERE  SalesPrice > :SalesPrice
```

All host variables used in a program unit must be declared in the Type Declaration Section in that program unit, as discussed earlier in this chapter under "Declaring Host Variables".

Explicit Status Checking

In explicit status checking, shown at 9 in Figure 3-1, you explicitly examine an SQLCA field for a particular value, then perform an operation depending on the field's value. In this example the SQLCA field named SQLCode is examined to determine whether it contains a value of:

- 0, indicating no error occurred.
- 100, indicating no rows qualified for the SELECT operation.
- -10002, indicating more than one row qualified for the SELECT operation.

If SQLCode contains any other value, subprogram unit *SQLStatusCheck* is executed. Values with greater negative values than -14024 indicate errors serious enough to warrant terminating the program.

Obtaining ALLBASE/SQL Messages

As shown at $\boxed{10}$ in Figure 3-1, you use the SQLEXPLAIN command to obtain a message from the ALLBASE/SQL message catalog that describes the condition related to the SQLCA value:

```
   EXEC SQL SQLEXPLAIN :SQLMessage
```

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

Sometimes more than one message may be needed to completely describe how an SQL command executed. To obtain multiple messages, the program in Figure 3-1 executes SQLEXPLAIN until SQLCode contains a value of 0. ALLBASE/SQL sets SQLCode to 0 when no more messages are available.

If you use the value of SQLCode to control the flow of the program, you may need to save the SQLCode value into a variable before the SQLEXPLAIN command is executed. The value of SQLCode represents the result of only the previously executed SQL command.

You can use SQLEXPLAIN in conjunction with either implicit or explicit status checking. In the program in Figure 3-1, the subprogram unit SQLStatusCheck is executed from the line labeled 500, which is used in conjunction with the first WHENEVER SQLERROR command in the program.

The default message catalog is SQLCTxxx.PUB.SYS. For native language users, the catalog is SQLCT000.PUB.SYS, where NATIVE-3000 is the current language. If this catalog is not available, ALLBASE/SQL issues a warning and uses the default catalog instead.

4

Host Variables

Host variables are data items used in SQL commands in both the main program unit and in subprogram units. They are used to pass the following information between an application program and ALLBASE/SQL:

- Data values.
- Null value indicators.
- Dynamic commands.
- Savepoint numbers.
- Messages from the ALLBASE/SQL message catalog.
- DBEnvironment Names

All host variables used in either the main program unit or a subprogram unit of a FORTRAN program must be declared in the Host Variable Declaration Section of the program unit where the host variable is used. 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 first identifies how and where in a program unit you can use host variables. Then it discusses how to write variable declaration types that complement the way they are used.

Using Host Variables

You use host variables in SQL commands as follows:

- To pass *data values*, when using the following data manipulation commands:

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

- To hold *null value indicators* in four data manipulation commands:

```
SELECT
INSERT
FETCH
UPDATE
```

- To pass *dynamic commands* at runtime, achieved by using the following commands:

```
PREPARE
EXECUTE IMMEDIATE
```

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

```
SAVEPOINT
ROLLBACK WORK TO
```

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

This chapter provides examples illustrating where, in the commands itemized above, the SQL syntax supports host variables. This chapter also takes a brief look at two special cases:

- Using host variables in subprogram units.
- Using host variables in conjunction with data in batch files.

Some of the examples are numbered so later in this chapter, under "Declaring Host Variables", you can quickly find declaration examples for the same host variables.

Host Variable Names

Host variable names in FORTRAN programs must:

- Contain from 1 to 30 ASCII characters.
- Conform to the rules for ALLBASE/SQL basic names.
- Contain only characters chosen from the following set: the 26 letters of the ASCII alphabet, the 10 decimal digits, or an underscore (_).
- Begin with an alpha character but not the prefix *EXEC SQL*.
- Not begin or end with a hyphen.
- Not be the same as any ALLBASE/SQL or FORTRAN reserved word.

In all SQL commands containing host variables, 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.

When using an input host variable, you must initialize it before using it.

Data Values and Null Indicators

Host variables containing *data values* can be input or output host variables. 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, one host variable, *PartNumber*, is used for both input *and* output.

Indicator Variables

Host variables that contain *null value indicators* are called **indicator variables**. **Indicator variables** are used in SELECT, FETCH, and INSERT commands to identify null values, and in SELECT and FETCH commands to identify truncated output strings.

In SELECT and FETCH commands, an indicator variable is an *output host variable* containing one of the following indicators, which describe the data ALLBASE/SQL returns:

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

In the INSERT command, an indicator variable is an *input host variable*. You put one of the following indicators into the indicator variable to tell ALLBASE/SQL when to insert a null value in a column:

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

An indicator variable must appear in an SQL command immediately *after* the host variable whose data it describes. 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 may occur:

```
   CREATE PUBLIC TABLE PurchDB.Parts
 1  (PartNumber      CHAR(16)     NOT NULL,
 2   PartName        CHAR(30),
 3   SalesPrice      DECIMAL(10,2))
 4   IN WarehFS
```

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.

You can use an indicator variable to detect truncated strings in the SELECT and FETCH commands. In the SELECT command illustrated above, *PartNameInd* contains a value greater than zero (>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.

Declaring Host Variables

You must declare all host variables in the Host Variable Declaration Section of the program unit where the host variable is used. In other words, a host variable used in the main program unit must be defined in the Host Variable Declaration Section of the main program unit. A host variable used in a subprogram unit must be defined in the Host Variable Declaration Section of that subprogram unit.

Declaring Variables for Data Types

All FORTRAN program units that contain embedded SQL commands must have a Host Variable Declaration Section. If your program unit does not use host variables but does contain embedded SQL statements, it must still contain a Host Variable Declaration Section to satisfy FORTRAN preprocessor requirements. If your program unit does not contain embedded SQL statements then no Host Variable Declaration Section is needed. The Host Variable Declaration Section is delimited by the EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION commands.

Host variables must be declared in every program unit where they are used. A Type Declaration Section may exist in any program or subprogram unit:

- If the program unit uses host variables from a called program or subprogram unit, you declare these host variables in the Host Variable Declaration Section of both the *calling* and the *called* program units.
- If host variable values come from an MPE XL data file or are written to an MPE XL data file in the program, you also declare these host variables in the Host Variable Declaration Section.

Regardless of where in a program unit the host variables are declared, they must appear between the BEGIN DECLARE SECTION and END DECLARE SECTION commands, as shown in Figure 4-1. These commands and any host variable declarations they delimit are referred to as a **Declare Section**. No more than one Declaration Section can appear in any one program unit.

Variable Declarations

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

```
EXEC SQL BEGIN DECLARE SECTION

CHARACTER*16   OrderNumber
|              |
|              |
|              a variable name
|
a variable type

EXEC SQL END DECLARE SECTION
```

The host variable name must be the same as the corresponding host variable name used in the SQL commands of that program unit. The variable type must satisfy ALLBASE/SQL and FORTRAN data type preprocessor requirements.

```
PROGRAM Main
.
.
.
EXEC SQL INCLUDE SQLCA
.
.
.
EXEC SQL BEGIN DECLARE SECTION
.
.   Declarations for host variables
.
EXEC SQL END DECLARE SECTION
.
.   Embedded SQL commands
.
END

SUBROUTINE Query
.
.
.
EXEC SQL INCLUDE SQLCA
.
.
.
EXEC SQL BEGIN DECLARE SECTION
.
.   Declarations for host variables
.
EXEC SQL END DECLARE SECTION
.
.   Embedded SQL commands
.
RETURN
END
```

Figure 4-1. Host Variable Declarations

Data Types

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

You can also declare program variables that are not host variables within a declare section. All variables that appear in a declare section, however, must have FORTRAN data types among those illustrated in Table 4-1. Table 4-2 shows data descriptions for ALLBASE/SQL program elements.

CHARACTER Data

You can insert strings ranging from 1 to 3996 bytes into a CHARACTER column.

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

VARCHAR Data

VARCHAR strings can range from 1 to 3996 bytes. ALLBASE/SQL stores the actual length of the string in a four-byte field preceding the string itself. In addition, ALLBASE/SQL stores only the actual value of the string, not any trailing blanks.

The CHARACTER data type in FORTRAN is equivalent to the VARCHAR data type in ALLBASE/SQL. The VendorRemarks column in the PurchDB.Vendors table is defined as VARCHAR(60). It is therefore declared as follows:

```
CHARACTER*60        VendorRemarks
```

SMALLINT Data

You can assign values ranging from -32768 to 32767 to a column defined as SMALLINT. The INTEGER*2 data type in FORTRAN is equivalent to the SMALLINT data type in ALLBASE/SQL.

```
INTEGER*2           VariableName
```

INTEGER Data

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

Table 4-1. Data Description Entries for Host Variables

SQL DATA TYPES	FORTRAN DATA DECLARATIONS
CHAR(1)	`CHARACTER` *DataName*
CHAR(n)	`CHARACTER*n` *DataName*
VARCHAR(n)	`CHARACTER*n` *DataName* *
SMALLINT	`INTEGER*2` *DataName*
INTEGER	`INTEGER` *DataName*
REAL	`REAL` *DataName* `or`
	`REAL*4` *DataName*
FLOAT(1..24)	`REAL` *DataName* `or`
	`REAL*4` *DataName*
FLOAT(1..53)	`DOUBLE PRECISION` *DataName* `or`
	`REAL*8` *DataName*
DOUBLE PRECISION	`DOUBLE PRECISION` *DataName* `or`
	`REAL*8` *DataName*
BINARY	`CHARACTER` *DataName* `or`
	`CHARACTER*n` *DataName*
VARBINARY	`CHARACTER*n` *DataName*
DECIMAL	`DOUBLE PRECISION` *DataName* `or`
	`REAL*8` *DataName*
DATE	`CHARACTER*10` *DataName*
TIME	`CHARACTER*8` *DataName*
DATETIME	`CHARACTER*23` *DataName*
INTERVAL	`CHARACTER*20` *DataName*
	* This declaration is for non-dynamic commands only. Refer to the chapter, “Using Dynamic Operations”, for a description of how to use VARCHAR dynamically.

Table 4-2. Program Element Declarations

PROGRAM ELEMENT	FORTRAN DATA DECLARATIONS	
Indicator variable	`SQLIND or INTEGER*2`	*IndVarName*
Dynamic commands	`CHARACTER*`*n*	*CommandName*
Savepoint numbers	`INTEGER`	*SavepointName*
Message catalog messages	`CHARACTER*`*n*	*MessageName*
DBEnvironment name	`CHARACTER*`*n*	*DBEnvironmentName*

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), map to a 4-byte float. The FLOAT(25) through FLOAT(53) and DOUBLE PRECISION specifications map to an 8-byte float.

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

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

Table 4-3. ALLBASE/SQL Floating Point Column Specifications

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

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

BINARY Data

As with other data types, use the CREATE TABLE or ALTER TABLE command to define a binary or varbinary column. Up to 3996 bytes can be stored in such a column. Each byte contains two hexadecimal digits. For example, suppose you insert data via a host variable into a database column defined as binary. The host variable contains the digits, 1234. In the database, these four digits are stored in two bytes. Each nibble (half byte) contains one digit in hexadecimal format.

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

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

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

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

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.

DECIMAL Data

The DECIMAL data type is not supported by FORTRAN 77. The DECIMAL data type is compatible with a FORTRAN DOUBLE PRECISION 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.

DATE, TIME, DATETIME, and INTERVAL Data

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

```
           EXEC SQL BEGIN DECLARE SECTION
   C       *** DATETIME DATA TYPE    ***
           CHARACTER*23  BatchStamp
   C       *** DATE DATA TYPE        ***
           CHARACTER*10  TestDate
           SQLIND        TestDateInd
   C       *** TIME DATA TYPE        ***
           CHARACTER*8   TestStart
           SQLIND        TestStartInd
   C       *** INTERVAL DATA TYPE    ***
           CHARACTER*20  LabTime
           SQLIND        LabTimeInd
           EXEC SQL END DECLARE SECTION

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

    EXEC SQL FETCH C1
   1                 INTO :BatchStamp,
   2                      :TestDate  :TestDateInd,
   2                      :TestStart :TestStartInd,
   4                      :LabTime   :LabTimeInd
```

Using Default Data Values

You can choose a default value other than NULL when you create or alter a table by using the DEFAULT specification. Then when data is inserted, and a given column is not in the insert list, the specified default value is inserted. Or when you alter a table, adding a column to existing rows, every occurrence of the column is initialized to the default value. (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 information on using the data types in your program. Also refer to the section, "Data Type 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
```

Indicator Variable Declaration

Each indicator variable must be declared immediately following the host variable it describes.

```
    SQLIND                VariableNameInd
```

When the FORTRAN preprocessor encounters SQLIND, it generates the following declaration in its place in the modified source file:

```
    INTEGER*2             VariableNameInd
```

Dynamic Command Variable Declaration

The maximum size of the host variables used to hold dynamic commands is 32,762 bytes.

Savepoint Number Variable Declaration

Since the maximum savepoint number is 2,147,483,647, a host variable for holding a savepoint number should be declared as an INTEGER.

Message Catalog Variable Declaration

The maximum size of host variables used to hold messages from the ALLBASE/SQL message catalog is 32,762 bytes.

DBEnvironment Name

The maximum file name, 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.

Data Type Compatibility

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

- When the data types of values transferred between your program and a DBEnvironment do not match.
- When data of one type is moved to a host variable of a different type.
- When values of different types appear on the same expression.

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

Table 4-4. Data Type Equivalency and Compatibility

ALLBASE/SQL DATA TYPE	CHARACTER*n	INTEGER	DOUBLE PRECISION REAL*8
CHAR	E	I	I
VARCHAR	E	I	I
BINARY	C	I	I
VARBINARY	C	I	I
DATE	C	I	I
TIME	C	I	I
DATETIME	C	I	I
INTERVAL	C	I	I
SMALLINT	I	E	C
INTEGER	I	E	C
DECIMAL	I	C	C
REAL	I	C	C
FLOAT	I	C	C

In some cases, data conversion may lead to truncation or overflow.

Character Data Conversion

When ALLBASE/SQL moves string data of one type to a host variable declared as a compatible type, the following occurs:

- When moving CHAR data to a VARCHAR variable, ALLBASE/SQL places the length of the string in the appropriate variable and pads the string on the right with spaces to fill up the VARCHAR string variable.
- When moving VARCHAR data to a CHAR variable, ALLBASE/SQL pads the string on the right with spaces to fill up the CHAR string variable.

When ALLBASE/SQL stores the value of a string host variable into a CHARACTER column, ALLBASE/SQL pads the value on the right with spaces to fill up the column.

Character 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:

```
      SUBROUTINE Select
      .
      .
      .
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*40             LittleString
      SQLIND                   LittleStringInd
      EXEC SQL END DECLARE SECTION
      .
      .
      .
      EXEC SQL SELECT  BigString
     1                INTO :LittleString :LittleStringInd
      .
      .
      .
      RETURN
      END
```

When the string 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*.

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

It is possible to store native language data in a character column defined as ASCII. If this happens, the results of truncation may be unpredictable. 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 available in FORTRAN have the following precedence:

- DOUBLE PRECISION
- INTEGER

The following example illustrates numeric type conversion:

```
    SUBROUTINE Select
    .
    .
    .
    EXEC SQL BEGIN DECLARE SECTION
    INTEGER            Discount
    INTEGER            PurchasePrice
    EXEC SQL END DECLARE SECTION
    .
    .
    .
    EXEC SQL SELECT (MAX)PurchasePrice * :Discount
   1                INTO :PurchasePrice
   2                FROM  PurchDB.OrderItems
    .
    .
    .
    RETURN
    END
```

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

After subtraction, data conversion occurs again before the DOUBLE PRECISION result is stored in the INTEGER host variable *MAXPurchasePrice*. In this case, the fractional part of the DOUBLE PRECISION 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 Host Variables for Data Values and Indicator Variables

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

```
isql=> INFO PurchDB.OrderItems;

Column Name         Data Type (length)         Nulls Allowed
--------------------------------------------------------------------
OrderNumber         Integer                         NO
ItemNumber          Integer                         NO
VendPartNumber      Char (16)                       YES
PurchasePrice       Decimal (10,2)                  NO
OrderQty            SmallInt                        YES
ItemDueDate         Char (8)                        YES
ReceivedQty         SmallInt                        YES
```

The declare section illustrated in Figure 4-2 contains variable types equivalent to the data types in the *PurchDB.OrderItems* table:

- *PurchasePrice* is declared as a DOUBLE PRECISION variable because it holds the DECIMAL result of an aggregate function on a DECIMAL column.
- *Discount* is declared as a DOUBLE PRECISION variable because it is used in an arithmetic expression with a DECIMAL value column, PurchasePrice.
- *OrderQty* is declared as an INTEGER*2 variable because it holds the SMALLINT column, *OrderQty*.
- *OrderQtyInd* is an indicator variable, necessary because the resulting of *OrderQty* can contain only null values. Note in the INFO example above that this column allows null values.
- *OrderNumber* is declared as INTEGER because the column whose data it holds is INTEGER.

```
    SUBROUTINE Select
    .
    .
    .
    EXEC SQL BEGIN DECLARE SECTION
    DOUBLE PRECISION             Discount
    DOUBLE PRECISION             PurchasePrice
    INTEGER*2                    OrderQty
    SQLIND                       OrderQtyInd
    INTEGER                      OrderNumber
    EXEC SQL END DECLARE SECTION
    .
    .
    .
    EXEC SQL SELECT  PurchasePrice * :Discount,
   1                        OrderQty
   2                  INTO :PurchasePrice,
   3                       :OrderQty :OrderQtyInd
   4                  FROM  PurchDB.OrderItems
   5                 WHERE  OrderNumber = :OrderNumber
    .
    .
    .
    RETURN
    END
```

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

The declare section illustrated in Figure 4-3 depicts how to declare host variables used in a SELECT command:

- *Discount* is a DOUBLE PRECISION host variable because it is used in an arithmetic expression with an aggregate function on a DECIMAL value. This eliminates any data truncation in the arithmetic operation between compatible but not equivalent data types.
- *PurchasePrice* is declared as a DOUBLE PRECISION host variable because it holds the DECIMAL result of an aggregate function on a DECIMAL column, PurchasePrice.
- *OrderQty* is declared as an INTEGER*2 variable because it holds the SMALLINT result of an aggregate function on a SMALLINT column, *OrderQty*.
- *OrderQtyInd* is an indicator variable, necessary because the result of *OrderQty* is null if column *OrderQty* contains only null values. Note in the previous INFO command example that the column OrderQty allows null values.

- *OrderNumber* is an INTEGER variable because the column whose data it holds, *OrderNumber*, is INTEGER.
- *LowValue* and *HighValue* are both declared as INTEGER host variables because they hold data compared with that in a column defined as INTEGER.
- *GroupCriterion* is declared as an INTEGER host variable because its value is compared in the HAVING clause with the result of a COUNT function, which is always an INTEGER value.

```
   SUBROUTINE Select
      .
      .
      .
      EXEC SQL BEGIN DECLARE SECTION
      DOUBLE PRECISION                 Discount
      DOUBLE PRECISION                 PurchasePrice
      INTEGER*2                        OrderQty
      SQLIND                           OrderQtyInd
      INTEGER                          OrderNumber
      INTEGER                          LowValue
      INTEGER                          HighValue
      INTEGER                          GroupCriterion
      EXEC SQL END DECLARE SECTION
      .
      .
      .
          EXEC SQL DECLARE Maxcursor CURSOR FOR
     1              SELECT  PurchasePrice * :Discount,
     2                      OrderQty,
     3                      OrderNumber
     4                FROM  PurchDB.OrderItems
     5               WHERE  OrderNumber
     6                      BETWEEN :LowValue AND :HighValue
     7            GROUP BY  OrderQty, OrderNumber
     8              HAVING  COUNT(ItemNumber) > :GroupCriterion
      .
      .
      .
          EXEC SQL FETCH   Maxcursor
     1              INTO  :PurchasePrice,
     2                    :OrderQty :OrderQtyInd,
     3                    :OrderNumber
      .
      .
      .
      RETURN
      END
```

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

Declaring Host Variables for Dynamic Commands

The command illustrated in Figure 4-4 names a host variable, *DynamicCommand*, for receiving an SQL command at runtime. This host variable should be declared as a CHARACTER variable.

```
      SUBROUTINE Prepare
      .
      .
      .
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*1024              DynamicCommand
      EXEC SQL END DECLARE SECTION
      .
      .
      .
          EXEC SQL PREPARE  CommandOnTheFly
     1                FROM :DynamicCommand
      .
      .
      .
      RETURN
      END
```

Figure 4-4. Declaring Host Variables for Dynamic Commands

Declaring Host Variables for Savepoint Numbers

The command illustrated in Figure 4-5 below sets a savepoint. The number associated with the savepoint is the number ALLBASE/SQL places in the host variable named *SavePoint1*. This host variable should be declared as INTEGER.

```
SUBROUTINE SavePoint1
.
.
.
EXEC SQL BEGIN DECLARE SECTION
INTEGER              SavePoint1
EXEC SQL END DECLARE SECTION
.
.
.
    EXEC SQL SAVEPOINT :SavePoint1
.
.
.
RETURN
END
```

Figure 4-5. Declaring Host Variables for Savepoint Numbers

Declaring Host Variables for Message Catalog Messages

The command illustrated in Figure 4-6 below puts a message from the ALLBASE/SQL message catalog into a host variable named *SQLMessage*. The following example illustrates how the host variable for holding the message might be declared.

```
      SUBROUTINE SQLStatusCheck
      .
      .
      .
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*132            SQLMessage
      EXEC SQL END DECLARE SECTION
      .
      .
      .
      EXEC SQL SQLEXPLAIN :SQLMessage
      WRITE(6,102)SQLMessage
 10   FORMAT(A132)
     .
     .
     .
      RETURN
      END
```

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

The host variable is declared as a CHARACTER data type. Regardless of how it is declared, ALLBASE/SQL moves spaces into the host variable before returning the message. Therefore the program does not have to initialize the host variable each time SQLEXPLAIN is executed.

Declaring Host Variables Passed from Subprograms

The example illustrated in Figure 4-7 below illustrates how to pass a host variable value between one subprogram unit and another subprogram unit. The passed host variable must be declared in both the *CallingProgram's* and the *CalledProgram's* type declaration section. Variables that are not used in an SQL command in the program need to be declared outside the Type Declaration Section for Host Variables.

```
      SUBPROGRAM CallingProgram
      .
      .
      .
      LOGICAL*2           PositiveResponse
      CHARACTER*16        PartNumber
      CHARACTER*30        PartName
      DOUBLE PRECISION    SalesPrice

      EXEC SQL INCLUDE SQLCA

      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION
      .
      .
      .
       EXEC SQL CONNECT TO 'PartsDBE'
      .
      .
      .
       IF (PositiveResponse) THEN
       WRITE(6,102) 'INSERT rows into the Parts Table.'
102    FORMAT (A80)

       CALL InsertSubpgm (PartNumber,PartName,SalesPrice)

       WRITE(6,103) PartNumber,PartName,SalesPrice
103       FORMAT('Part Number is:     ',A16,
     1        'Part Name is:       ',A30,
     2        'Sales Price is:     ',F10.2)
       ENDIF
      .
      .
      .
      RETURN
      END
```

Figure 4-7. Declaring Host Variables Passed From Subprograms

```
      SUBROUTINE INSERTSubpgm (PartNumber,PartName,SalesPrice)
      .
      .
      .
      EXEC SQL INCLUDE SQLCA
      .
      .
      .
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*16                PartNumber
      CHARACTER*30                PartName
      DOUBLE PRECISION            SalesPrice
      EXEC SQL END DECLARE SECTION
      .
      .
      .
          EXEC SQL INSERT INTO   PurchDB.Parts
     1                          (PartNumber,
     2                           PartName,
     3                           SalesPrice)
     4                  VALUES (:PartNumber,
     5                          :PartName
     6                          :SalesPrice)
      .
      .
      .
      RETURN
      END
```

Figure 4-7. Declaring Host Variables Passed From Subprograms (page 2 of 2)

Note that the INCLUDE SQLCA clause is in both the calling and the called code. The SQLCA Common Block must *always* be named in this clause in any program unit that has SQL commands to be executed. If the SQLCA Common Block is not included, the FORTRAN preprocessor will issue a warning message stating that the SQLCA Common Block is not included.

Declaring Host Variables for MPE XL File Values

The example illustrated in Figure 4-8 below illustrates the use of a host variable to hold data from an MPE XL file. As shown below, the host variable from the file is declared the same way as a host variable entered from the terminal.

```
      SUBROUTINE Dates

      EXEC SQL INCLUDE SQLCA

      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*8        OrderDate
      INTEGER            OrderNumber
      INTEGER            ItemNumber
      CHARACTER*30       PartName
      CHARACTER*16       PartNumber
      DOUBLE PRECISION   PurchasePrice
      EXEC SQL END DECLARE SECTION
      .
      .
      .
      OPEN (10, FILE = 'OrderDateFile',
     1 ACCESS = 'sequential', STATUS = 'old')
      READ (10,103) OrderDate
  103 FORMAT(A8)
      CLOSE (10)

          EXEC SQL SELECT    A.OrderNumber,
     1                       A.ItemNumber,
     2                       C.PartName,
     3                       A.PurchasePrice
     4                 INTO  :OrderNumber,
     5                       :ItemNumber,
     6                       :PartName,
     7                       :PurchasePrice
     8                 FROM  PurchDB.OrderItems    A,
     9                       PurchDB.SupplyPrice   B,
     1                       PurchDB.Parts         C,
     2                       PurchDB.Orders        D
     3                WHERE  A.VendPartNumber = B.VendPartNumber
     4                  AND  B.PartNumber     = C.PartNumber
     5                  AND  A.OrderNumber    = D.OrderNumber
     6                  AND  D.OrderDate      = :OrderDate
      .
      .
      .
      RETURN
      END
```

Figure 4-8. Declaring Host Variables for MPE XL File Values

Declaring Host Variables for DBEnvironment Names

The DBEnvironment whose name is stored in the host variable named SomeDBE is declared and initialized as illustrated in Figure 4-9.

```
      EXEC SQL BEGIN DECLARE SECTION
      .
      .
      .
      CHARACTER*128    SomeDBE
      .
      .
      .
      EXEC SQL END DECLARE SECTION
      .
      .
      .
      WRITE (6,101) 'Enter DBEnvironment name >'
101   FORMAT (A80)
      READ (5,102) SomeDBE
102   FORMAT (A128)
      .
      .
      .
      EXEC SQL CONNECT to :SomeDBE;
```

Figure 4-9. Declaring Host Variables for DBEnvironment Names

The host variable is declared as a CHARACTER. In this example, it is declared as a variable large enough to hold the relative file name of any DBECon file. Note that in this case, the DBEnvironment name does not have to be delimited by single quotation marks.

5

Runtime Status Checking and the SQLCA

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

- The command was successfully executed.
- The command could not be executed because an error condition occurred, but the current transaction may 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 character string was truncated.
- The command was executed, but a null value was eliminated from an aggregate function.
- The command was executed, but a warning condition resulted.
- 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.

ALLBASE/SQL returns status information into a common data structure block known as the SQLCA, which stands for **SQL C**ommunication **A**rea. The SQLCA has four data items which your programs can take advantage of:

- SQLCode, which is set to 0 if a command executes successfully, to a negative number identifying a specific error condition, or to 100 if no rows qualify for an SQL data manipulation operation.
- SQLErrd(3), which is set to the number of rows that ALLBASE/SQL put into output host variables for data retrieval operations or the number of rows that ALLBASE/SQL processed for data change operations. It is set to 0 when a single-row data change operation causes an error condition or when SQLCode = 100.
- SQLWarn(0), which is set to W when a warning condition occurs or when SQLWarn(6) is set to W.
- SQLWarn(1), which is set to W when a character string is truncated while being stored in a host variable.
- SQLWarn(2), which is set to W when a null value is eliminated from the argument set of an aggregate function.
- SQLWarn(6), which is set to W when an error occurs that caused ALLBASE/SQL to abort the current transaction.

These data items can be used in several ways to perform runtime status checking:

- You can use the WHENEVER command to perform **implicit status checking**. When you use this command, 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 FORTRAN code that explicitly examines one or more of the SQLCA data items, 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. When several errors or warnings occur, you can use SQLEXPLAIN to retrieve messages for all of them. Refer to the *ALLBASE/SQL Message Manual* for an explanation of all error and warning messages.

This chapter examines the need for runtime status checking. It describes the SQLCA COMMON block 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.

The Importance of Status Checking

Status checking is performed primarily for three reasons:

- To gracefully handle runtime error and warning conditions.
- To maintain data consistency.
- To return information about the most recently executed command, such as how many rows ALLBASE/SQL processed.

Handling Runtime Errors and Warnings

A program is said to be **robust** if it anticipates common runtime errors and handles them gracefully. In on-line 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 SQLEXPLAIN would retrieve 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 or automatically re-execute the transaction a finite number of times before notifying the user of the deadlock.

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 by transactions operating on tables having these dependencies.

In the case of the sample database, 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 in order 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.Orders
    1               WHERE OrderNumber = :OrderNumber


     .
     .    If this command succeeds, the program
     .    submits the following command.
     .
     EXEC SQL DELETE FROM PurchDB.OrderItems
    1               WHERE OrderNumber = :OrderNumber


     If this command succeeds, the program
     submits a COMMIT WORK command.  If this
     command does not succeed, the
     program submits a ROLLBACK WORK command
     to ensure that all rows related to the
     order are deleted at the same time.
```

Determining Number of Rows Processed

Knowing such information as the following about rows your program handles helps determine the action to take in the program:

- No rows qualify for a data retrieval or change operation.
- A certain number of rows were retrieved by ALLBASE/SQL and placed in output host variables.
- A certain number of rows were inserted, deleted, or updated.

When no rows qualify for an SQL command that retrieves, inserts, or changes rows, ALLBASE/SQL sets SQLCode to 100. In the following example, when a row in the *PurchDB.Orders* table does not exist for the order number specified in *OrderNumber*, SQLCode contains a 100 after ALLBASE/SQL processes the UPDATE command:

```
    EXEC SQL  UPDATE PurchDB.Orders
   1             SET OrderDate   = :OrderDate
   2           WHERE OrderNumber = :OrderNumber
```

When this situation arises, the program can inform the user that the update operation could not be performed and prompt for another order number.

When one or more rows do qualify for a data manipulation or retrieval operation, ALLBASE/SQL sets SQLErrd(3) to the number of rows processed. In the following example, the SQLErrd(3) value determines whether or not subprogram unit *DisplayRow* is executed:

```
         .
         .
         EXEC SQL SELECT  PartNumber, PartName
        1           INTO :PartNumber
        2                :PartName
        3           FROM  PurchDB.Parts
        4          WHERE PartNumber = :PartNumber
         .
         .
         .
         IF (SQLErrd(3) .GT. 1) THEN
         CALL SQLStatusCheck
         ELSE
         CALL DisplayRow
         ENDIF
         .
         .
         .
         SUBROUTINE DisplayRow
         .
         .      This subprogram unit displays one row
         .      and performs only one SQL command.
         .
         RETURN
         END
```

When more than one row qualifies for a SELECT operation, SQLCode is set to -10002, and ALLBASE/SQL returns none of the rows. Your program can warn the user that no rows could be displayed:

```
          SUBROUTINE SQLStatusCheck
          .
          .
          .
          IF (SQLCode .EQ. -10002) THEN
            WRITE(6,102) 'More than one row qualified for '
            WRITE(6,102) 'this operation; none of the rows '
            WRITE(6,102) 'can be displayed.'
   102      FORMAT(A80)
               ELSE
                 CALL DisplayRow
               ENDIF
           RETURN
           END
```

If one or more rows qualify for a data INSERT, DELETE, or UPDATE operation, ALLBASE/SQL sets SQLErrd(3) to that number. In the case of UPDATE and DELETE operations, if SQLErrd(3) contains a value greater than one, you can warn the user that more than one row will be updated or deleted and give the user the opportunity to COMMIT WORK or ROLLBACK WORK.

The SQLCA COMMON Block

Every ALLBASE/SQL FORTRAN program unit must have the EXEC SQL INCLUDE SQLCA statement before the Host Variable Declaration Section to declare the SQL Communication Area:

```
         EXEC SQL INCLUDE SQLCA

         EXEC SQL BEGIN DECLARE SECTION
         .
   C     Host Variable Declaration Section
         .
         EXEC SQL END DECLARE SECTION
```

The FORTRAN preprocessor generates the following declaration in the modified source file after it parses this SQL command:

```
   C**** Start SQL Preprocessor ****
   C     EXEC SQL INCLUDE SQLCA
   C
   C**** Start Inserted Statements ****
         CHARACTER SQLCAID*8
         INTEGER   SQLCABC,
        1          SQLCODE
         INTEGER   SQLERRL
         CHARACTER SQLERRM*254,
       1          SQLERRP*8
        INTEGER   SQLERRD(6)
        CHARACTER SQLWARN(0:7)
        INTEGER   SQLEXT(2)
        CHARACTER SQLWARN0,SQLWARN1,SQLWARN2,SQLWARN3,
       1          SQLWARN4,SQLWARN5,SQLWARN6,SQLWARN7
        EQUIVALENCE (SQLWARN0,SQLWARN(0)),
       1            (SQLWARN1,SQLWARN(1)),
       1            (SQLWARN2,SQLWARN(2)),
       1            (SQLWARN3,SQLWARN(3)),
       1            (SQLWARN4,SQLWARN(4)),
       1            (SQLWARN5,SQLWARN(5)),
       1            (SQLWARN6,SQLWARN(6)),
       1            (SQLWARN7,SQLWARN(7))
        COMMON /Sqlca/ SQLCAID,SQLCABC,SQLCODE,SQLERRL,
       1            SQLERRM,SQLERRP,SQLERRD,SQLWARN,SQLEXT
 C**** End SQL Preprocessor ****
```

The following fields in this record are available for you to use in status checking.

```
SQLCODE
SQLERRD(3)
SQLWARN(0)
SQLWARN(1)
SQLWARN(2)
SQLWARN(6)
```

The other fields are reserved for use by ALLBASE/SQL *only*.

As discussed in Chapter 4, the SQLCA COMMON block must be included whenever a program unit executes SQL commands. If no EXEC SQL INCLUDE SQLCA statement is included, the FORTRAN preprocessor will issue a warning message. If a program accesses multiple DBEnvironments, each DBEnvironment requires a *separate* SQLCA. Consequently, ensure that all program units that access the same DBEnvironment are preprocessed separately from any program units that access a different DBEnvironment.

SQLCODE

SQLCode can contain one of the following values:

- 0, when an SQL command executes without generating a warning or error condition.
- A negative number, when an SQL command cannot be executed because an error condition exists.
- 100, when no row qualifies for one of the following commands, but no error condition exists:

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

Note that when you execute UPDATE or DELETE commands dynamically and no rows qualify for the operation, SQLCode is not set to 100. You can use SQLErrd(3) to detect this condition as discussed later in this chapter.

Negative SQLCode values are the same as the numbers associated with their corresponding messages in the ALLBASE/SQL message catalog. For example, the error message associated with an SQLCode of -2613 is:

```
Precision digits lost in decimal operation multiply.  (DBERR 2613)
```

SQLCode is set by all SQL commands *except* the following directives:

```
BEGIN DECLARE SECTION
DECLARE CURSOR
END DECLARE SECTION
INCLUDE SQLCA
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 information 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 (SQLCode .EQ. 100) THEN
            WRITE(6,102) 'No rows qualified for this operation.'
102         FORMAT(A80)
         ELSEIF (SQLCode .LT. 0) THEN
            CALL SQLStatusCheck
         ENDIF
         ⋮
     SUBROUTINE SQLStatusCheck
         ⋮
     SQLCodeTmp = SQLCode
     DO WHILE (SQLCode .NE. 0)
       EXEC SQL SQLEXPLAIN :SQLMessage
       CALL WriteOut (SQLMessage)
     END DO
     SQLCode = SQLCodeTmp
     .
     .
     RETURN
     END
```

The subroutine 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 subsequently executed, the *next* SQLCode becomes available to the program, and so on until SQLCode equals zero. If the user needs to have further access to a SQLCode value, the SQLCode value needs to be saved into another data variable. Each time SQLEXPLAIN or any other SQL command is executed, the SQLCode value changes to reflect the result of the previously executed command.

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 only negative SQLCodes and the SQLWarn(0) *W* flag have messages to describe their corresponding conditions.

The SQLCode is also used in implicit status checking:

- ALLBASE/SQL tests for the condition SQLCode less than zero (<0) when you use the SQLERROR option of the WHENEVER command.
- ALLBASE/SQL tests for the condition SQLCode equal to 100 (=100) when you use the NOT FOUND option of the WHENEVER command.

In the following situation, when ALLBASE/SQL detects a negative SQLCode, the code routine at *Label 2000* in the same program unit is executed. When ALLBASE/SQL detects an SQLCode of 100, the code routine at label *4000* in the same program unit is executed instead:

```
   EXEC SQL WHENEVER SQLERROR GOTO 2000
   EXEC SQL WHENEVER NOT FOUND GOTO 4000
```

WHENEVER commands remain in effect for *all* SQL commands that appear sequentially after them in the modified source code until another WHENEVER command for the same condition occurs. The following WHENEVER command, for example, changes the effect of an SQLCode of 100. Instead of the code routine at *Label 4000* in the same program unit being executed, the code routine at label *4500* in the same program unit is executed:

```
   EXEC SQL WHENEVER NOT FOUND GOTO 4500
```

The scope of WHENEVER commands is fully explained later in this chapter under “Implicit Status Checking.”

SQLERRD(3)

SQLErrd(3) can contain one of the following values:

- 0, when SQLCode is 100 or when one of the following commands causes an error condition:

  ```
  INSERT
  UPDATE
  DELETE
  UPDATE WHERE CURRENT
  DELETE WHERE CURRENT
  ```

  If an error occurs during execution of an INSERT, UPDATE, or DELETE command, one or more rows may have been processed prior to the error. In these cases, you may want to either COMMIT WORK or ROLLBACK WORK depending on the application. For example, if for logical data consistency all or no rows should be deleted, use ROLLBACK WORK. If logical data consistency is not an issue, COMMIT WORK may minimize re-processing time.
- A positive number that provides information about the number of rows processed in any data manipulation command.

The meaning of any positive SQLErrd(3) value depends on the SQLCode value.

When SQLCode is *0*, SQLErrd(3) indicates:

- The number of rows processed 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
  FETCH
  ```

SQLWARN(0)

A *W* in SQLWarn(0) in conjunction with a 0 (zero) 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 authority.  (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 command. If the transaction performed no UPDATE, INSERT, or DELETE operations, this situation causes no 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.

An error and a warning condition may exist at the same time. In this event, SQLCode is set to a negative number, but SQLWarn(0) is set to *W* only if SQLWarn(6) is set to *W*. Messages describing all the warnings and errors can be displayed as follows:

```
            .
            .
            IF (SQLCode .NE. 0) THEN
               DO WHILE (SQLCode .NE. 0)
               CALL DisplayMessage
               END DO
            ENDIF
            .
            .
         SUBROUTINE DisplayMessage
         EXEC SQL SQLEXPLAIN :SQLMessage
         WRITE(6,102) SQLMessage
   102   FORMAT(A120)
         .
         .
         RETURN
         END
```

If multiple warnings but no errors result when ALLBASE/SQL processes a command, SQLWarn(0) is set to *W* and remains set until the last warning message has been retrieved by SQLEXPLAIN or another SQL command is executed. In the following example, *DisplayWarning* is executed when this condition exists:

```
         ⋮
         IF ((SQLWarn(0) .EQ. 'W') .AND. (SQLCode .EQ. 0)) THEN
            DO WHILE (SQLWarn(0) .EQ. 'W')
            CALL DisplayWarning
            END DO
         ENDIF
         ⋮
         SUBROUTINE DisplayWarning
         ⋮
         EXEC SQL SQLEXPLAIN :SQLMessage
         WRITE(6,102) SQLMessage
   102   FORMAT(A120)
         ⋮
         RETURN
         END
```

When you use the SQLWARNING option of the WHENEVER command, ALLBASE/SQL checks for a *W* in SQLWarn(0). You can use the WHENEVER command to do *implicit* status checking equivalent to that done *explicitly* above as follows:

```
      EXEC SQL WHENEVER SQLWARNING GOTO 3000
      EXEC SQL WHENEVER SQLERROR GOTO 2000
```

When a warning condition that sets SQLWarn(0) occurs, SQLCode does not contain a value that describes the warning. Therefore you cannot explicitly evaluate the contents of SQLCode in order to conditionally handle warnings. You can either display the message SQLEXPLAIN retrieves from the ALLBASE/SQL catalog or you can ignore the warning.

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:

- 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 aggregrate function.

For example:

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

If any OrderQty values are null:

- :SQLWarn(2) is set to W.
- SQLCode is set to 0.
- SQLEXPLAIN retrieves the message:

```
NULL values eliminated from the argument of an aggregate
function.   (DBWARN 2041)
```

SQLWARN(6)

When an error exists so serious that ALLBASE/SQL has 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 is *-14024* or less:

- An SQLCode of -4008 indicates that ALLBASE/SQL does not have access to the amount of shared memory required to complete the execution of an open transaction:

  ```
  ALLBASE/SQL Shared Memory allocation failed in DBCore. (DBERR 4008)
  ```

- An SQLCode of -14024 indicates that a deadlock has occurred:

  ```
  Deadlock detected.  (DBERR 14024)
  ```

  A deadlock exists when each of two transactions needs 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 with a greater negative value than -14024 indicates that the error is serious enough to warrant terminating your program. For example, when the log file is full, log space needs to be reclaimed before ALLBASE/SQL can process any additional transactions:

  ```
  Log full.  (DBERR 14046)
  ```

When these errors occur, ALLBASE/SQL sets SQLWarn(6) to *W*, SQLWarn(0) to *W*, and SQLCode to a *negative number*. You only need to examine SQLWarn(6) if you want to terminate your program any time ALLBASE/SQL has to roll back the current transaction:

```
IF ((SQLCode .LT. 0) .AND. ( SQLWARN(6) .EQ. 'W')) THEN
     CALL SQLStatusCheck
     CALL TerminateProgram
ELSE
     CALL SQLStatusCheck
ENDIF
```

In this example, the program executes subprogram unit *SQLStatusCheck* when an error occurs. The program terminates whenever SQLWarn(6) is *W*, but continues if SQLWarn(6) is *not W*.

If a deadlock or a shared memory problem occurs, the contention that caused it may not exist if the transaction is restarted. In this case, you may want to examine *both* SQLWarn(6) and SQLCode and terminate the program only when SQLCode is less than -14024:

```
      .
      .
      .

100   CONTINUE
C     This is the RESTART POINT
      .
      .
      .
      IF (SQLCode .GT. -14025) THEN
        DO WHILE (SQLCode .NE. 0)
          EXEC SQL SQLEXPLAIN :SQLMessage
          CALL WriteOut (SQLMessage)
        END DO
        GOTO 100
      ENDIF
      IF ((SQLWARN(6) .EQ. 'W') .AND. (SQLCode .LT. -14024)) THEN
        DO WHILE (SQLCode .NE. 0)
          EXEC SQL SQLEXPLAIN :SQLMessage
          CALL WriteOut (SQLMessage)
        END DO
        CALL TerminateProgram
      ENDIF
```

If a deadlock or a shared memory problem occurs, the program displays all the messages, then continues. The program also continues when an error exists but is not serious enough to cause ALLBASE/SQL to roll back the current transaction. In the case of serious errors, however, SQLCode is set to less than -14024, and the program terminates after displaying all the messages.

If multiple SQLCodes result when ALLBASE/SQL processes a command that causes the current transaction to be rolled back, SQLWarn(6) is set to *W* in conjunction with the *first available* SQLCode. Therefore, if your program needs to examine SQLWarn(6), ensure that you examine it *before* using SQLEXPLAIN for the second time or it will be reset.

If one or more errors are detected before an automatic rollback occurs, the first SQLCode available to your program will not be equal to -4008 or greater than or equal to -14024. However, should one of these conditions occur, the corresponding SQLCode is guaranteed to be the last SQLCode available to your program, since ALLBASE/SQL rolls back the current transaction and does not continue to look for additional errors. You can use this characteristic to construct a test such that a transaction is automatically reapplied behind the program user's back only if a deadlock or a shared memory problem occurs but no other errors were detected first:

```
      TryCounter = 0
      TryLimit   = 3
      .
      .
100   IF (SQLCommandDone) THEN
        .
        .  Program user is prompted for a part number.
        .
      SQLCommandDone = .TRUE.
        .
        .  A SELECT command is attempted.
        .
        Trycounter = TryCounter +1
      .
      .
      IF ((SQLCode .EQ. -14024).OR.(SQLCode .EQ. -4008)) THEN
        IF (Trycounter .EQ. TryLimit) THEN
          SQLCommandDone = .FALSE.
          WRITE (*,*) 'Could not complete transaction.'
          WRITE (*,*)   'Try again later if you want.'
        ELSE
          SQLCommandDone = .TRUE.
        ENDIF
      ELSE
        Abort = .FALSE.
        IF (SQLWarn(6) .EQ. 'W') THEN
          Abort = .TRUE.
        ENDIF
        DO WHILE (SQLCode .NE. 0)
          EXEC SQL SQLEXPLAIN :SQLMessage
          WRITE (*,110) SQLMessage
110       FORMAT(A120)
        END DO
        IF (Abort) THEN
          CALL TerminateProgram
        ELSE
          SQLCommandDone = .TRUE.
        ENDIF
      ENDIF
      .
      .
      GOTO 100
```

At this point, a SELECT command is executed. If an error occurs, and 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 reapplied. If an error with an SQLCode less then -14024 occurred, the program is terminated after the error messages are displayed.

Approaches to Status Checking

You can use one or both of the following approaches to checking SQLCA values:

- Implicit status checking. This approach utilizes the WHENEVER command to check SQLWarn(0) or SQLCode values. This type of status checking is most useful when control can be passed to *one predefined point* in the program unit to handle warnings and errors.
- Explicit status checking. This approach uses your own FORTRAN statements to explicitly examine SQLWarn(0), SQLWarn(6), SQLCode, or SQLErrd(3). This type of status checking is useful when you want to test for specific SQLCA values before passing control to *one of several locations* in the program.

Error and warning conditions detected by either type of status checking can be conveyed to the program user in several 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 has messages for every negative SQLCode and for every condition that sets SQLWarn(0).
- Your own messages can be displayed when a certain condition occurs.
- No message may be displayed, as when a condition exists that is irrelevant to the program user.

This section illustrates various ways to use explicit and implicit status checking and notify program users of the results of status checking.

Implicit Status Checking

The WHENEVER command consists of two components: a **condition** and an **action**:

`EXEC SQL WHENEVER` *Condition Action*

There are three conditions:

- SQLERROR. If WHENEVER SQLERROR is in effect, ALLBASE/SQL checks for the existence of a negative SQLCode after processing any SQL command *except*:

```
BEGIN DECLARE SECTION                    INCLUDE
DECLARE CURSOR                           SQLEXPLAIN
END DECLARE SECTION                      WHENEVER
```

- SQLWARNING. If WHENEVER SQLWARNING is in effect, ALLBASE/SQL checks for the existence of a *W* in SQLWarn(0) after processing any SQL command *except*:

```
BEGIN DECLARE SECTION                   INCLUDE
DECLARE CURSOR                          SQLEXPLAIN
END DECLARE SECTION                     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 also three actions:

- STOP. If WHENEVER *Condition* STOP is in effect, ALLBASE/SQL rolls back the current transaction and terminates the DBE session and the program is terminated 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 *Label*. If WHENEVER *Condition* GOTO *Label* is in effect, the code routine located at that numeric label is executed when the *Condition* exists. The label must appear in the same program unit where the condition exists. GOTO and GO TO forms of this action have exactly the same effect.

Any of these three actions may be specified for any of these three *conditions*.

The WHENEVER command causes the FORTRAN preprocessor to generate status-checking and status-handling code for each SQL command that comes after it *sequentially* in the program. In the following program sequence, for example, the WHENEVER command in *SubprogramUnit1* is in effect for *SQLCOMMAND1*, but not for *SQLCOMMAND2*, even though *SQLCOMMAND1* is executed first at runtime:

```
           .
           .
           CALL SubprogramUnit1
           CALL SubprogramUnit2
           .
           .
        SUBROUTINE SubprogramUnit2
        .
        .
            EXEC SQL SQLCOMMAND2
        .
        .
        RETURN
        END

        SUBROUTINE SubprogramUnit1
        .
        .
           EXEC SQL WHENEVER SQLERROR GOTO 2000
           EXEC SQL WHENEVER SQLWARNING GOTO 3000
           EXEC SQL WHENEVER NOT FOUND GOTO 4000
        .
        .
           EXEC SQL SQLCOMMAND1
        .
        .
  2000  CALL ErrorHandler
        .
  3000  CALL WarningHandler
        .
  4000  CALL NotFoundHandler
        .
        .
           EXEC SQL WHENEVER SQLERROR CONTINUE
           EXEC SQL WHENEVER SQLWARNING CONTINUE
           EXEC SQL WHENEVER NOT FOUND CONTINUE
        RETURN
        END
```

The code generated reflects the condition and action in a WHENEVER command. In the example above, the preprocessor inserts both a test for a negative value in SQLCode, an SQLCode value equal to 100, and an SQLWarn(0) value equal to W, and a statement that invokes the error handling code routines located at *Labels 2000, 3000,* and *4000* respectfully:

```
      SUBROUTINE SubprogramUnit1
      .
      .
C**** Start SQL Preprocessor ****
C        EXEC SQL WHENEVER SQLERROR GOTO 2000
C        EXEC SQL WHENEVER SQLWARNING GOTO 3000
C        EXEC SQL WHENEVER NOT FOUND GOTO 4000
C**** Start Inserted Statements ****
C**** End SQL Preprocessor ****
      .
      .
C   **** Start SQL Preprocessor ***
C        EXEC SQL SQLCOMMAND1
C   **** Start Inserted Statements ****
         IF (SQLCODE .EQ. 0) THEN
         CALL SQLXCO(SQLCAID,Statements for executing
   1                SQLCOMMAND1 appear here)
           IF (SQLWARN(0) .EQ. 'W') THEN
             GO TO 3000
           END IF
         ELSE IF (SQLCODE .EQ. 100) THEN
           GO TO 4000
         ELSE IF (SQLCODE .LT. 0) THEN
           GO TO 2000
         END IF
C   **** End SQL Preprocessor ****
      .
      .
2000  CALL ErrorHandler
      .
3000  CALL WarningHandler
      .
4000  CALL NotFoundHandler
      .
      .
C**** Start SQL Preprocessor ****
C        EXEC SQL WHENEVER SQLERROR CONTINUE
C**** Start Inserted Statements ****
C**** End SQL Preprocessor   ****
C**** Start SQL Preprocessor ****
C        EXEC SQL WHENEVER SQLWARNING CONTINUE
C**** Start Inserted Statements ****
C**** End SQL Preprocessor   ****
C**** Start SQL Preprocessor ****
C        EXEC SQL WHENEVER NOT FOUND CONTINUE
C**** Start Inserted Statements ****
C**** End SQL Preprocessor   ****
```

```
      RETURN
      END
```

As this example illustrates, you can pass control with a WHENEVER command to an exception-handling code routine within the same program unit where the error condition occurred. Because you use a GOTO statement rather than a CALL statement, after the exception-handling subprogram unit is executed, control cannot *automatically* return to the statement which caused the error to occur. You must use another GOTO or a CALL statement to explicitly pass control to a specific point in your program:

```
      SUBROUTINE ErrorHandler
        .
        .
          IF (SQLCode .LT. -14024) THEN
            CALL TerminateProgram
          ELSE
            DO WHILE (SQLCode .NE. 0)
               EXEC SQL SQLEXPLAIN :SQLMessage
               CALL WriteOut (SQLMessage)
            END DO
              CALL BeginningOfProgram
C            (* CALL Restart/Reentry point of program *)
          ENDIF
        .
        .
        RETURN
        END
```

This exception-handling subprogram unit *explicitly* checks the first SQLCode returned. The program terminates or it continues from the *Restart/Reentry point* after all warning and error messages are displayed. Note that a CALL statement had to be used in this code routine in order to allow the program to transfer control to a specific point. A GOTO statement transfers control only to another point in the same subprogram unit and a RETURN statement returns control to the point in the program where the error handling subprogram unit was called. Using a CALL 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. How to handle this case is discussed under "Explicit Status Checking" later in this chapter.

The FORTRAN preprocessor generates status-checking and status-handling code for each SQL command that comes after a WHENEVER statement in the source code until another WHENEVER statement is found. If the WHENEVER statement includes a GOTO, there must be a corresponding label in each subsequent subprogram unit following the WHENEVER statement that includes SQL commands, or until another WHENEVER statement is encountered. It is recommended that a WHENEVER condition CONTINUE statement be included at the end of each subprogram unit that contains a WHENEVER condition GOTO statement to eliminate the possibility of having an unresolved external error at compile time.

Implicitly Invoking Status-Checking Subprogram Units

The program illustrated in Figure 5-1 contains five WHENEVER commands:

- The WHENEVER command numbered 1 handles errors associated with the following commands:

  ```
  CONNECT
  BEGIN WORK
  COMMIT WORK
  ```

- The WHENEVER command numbered 2 turns off the previous WHENEVER command.
- The WHENEVER commands numbered 3 through 5 handle warnings and errors associated with the SELECT command.
- The WHENEVER commands numbered 6 turns off the previous WHENEVER commands.

The code routine located at *Label 1000* is executed when an error occurs during the processing of session-related and transaction-related commands. The program terminates after displaying all available error messages. If a warning condition occurs during the execution of these commands, the warning condition is ignored, because the WHENEVER SQLWARNING CONTINUE command is in effect by default.

The code routine located at *Label 2000* is executed when an error occurs during the processing of the SELECT command. 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, subprogram unit SQLStatusCheck is executed. SQLStatusCheck *explicitly* examines SQLCode to determine whether a deadlock or shared memory problem occurred (SQLCode = -14024 or -4008 respectively) or whether the error was serious enough to warrant terminating the program (SQLCode < -14024).

- If a deadlock or shared memory problem occurred, the program attempts to execute the SelectQuery subprogram unit starting at *Label 1001* as many as three times before notifying the user of the deadlock or shared memory condition and terminating the program.

- 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 returns to subprogram unit SelectQuery and prompts the user for another PartNumber.

The code routine located at *Label 3000* is executed when only a warning condition results during execution of the SELECT command. This code routine displays a message and the row of data retrieved, commits work, and then prompts the user for another PartNumber.

The NOT FOUND condition that may be associated with the SELECT command is handled by the code routine located at *Label 4000*. This code routine displays the message, Row not found!, then passes control to subprogram unit *EndTransaction*. SQLEXPLAIN does not provide a message for the NOT FOUND condition, so the program must provide one itself.

```
      PROGRAM forex5
C
C     ******************************************************
C     *  This program illustrates the use of SQL's SELECT  *
C     *  command to retrieve one row or tuple of data at   *
C     *  a time. BEGIN WORK is executed before the SELECT  *
C     *  and COMMIT WORK is executed after the SELECT. An  *
C     *  indicator variable is used for SalesPrice.        *
C     *  This program is like forex2  except this program  *
C     *  handles deadlocks and error handling differently. *
C     ******************************************************
C
C            (* Begin SQL Communication Area *)
C
      EXEC SQL INCLUDE SQLCA
C
C
C     ****************************************************
C     *  Data Type Conversions :                         *
C     *    Character         = SQL Char(1)               *
C     *    Character*n       = SQL Char(n)               *
C     *    Character*n       = SQL VarChar               *
C     *    Double Precision  = SQL Float                 *
C     *    Double Precision  = SQL Decimal               *
C     *    Integer           = SQL Integer               *
C     *    Integer*2         = SQL SmallInt              *
C     ****************************************************
C
C             (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION
C
C            (* End Host Variable Declarations *)
C
C            (* Beginning of the Main Program *)
C
      WRITE (*,*) CHAR(27), 'U'
      WRITE (*,*) 'Program to SELECT specified rows from the Parts table
     1 -- forex5'
      WRITE (*,*) 'Event List:'
      WRITE (*,*) '  CONNECT TO PartsDBE'
      WRITE (*,*) '  BEGIN WORK'
      WRITE (*,*) '  SELECT a specified row from the Parts table until u
     1ser enters a "/"'
      WRITE (*,*) '  COMMIT WORK'
      WRITE (*,*) '  RELEASE PartsDBE'
```

Figure 5-1. Program forex5: Implicit and Explicit Status Checking

```
C
      CALL ConnectDBE
      CALL SelectQuery
      CALL TerminateProgram
C
      STOP
      END
C
C     (* Beginning of the Sub-Routines *)
C
      SUBROUTINE ConnectDBE
C            (* Subroutine to Connect to PartsDBE *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION
C
      EXEC SQL WHENEVER SQLERROR GOTO 1000
C
      WRITE (*,*) ' '
      WRITE (*,*) 'CONNECT TO PartsDBE'
      EXEC SQL CONNECT TO 'PartsDBE'
      GOTO 1100
C
1000  CALL SQLStatusCheck
      CALL TerminateProgram
C
1100  RETURN
      EXEC SQL WHENEVER SQLERROR CONTINUE
      END
C     (* End of ConnectDBE Subroutine *)
C
C
      SUBROUTINE BeginTransaction
C            (* Subroutine to Begin Work *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
```

Figure 5-1. Program forex5: Implicit and Explicit Status Checking (page 2 of 8)

```
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION
C
      EXEC SQL WHENEVER SQLERROR GOTO 1000
C
      WRITE (*,*) 'BEGIN WORK'
      EXEC SQL BEGIN WORK
      GOTO 1100
C
1000  CALL SQLStatusCheck
      CALL TerminateProgram
C
1100  RETURN
      EXEC SQL WHENEVER SQLERROR CONTINUE
      END
C     (* End BeginTransaction Subroutine *)
C
C
      SUBROUTINE EndTransaction
C     (* Subroutine to Commit Work *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION
C
      EXEC SQL WHENEVER SQLERROR GOTO 1000
C
      WRITE (*,*) 'COMMIT WORK'
      EXEC SQL COMMIT WORK
      GOTO 1100
C
1000  CALL SQLStatusCheck
      CALL TerminateProgram
C
1100  RETURN
      EXEC SQL WHENEVER SQLERROR CONTINUE
      END
C     (* End EndTransaction Subroutine *)
```

Figure 5-1. Program forex5: Implicit and Explicit Status Checking (page 3 of 8)

```
C
C
      SUBROUTINE TerminateProgram
C     (* Subroutine to Release PartsDBE *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION
C
      WRITE (*,*) 'RELEASE PartsDBE'
      EXEC SQL RELEASE
      WRITE (*,*) 'Terminating Program'
      RETURN
      END
C     (* End ReleaseDBE Subroutine *)
C
C
      SUBROUTINE SelectQuery
C     (* Subroutine to prompt user for Query Input *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
      LOGICAL              SQLCommandDone
      CHARACTER*16         response
      INTEGER              trycounter
      INTEGER              multiplerows
      INTEGER              deadlock
      INTEGER              OK
      INTEGER              notfound
C
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*16         PartNumber
      CHARACTER*30         PartName
      DOUBLE PRECISION     SalesPrice
      SQLIND               SalesPriceInd
      EXEC SQL END DECLARE SECTION
```

Figure 5-1. Program forex5: Implicit and Explicit Status Checking (page 4 of 8)

```
C
      EXEC SQL WHENEVER SQLERROR GOTO 2000
      EXEC SQL WHENEVER SQLWARNING GOTO 3000
      EXEC SQL WHENEVER NOT FOUND GOTO 4000
C
      trycounter = 0
      multiplerows = -10002
1000  CONTINUE
      DO WHILE (PartNumber .NE. '/')
         SQLCommandDone = .TRUE.
         WRITE (*,100)
100      FORMAT(/$,' Enter PartNumber from Parts table or / to STOP > ')
         READ (5,110) PartNumber
110      FORMAT(A16)
         IF (PartNumber .NE. '/') THEN
            CALL BeginTransaction
C
            DO WHILE (SQLCommandDone)
C
               WRITE (*,*) 'SELECT PartNumber, PartName, SalesPrice'
C
               EXEC SQL SELECT  PartNumber, PartName, SalesPrice
     1              INTO :PartNumber,
     2                   :PartName,
     3                   :SalesPrice :SalesPriceInd
     4              FROM  PurchDB.Parts
     5             WHERE  PartNumber = :PartNumber
C
               SQLCommandDone = .FALSE.
               CALL DisplayRow (PartNumber,PartName,SalesPrice,
     1                          SalesPriceInd)
            END DO
            CALL EndTransaction
         ENDIF
      END DO
      GOTO 5000
C
```

Figure 5-1. Program forex5: Implicit and Explicit Status Checking (page 5 of 8)

```
 2000  IF (SQLCode .EQ. multiplerows) THEN
       WRITE (*,*) 'WARNING: More than one row qualifies!'
       ENDIF
       CALL SQLStatusCheck (trycounter)
       CALL DisplayRow (PartNumber,PartName,SalesPrice,SalesPriceInd)
       CALL EndTransaction
       GOTO 1000
 C
 3000  WRITE (*,*) 'An SQL WARNING has occurred. The following row'
       WRITE (*,*) 'of data may not be valid! '
       CALL DisplayRow (PartNumber,PartName,SalesPrice,SalesPriceInd)
       CALL EndTransaction
       GOTO 1000
 C
 4000  WRITE (*,*) 'Row not found!'
       CALL EndTransaction
       GOTO 1000
 C
 5000  RETURN
       EXEC SQL WHENEVER SQLERROR CONTINUE
       EXEC SQL WHENEVER SQLWARNING CONTINUE
       EXEC SQL WHENEVER NOT FOUND CONTINUE
       END
 C
 C     (* End QueryTable Subroutine *)
 C
 C
       SUBROUTINE SQLExplain
 C     (* Subroutine to CALL SQLExplain *)
 C
       EXEC SQL INCLUDE SQLCA
 C
 C            (* Begin SQL Communication Area *)
 C
 C            (* Begin Host Variable Declarations *)
 C
       EXEC SQL BEGIN DECLARE SECTION
       CHARACTER*80   SQLMessage
       EXEC SQL END DECLARE SECTION
 C
       EXEC SQL SQLEXPLAIN :SQLMessage
       WRITE (*,*) SQLMessage
 C
       RETURN
       END
 C
 C     (* End SQLExplain Subroutine *)
```

Figure 5-1. Program forex5: Implicit and Explicit Status Checking (page 6 of 8)

```
      SUBROUTINE SQLStatusCheck (trycounter)
C     (* Subroutine to Check for DeadLocks *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
      LOGICAL              SQLCommandDone
      LOGICAL              Abort
      INTEGER              deadlock
      INTEGER              trycounter
      INTEGER              trycounterlimit
C
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*80   SQLMessage
      EXEC SQL END DECLARE SECTION
C
      deadlock = -14024
      trycounterlimit = 3
      SQLCommandDone = .FALSE.
C
      IF (SQLCode .EQ. deadlock) THEN
         IF (trycounter .EQ. trycounterlimit) THEN
            SQLCommandDone = .TRUE.
            WRITE (*,*) 'Deadlock occurred. You may want to try again'
      ELSE
            trycounter = trycounter + 1
            SQLCommandDone = .FALSE.
      ENDIF
      ENDIF
      Abort = .FALSE.
      IF (SQLCode .LT. deadlock) THEN
         Abort = .TRUE.
      ENDIF
      DO WHILE (SQLCode .NE. 0)
         CALL SQLExplain
      END DO
C
      IF (Abort) THEN
         CALL TerminateProgram
      ENDIF
C
      RETURN
      END
C
C     (* End DeadLockCheck Subroutine *)
```

Figure 5-1. Program forex5: Implicit and Explicit Status Checking (page 7 of 8)

```
C
C
      SUBROUTINE DisplayRow (PartNumber,PartName,SalesPrice,
     1SalesPriceInd)
C     (* Subroutine to Display a Selected Row *)
C
      EXEC SQL INCLUDE SQLCA
C
C            (* Begin SQL Communication Area *)
C
C            (* Begin Host Variable Declarations *)
C
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*16       PartNumber
      CHARACTER*30       PartName
      DOUBLE PRECISION   SalesPrice
      SQLIND             SalesPriceInd
      CHARACTER*80       SQLMessage
      EXEC SQL END DECLARE SECTION
C
      WRITE(*,100) PartNumber
      WRITE(*,110) PartName
C
C     IF (SalesPriceInd .LT. 0) THEN
         IF (SalesPrice .LT. 0) THEN
           WRITE (*,*) 'Sales Price is NULL'
         ELSE
           WRITE(*,120) SalesPrice
         ENDIF
      ENDIF
100   FORMAT('    Part Number:     ',A16)
110   FORMAT('    Part Name:       ',A30)
120   FORMAT('    Sales Price:     ',F10.2)
C
      WRITE (*,*) 'Was retrieved from the PurchDB.Parts table'
C
      RETURN
      END
C     (* End DisplayRow Subroutine *)
```

Figure 5-1. Program forex5: Implicit and Explicit Status Checking (page 8 of 8)

Explicit Status Checking

The example examined under "Implicit Status Checking" has already illustrated several uses for explicit status checking:

```
      PROGRAM SQLError
      :
C     (* Restart/Reentry point *)
600   CONTINUE
      .
      .  SQL SELECT Command
      .
      IF (SQLCode .EQ. MultipleRows) THEN
        WRITE(6,602) 'WARNING:  More than one row qualifies.'
602   FORMAT(A80)
      ELSE
      CALL SQLStatusCheck (trycounter)
      ENDIF
      CALL DisplayRow (PartNumber,PartName,SalesPrice,SalesPriceInd)
      CALL EndTransaction
      GOTO 600
      :
      END
C
      SUBROUTINE SQLStatusCheck (trycounter)
           :
           IF (SQLCode .EQ. deadlock) THEN
             IF (TryCounter .EQ. TryCounterLimit) THEN
               WRITE(6,102) 'Deadlock occurred, you may want to try again.'
102            FORMAT(A80)
               CALL TerminateProgram
             ELSE
               trycounter = trycounter + 1
             ENDIF
           ENDIF
           Abort = .FALSE.
             IF (SQLCode .LT. deadlock) THEN
               Abort = .TRUE.
             ENDIF
           DO WHILE (SQLCode .NE. 0)
             CALL SQLExplain :SQLMessage
             CALL WriteOut (SQLMessage)
           END DO
           IF (Abort) THEN
              CALL TerminateProgram
           ENDIF
      :
      RETURN
      END
```

SQLCA values are explicitly examined in this example in order to:

- Isolate errors so critical that they caused ALLBASE/SQL to rollback the current transaction.
- Control the number of times SQLEXPLAIN is executed.
- Detect when more than one row qualifies for the SELECT operation.
- Detect when a deadlock condition exists and control program execution.

This section examines when you may want to invoke such status-checking code routines *explicitly* rather than *implicitly*. In addition, this section illustrates how SQLErrd(3) and several SQLCode values can be explicitly used to monitor the number of rows operated on by data manipulation commands.

Explicitly Invoking Status-Checking Subprogram Units

The example in Figure 5-1 illustrates how status-checking code can be consolidated within individual subprogram units. This approach 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* places in the program after execution of such a subprogram unit increases. In this case, you invoke the subprogram units *after explicitly checking SQLCA values* rather than using the WHENEVER command to *implicitly* check these values.

The example shown in Figure 5-2 contains four data manipulation operations: INSERT, UPDATE, DELETE, and SELECT. Each of these operations is executed from its own subprogram unit.

As in the program in Figure 5-1, one subprogram unit is used for explicit error handling: SQLStatusCheck. Unlike the program in Figure 5-2; however, this subprogram unit is invoked after *explicit* test of SQLCode is made, immediately following each data manipulation operation. In the program in Figure 5-2, tests for warning conditions are omitted.

Because error handling is performed in a subprogram unit rather than in a code routine following the embedded SQL command, control returns to the point in the program where SQLStatusCheck is invoked.

```
PROGRAM Main
   .
   .
   .
   CALL SelectActivity
   .
   .
   .
   STOP
   END

   SUBROUTINE SelectActivity

        This subprogram unit prompts for a number that indicates
     whether the user wants to SELECT, UPDATE, DELETE,
     or INSERT rows, then invokes the subprogram unit that
     accomplishes the selected activity.  The DONE flag
     is set when the user enters a slash.
   .
   .
   .
   RETURN
   END

   SUBROUTINE InsertData
   .
   .
   .
Statements that accept data from the user appear here.

       EXEC SQL INSERT
   1             INTO PurchDB.Parts (PartNumber,
   2                                 PartName,
   3                                 SalesPrice)
   4                          VALUES (:PartNumber,
   5                                  :PartName,
   6                                  :SalesPrice)

       IF (SQLCode .NE. OK) THEN
          CALL SQLStatusCheck                             3
       ENDIF
   .
   .
   .
   RETURN
   END
   SUBROUTINE UpdateData
```

Figure 5-2. Explicitly Invoking Status-Checking Subprogram Units

```
     .
     .
     .
       This subprogram unit verifies that the row(s) to be changed
     exist, then invokes subprogram unit DisplayUpdate to accept
     new data from the user.

         EXEC SQL SELECT  PartNumber, PartName, SalesPrice
     1             INTO :PartNumber,
     2                  :PartName,
     3                  :SalesPrice
     4             FROM  PurchDB.Parts
     5            WHERE  PartNumber = :PartNumber

         IF (SQLCode .EQ. OK) THEN
           CALL DisplayUpdate
         ELSE
         IF (SQLCode .EQ. MultipleRows) THEN
             WRITE(6,102) 'Warning; more than one row qualifies!'
102          FORMAT (A80)
             CALL DisplayUpdate
         ELSE
             IF (SQLCode .EQ. NotFound) THEN                          5
               WRITE (6,103) 'Row not found!'
103            FORMAT (A80)
             ELSE
               CALL SQLStatusCheck                                    3
             ENDIF
           ENDIF
         ENDIF
     .

     SUBROUTINE DisplayUpdate
     .
     .
     .
  Statements that prompt user for new data appear here.
         EXEC SQL UPDATE PurchDB.Parts
     1              SET PartName = :PartName,
     2                  SalesPrice = :SalesPrice,
     3            WHERE PartNumber = :PartNumber

      IF (SQLCode .NE. OK)  THEN                                      3
        CALL SQLStatusCheck
      ENDIF
     .
```

Figure 5-2. Explicitly Invoking Status-Checking Subprogram Units (page 2 of 5)

```
    .
    .
    RETURN
    END

    SUBROUTINE DeleteData
    .
    .
    .
```

This subprogram unit verifies that the row(s) to be deleted exist, then invokes subprogram unit DisplayDelete to delete the row(s).

```
        EXEC SQL SELECT PartNumber, PartName, SalesPrice
    1             INTO :PartNumber,
    2                  :PartName,
    3                  :SalesPrice
    4              FROM PurchDB.Parts
    5             WHERE PartNumber = :PartNumber

        IF (SQLCode .EQ. OK) THEN
          CALL DisplayDelete
        ELSE
          IF (SQLCode .EQ. MultipleRows) THEN
            WRITE(6,102) 'Warning; more than one row qualifies!'
102         FORMAT(A80)
            CALL DisplayDelete
          ELSE
            IF (SQLCode = NotFound) THEN                              5
              WRITE (6,103) 'Row not found!'
103           FORMAT(A80)
            ELSE
              CALL SQLStatusCheck                                     3
            ENDIF
          ENDIF
        ENDIF
    .
    .
    .
    RETURN
    END

    SUBROUTINE DisplayDelete
    .
    .
    .
```

Figure 5-2. Explicitly Invoking Status-Checking Subprogram Units (page 3 of 5)

Statements that verify that the deletion should actually occur appear here.

```
       EXEC SQL DELETE FROM PurchDB.Parts
      1               WHERE PartNumber = :PartNumber

       IF (SQLCode .NE. OK) THEN                                    3
            CALL SQLStatusCheck
        ENDIF
        .
        .
        .
        RETURN
        END

        SUBROUTINE SelectData
        .
        .
        .
```

Statements that prompt for a partnumber appear here.

```
          EXEC SQL SELECT PartNumber, PartName, SalesPrice
      1             INTO :PartNumber,
      2                  :PartName,
      3                  :SalesPrice
      4              FROM PurchDB.Parts
      5             WHERE PartNumber = :PartNumber

          IF (SQLCode .EQ. OK) THEN
             CALL DisplayRow
          ELSE
            IF (SQLCode .EQ. MultipleRows) THEN
              WRITE(6,102) 'Warning; more than one row qualifies!'
 102          FORMAT(A80)
            ELSE
              IF (SQLCode = NotFound) THEN                          5
                WRITE (6,103) 'Row not found!'
 103            FORMAT(A80)
              ELSE
                CALL SQLStatusCheck                                 3
              ENDIF
            ENDIF
          ENDIF
         .
         .
        RETURN
        END
```

Figure 5-2. Explicitly Invoking Status-Checking Subprogram Units (page 4 of 5)

```
      SUBROUTINE SQLStatusCheck
       .
       .
       .
          IF (SQLCode .EQ. DeadLock) THEN
            IF (trycounter .EQ. trycounterlimit) THEN
              WRITE(6,102) 'Deadlock occurred; you may want to try again.'
102           FORMAT(A80)
              CALL EndTransaction
            ELSE
              trycounter = trycounter + 1
            ENDIF
          ENDIF
            Abort = .FALSE.
            IF (SQLCode .LT. DeadLock) THEN
              Abort = .TRUE.
            ENDIF
            DO WHILE (SQLCode .NE. 0)
              EXEC SQL SQLEXPLAIN :SQLMessage
              CALL WriteOut (SQLMessage)
            END DO
              IF (Abort) THEN
                CALL TerminateProgram
              ENDIF
       .
       .
       .
       RETURN
       END
```

Figure 5-2. Explicitly Invoking Status-Checking Subprogram Units (page 5 of 5)

Explicitly Checking for Number of Rows

SQLErrd(3) is useful in determining how many rows were processed in one of the following operations when the operation could be executed without error:

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

The SQLErrd(3) value can be used in these cases only when SQLCode does not contain a negative number. When SQLCode is 0, SQLErrd(3) is *always equal to 1* for SELECT, FETCH, UPDATE WHERE CURRENT, and DELETE WHERE CURRENT operations. SQLErrd(3) may be *greater than 1* if more than one row qualifies for an INSERT, UPDATE, or DELETE operation. When SQLCode is 100, SQLCA.SQLErrd(3) is 0.

The remainder of this chapter examines techniques for explicitly checking SQLErrd(3) as well as using SQLCodes of 100 and -10002 in data manipulation logic.

Using SQLErrd(3) for UPDATE and DELETE Operations. The example in Figure 5-3 could be modified to display the number of rows updated or deleted by using SQLErrd(3). In the case of the update operation, for example, the actual number of rows updated could be displayed after the UPDATE command is executed:

```
       SUBROUTINE DisplayUpdate
       .
       .
       EXEC SQL INCLUDE SQLCA
C
       INTEGER          OK
       INTEGER          NumberOfRows
C
       EXEC SQL BEGIN DECLARE SECTION
       CHARACTER*16     PartNumber
       CHARACTER*30     PartName
       DOUBLE PRECISION SalesPrice
       EXEC SQL END DECLARE SECTION
       .
       .
```

Statements that prompt user for new data appear here.

```
           EXEC SQL UPDATE PurchDB.Parts
     1                 SET PartName   = :PartName,
     2                     SalesPrice = :SalesPrice,
     3               WHERE PartNumber = :PartNumber
           IF (SQLCode .EQ. OK) THEN
             NumberOfRows = SQLErrd(3)
             WRITE(6,102) 'The number of rows updated was: ', NumberOfRows
102          FORMAT(A80,I)
           ELSE
             WRITE(6,103) 'No rows could be updated!'
103          FORMAT(A80)
             CALL SQLStatusCheck
           ENDIF
       .
       .
       RETURN
       END
```

If the UPDATE command is successfully executed, SQLCode equals zero and SQLErrd(3) contains the number of rows updated. If the UPDATE command cannot be successfully executed, SQLCode contains a negative number and SQLErrd(3) contains a zero.

In the case of the delete operation, the actual number of rows deleted could be displayed after the DELETE command is executed:

```
       SUBROUTINE DisplayDelete
       .
       .
       EXEC SQL INCLUDE SQLCA
C
       INTEGER           OK
       INTEGER           NumberOfRows
       CHARACTER         response
C
       EXEC SQL BEGIN DECLARE SECTION
       CHARACTER*16      PartNumber
       CHARACTER*30      PartName
       DOUBLE PRECISION  SalesPrice
       EXEC SQL END DECLARE SECTION
       .
       .
```

Statements that verify that the deletion should actually occur appear here.

```
           EXEC SQL DELETE FROM PurchDB.Parts
     1               WHERE PartNumber = :PartNumber

           IF (SQLCode .EQ. OK) THEN
             NumberOfRows = SQLErrd(3)
             WRITE(6,102) 'The number of rows deleted was: ', NumberOfRows
102          FORMAT(A35,I)
             WRITE(6,103) 'Do you want to COMMIT WORK? Y or N:'
103          FORMAT(A80)
             READ(5,104) response
104          FORMAT(A1)
             IF (response .EQ. 'Y') THEN
               EXEC SQL COMMIT WORK
             ELSE
               EXEC SQL ROLLBACK WORK
             ENDIF
           ELSE
             CALL SQLStatusCheck
           ENDIF
       .
       .
       RETURN
       END
```

If the DELETE command is successfully executed, SQLCode equals 0 and SQLErrd(3) contains the number of rows deleted. If the DELETE command cannot be successfully executed, SQLCode contains a negative number and SQLErrd(3) contains a 0.

Using SQLCode of 100. The programs already examined in this chapter have illustrated how an SQLCode of *100* can be detected and handled for data manipulation commands that do not use a cursor. When a cursor is being used, this SQLCode value is used to determine when all rows in an active set have been fetched:

```
          SUBROUTINE Cursor
          .
          .
          EXEC SQL INCLUDE SQLCA
   C
          INTEGER           OK
          INTEGER           NotFound
          LOGICAL           donefetch
   C
          EXEC SQL BEGIN DECLARE SECTION
          CHARACTER*16      PartNumber
          CHARACTER*30      PartName
          DOUBLE PRECISION  SalesPrice
          EXEC SQL END DECLARE SECTION
          .
          .
          CALL DeclareCursor
   C
          EXEC SQL OPEN Cursor1
          .
          .
          DO WHILE (donefetch)
            CALL FetchRow (donefetch)
          END DO
          .
          .
          RETURN
          END

          SUBROUTINE FetchRow (donefetch)
          .
          .
             EXEC SQL FETCH  Cursor1
        1              INTO :PartNumber,
        2                   :PartName,
        3                   :SalesPrice

             IF (SQLCode .EQ. OK) THEN
               CALL DisplayRow
             ELSE
               IF (SQLCode .EQ. NotFound) THEN
                 donefetch = .FALSE.
                 WRITE(6,102) ' '
```

```
              WRITE(6,102) 'Row not found or no more rows!'
102           FORMAT(A80)
            ELSE
              CALL DisplayError
            ENDIF
           ENDIF
       .
       .
       RETURN
       END
```

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; after the last row has been fetched from the active set the next attempt to FETCH sets SQLCode to a value of *100*. If no rows qualify for the active set, SQLCode equals 100 the first time subprogram unit *FetchRow* is executed.

Using SQLCode of -10002. If more than one row qualifies for a SELECT or FETCH operation, ALLBASE/SQL sets SQLCode to -10002. The program in Figure 5-3 contains an explicit test for this value. When SQLCode is equal to MultipleRows (defined as -10002 in the Type Declaration Section), a status checking subprogram unit is not invoked, but a warning message is displayed:

```
      SUBROUTINE UpdateData
       .
       .
       EXEC SQL INCLUDE SQLCA
C
```

```
       INTEGER           OK
       INTEGER           NotFound
       INTEGER           MultipleRows
       LOGICAL           donefetch
C
       EXEC SQL BEGIN DECLARE SECTION
       CHARACTER*16      PartNumber
       CHARACTER*30      PartName
       DOUBLE PRECISION  SalesPrice
       EXEC SQL END DECLARE SECTION
C
       OK = 0
       NotFound = 100
       MultipleRows = -10002
       .
       .
```

This subprogram unit verifies that the row(s) to be changed exists, then invokes subprogram unit DisplayUpdate to accept new data from the user.

```
       .
       .
          EXEC SQL SELECT  PartNumber, PartName, SalesPrice
     1               INTO :PartNumber,
     2                    :PartName,
     3                    :SalesPrice
     4               FROM  PurchDB.Parts
     5              WHERE  PartNumber = :PartNumber

          IF (SQLCode .EQ. OK) THEN
            CALL DisplayUpdate
          ELSE
            IF (SQLCode .EQ. MultipleRows) THEN
              WRITE(6,102) ' '
              WRITE(6,102) 'Warning; more than one row will be changed!'
102           FORMAT(A80)
              CALL DisplayUpdate
            ELSE
              IF (SQLCode .EQ. NotFound) THEN
                WRITE(6,103) ' '
                WRITE(6,103) 'Row not found.')
103             FORMAT(A80)
              ELSE
                CALL SQLStatusCheck
              ENDIF
            ENDIF
          ENDIF
      .
      .
      RETURN
      END
```

6

Overview of Data Manipulation

To manipulate data in an ALLBASE/SQL DBEnvironment, you use one of the following SQL commands:

- SELECT: to retrieve one or more rows from one or more tables.
- INSERT: to insert one or more rows into a single table.
- DELETE: to delete one or more rows from a single table.
- UPDATE: to change the value of one or more columns in one or more rows in a single table.

Three techniques exist for using these commands in a program:

- In **simple data manipulation**, you retrieve or insert *single rows* or you delete or update one or more rows based on a *specific criterion*.
- In **sequential table processing**, you operate on a *set of rows*, one row at a time, using a cursor. A **cursor** is a pointer that identifies one row in the set of rows, called the **active set**. You move through the active set, retrieving a row at a time and optionally updating or deleting it.
- In **dynamic operations**, you preprocess SQL commands *at runtime*, as when the program accepts data manipulation commands from a user.

Table 6-1 summarizes which data manipulation commands can be used in each technique. Note that the FETCH command is included in this table, since it must be used when you manipulate data using a cursor.

Table 6-1. How Data Manipulation Commands May Be Used

	USABLE SQL COMMANDS						
TYPE OF OPERATION	**SELECT**	**FETCH**	**INSERT**	**DELETE**	**UPDATE**	**DELETE WHERE CURRENT**	**UPDATE WHERE CURRENT**
Simple	X		X	X	X		
Sequential	X	X				X	X
Dynamic		X	X	X	X		

The remainder of this chapter briefly examines each of the three data manipulation techniques (each technique is discussed in detail in Chapters 7 through 9) and introduces the use of a cursor for data manipulation. First, however, this chapter addresses the **query**, or the description of data you want to retrieve. Queries are fundamental to ALLBASE/SQL data manipulation because some of the elements of a query are also used to describe and limit data when you update or delete it. In addition, it is common programming practice to retrieve and display rows prior to changing or deleting them.

The Query

A query is a SELECT command that describes to ALLBASE/SQL the data you want retrieved. You can retrieve all or only certain data from a table. You can have ALLBASE/SQL group or order the rows you retrieve or perform certain calculations or comparisons before presenting data to your program. You can retrieve data from multiple tables. You can also retrieve data using views or combinations of tables and views.

The SELECT Command

The SELECT command identifies the columns and rows you want in your query result as well as the tables and views to use for data access. The columns are identified in the **select list**. The rows are identified in several **clauses** (GROUP BY, HAVING, and ORDER BY). The tables and views to access are identified in the FROM clause. Data thus specified is returned into host variables named in the INTO clause:

```
      EXEC SQL     SELECT SelectList
     1                INTO HostVariables
     2                FROM TableNames
     3               WHERE SearchCondition1
     4            GROUP BY ColumnName
     5              HAVING SearchCondition2
     6            ORDER BY ColumnID
```

To retrieve *all* data from a table, the SELECT command need specify only the following:

```
      EXEC SQL SELECT  *
     1           INTO :HostVariable1
     2                :HostVariable2
     3                 .
     4                 .
     5                 .
     6           FROM  OwnerName.TableName
```

Although the shorthand notation * can be used in the select list to indicate you want *all columns* from one or more tables or views, it is better programming practice to explicitly name columns. Then, if the tables or views referenced are altered, your program will still retrieve only the data its host variables are designed to accommodate:

```
     EXEC SQL SELECT  PartNumber,
   1                   PartName,
   2                   SalesPrice
   3             INTO :PartNumber,
   4                  :PartName,
   5                  :SalesPrice
   6             FROM  PurchDB.Parts
```

The SELECT command has several clauses you can use to format the data retrieved from any table:

- the WHERE clause specifies a search condition. A **search condition** consists of one or more predicates. A **predicate** is a test each row must pass before it is returned to your program.
- the GROUP BY clause and the HAVING clause tell ALLBASE/SQL how to group rows retrieved before applying any aggregate function in the select list to each group of rows.
- the ORDER BY clause causes ALLBASE/SQL to return rows in ascending or descending order, based on the value in one or more columns.

The following SELECT command contains a WHERE clause that limits rows returned to those not containing a salesprice; the predicate used in the WHERE clause is known as the **null predicate**:

```
     EXEC SQL SELECT  PartName,
   1                   SalesPrice
   2             INTO :PartName,
   3                  :SalesPrice
   4             FROM  PurchDB.Parts
   5            WHERE  SalesPrice IS NULL
```

In the UPDATE and DELETE commands, you may need a WHERE clause to limit the rows ALLBASE/SQL changes or deletes. In the following case, the sales price of parts priced lower than $1000 is increased 10 percent; the WHERE clause in this case illustrates the **comparison predicate**:

```
EXEC SQL UPDATE PurchDB.Parts
                SET SalesPrice = SalesPrice * 1.1
              WHERE SalesPrice < 1000.00
```

The *ALLBASE/SQL Reference Manual* details the syntax and semantics for these and other predicates.

When you use an *aggregate function* in the select list, you can use the GROUP BY clause to indicate how ALLBASE/SQL should group rows before applying the function. You can also use the HAVING clause to limit the groups to only those satisfying certain criteria. The following SELECT command will produce a query result containing two columns: a sales price and a number indicating how many parts have that price:

```
EXEC SQL SELECT  SalesPrice,
                  COUNT(PartNumber)
            INTO  :SalesPrice,
                  :Count
            FROM  PurchDB.Parts
        GROUP BY  SalesPrice
          HAVING  AVG(SalesPrice) > 1500.00
```

The GROUP BY clause in this example causes ALLBASE/SQL to group all parts with the same sales price together. The HAVING clause causes ALLBASE/SQL to ignore any group having an average sales price less than or equal to $1500.00. Once the groups have been defined, ALLBASE/SQL applies the aggregate function *COUNT* to each group.

Each null value in a GROUP BY column constitutes a separate group. Therefore a query result having a null value in the column(s) used to group rows would contain a separate row for each null value.

An aggregate function is one example of an ALLBASE/SQL expression. An *expression* specifies a *value*. An expression can be used in several places in the SELECT command as well as in the other data manipulation commands. Refer to the *ALLBASE/SQL Reference Manual* for the syntax and semantics of expressions, as well as the effect of null values on them.

The rows in the query result obtained with the preceding query could be returned in a specific order by using the ORDER BY clause. In the following case, the rows are returned in *descending sales price order*:

```
EXEC SQL SELECT   SalesPrice,
                   COUNT(PartNumber)
              INTO :SalesPrice,
                   :Count
              FROM  PurchDB.Parts
          GROUP BY  SalesPrice
            HAVING  AVG(SalesPrice) > 1500.00
          ORDER BY  SalesPrice DESC
```

The examples shown so far have all included queries where results would most likely contain more than one row. The sequential table processing technique using cursors could also be used to handle multiple-row query results. Later in this chapter you'll find examples of this technique, as well as examples illustrating simple data manipulation, in which only one-row query results are expected.

Selecting from Multiple Tables

To retrieve data from more than one table or view, the query describes to ALLBASE/SQL how to *join* the tables before deriving the query result:

- In the FROM clause, you identify the tables and views to be joined.
- In the WHERE clause, you specify a *join condition*. A join condition defines the condition(s) under which *rows* should be joined.

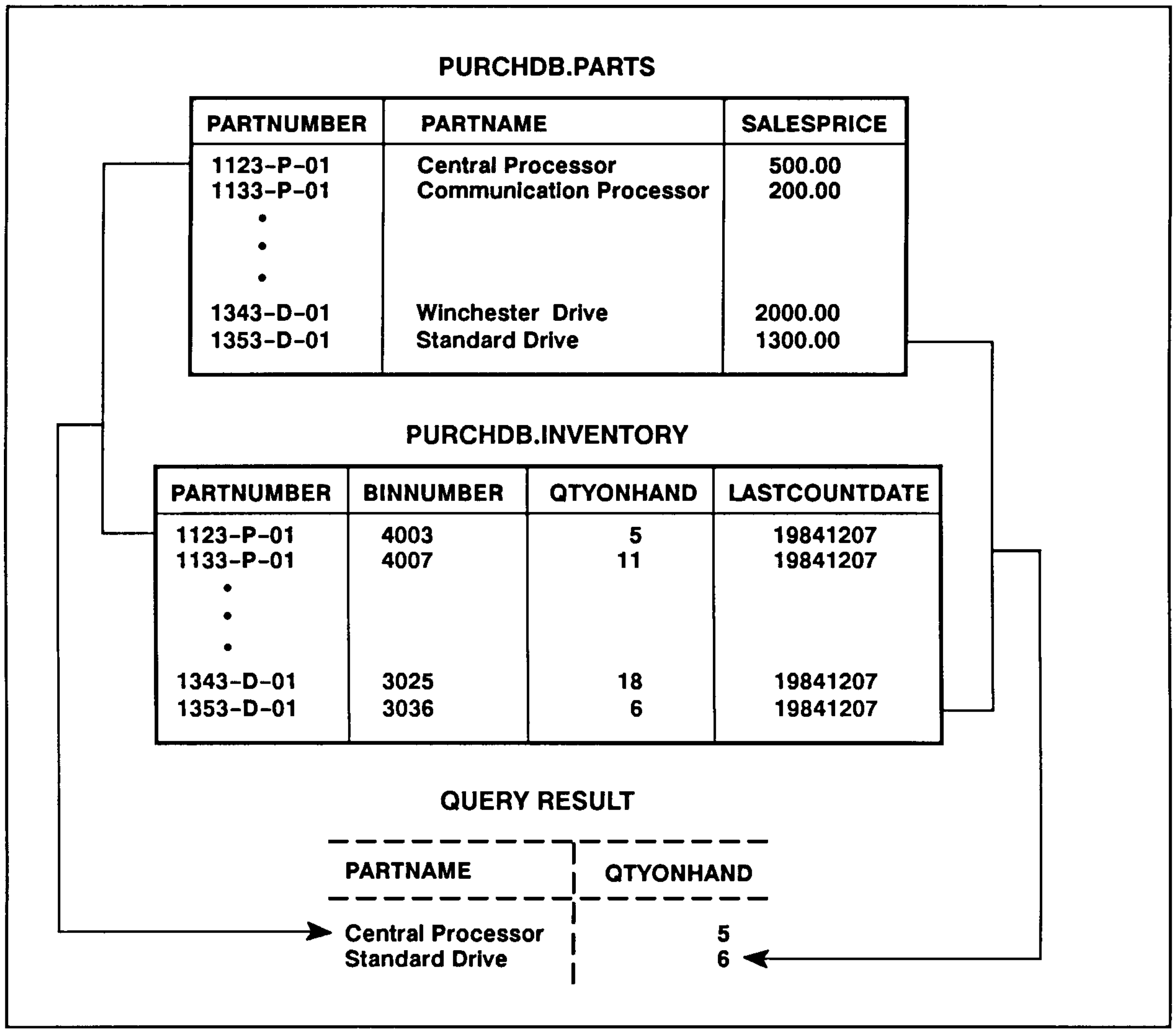
To obtain a query result consisting of the name of each part and its quantity-on-hand, you need data from two tables in the sample database: *PurchDB.Parts* and *PurchDB.Inventory*. The join condition in this case is that you want ALLBASE/SQL to join rows in these tables that have the same part number:

```
      EXEC SQL SELECT   PartName,
     1                   QtyOnHand
     2            INTO :PartName,
     3                 :QtyOnHand
     4            FROM  PurchDB.Parts,
     5                  PurchDB.Inventory
     6           WHERE  PurchDB.Parts.PartNumber =
     7                  PurchDB.Inventory.PartNumber
```

Whenever two or more columns in a query have the same name but belong to different tables, you avoid ambiguity by qualifying the column names with table and owner names. Because the columns specified in the join condition shown above have the same name (*PartNumber*) in both tables, they are fully qualified with table and owner names (*PurchDB.Parts* and *PurchDB.Inventory*). If one of the columns named *PartNumber* were named *PartNum*, the WHERE clause could be written without having the fully qualified column name as follows:

```
     WHERE PartNumber = PartNum
```

ALLBASE/SQL creates a row for the query result whenever the *PartNumber* value in one table matches that in the second table. Any row containing a null *PartNumber* is excluded from the join, as are rows that have a *PartNumber* value in one table, but not the other:



PURCHDB.PARTS

PARTNUMBER	PARTNAME	SALESPRICE
1123-P-01	Central Processor	500.00
1133-P-01	Communication Processor	200.00
•		
•		
•		
1343-D-01	Winchester Drive	2000.00
1353-D-01	Standard Drive	1300.00

PURCHDB.INVENTORY

PARTNUMBER	BINNUMBER	QTYONHAND	LASTCOUNTDATE
1123-P-01	4003	5	19841207
1133-P-01	4007	11	19841207
•			
•			
•			
1343-D-01	3025	18	19841207
1353-D-01	3036	6	19841207

QUERY RESULT

PARTNAME	QTYONHAND
Central Processor	5
Standard Drive	6

LG200125_009a

Figure 6-1. Sample Query Joining Multiple Tables

You can also join a table *to itself*. This type of join is useful when you want to identify *pairs of values* within one table that have certain relationships.

The *PurchDB.SupplyPrice* table contains the unit price, delivery time, and other data for every vendor that supplies any part. Most parts are supplied by more than one vendor, and prices vary with vendor. You can join the *PurchDB.SupplyPrice* table to itself in order to identify for which parts the difference among vendor prices is greater than $50. The query and its result would appear as follows:

The query:

```
      EXEC SQL SELECT  X.PartNumber,
     1                 X.VendorNumber,
     2                 X.UnitPrice,
     3                 Y.VendorNumber,
     4                 Y.UnitPrice
     5           INTO :PartNumber,
     6                :VendorNumber1,
     7                :UnitPrice1,
     8                :VendorNumber2,
     9                :UnitPrice2
     1           FROM  PurchDB.SupplyPrice X,
     2                 PurchDB.SupplyPrice Y
     3          WHERE  X.PartNumber = Y.PartNumber AND
     4                 X.UnitPrice  > (Y.UnitPrice + 50.00)
```

The result:

```
----------------+------------+--------------+------------+--------------
PARTNUMBER      |VENDORNUMBER|UNITPRICE     |VENDORNUMBER|UNITPRICE
----------------+------------+--------------+------------+--------------
1123-P-01       |        9007|        550.00|        9002|        450.00
1123-P-01       |        9012|        525.00|        9002|        450.00
1123-P-01       |        9007|        550.00|        9008|        475.00
1123-P-01       |        9007|        550.00|        9003|        475.00
1433-M-01       |        9007|        700.00|        9003|        645.00
1623-TD-01      |        9011|       1800.00|        9015|       1650.00
                |___________________________|
                             |
```

These vendors charge at least $50 more for a part than the vendors identified in the next two columns.

To obtain such a query result, ALLBASE/SQL joins one copy of the table with another copy of the table, using the join condition specified in the WHERE clause:

- You name each copy of the table in the FROM clause by using a join variable. In this example, the join variables are X and Y. Then you use the join variable to qualify column names in the select list and other clauses in the query.
- The join condition in this example specifies that for each part number, the query result should contain a row only when the price of the part from vendor to vendor differs by more than $50.

Join variables can be used in *any* query as a shorthand way of referring to a table, but they *must* be used in queries that join a table to itself so that ALLBASE/SQL can distinguish between the two copies of the table.

Selecting Using Views

Views are used to restrict data visibility as well as to simplify data access:

- Data visibility can be limited using views by defining them such that only certain columns and/or rows are accessible through them.
- Data access can be simplified using views by creating views based on joins or containing columns that are derived from expressions or aggregate functions.

The sample database has a view called *PurchDB.VendorStatistics*, defined as follows:

```
EXEC SQL CREATE VIEW  PurchDB.VendorStatistics
             (VendorNumber,
              VendorName,
              OrderDate,
              OrderQuantity,
              TotalPrice)
          AS
      SELECT  PurchDB.Vendors.VendorNumber,
              PurchDB.Vendors.VendorName,
              OrderDate,
              OrderQty,
              OrderQty * PurchasePrice
        FROM  PurchDB.Vendors,
              PurchDB.Orders,
              PurchDB.OrderItems
       WHERE  PurchDB.Vendors.VendorNumber    =
              PurchDB.Orders.VendorNumber    AND
              PurchDB.OrderItems.OrderNumber =
              PurchDB.OrderItems.OrderNumber
```

This view combines information from three base tables to provide a summary of data on existing orders with each vendor. One of the columns in the view consists of a computed expression: the total cost of an item on order with the vendor.

Note that the select list of the SELECT command defining this view contains some qualified and some unqualified column names. Columns *OrderDate*, *OrderQty*, and *PurchasePrice* need not be qualified, because these names are unique among the column names in the three tables joined in this view. In the WHERE clause, however, both join conditions must contain fully qualified column names since the columns are named the same in each of the joined tables.

You can use a view in a query without restriction. In the FROM clause, you identify the view as you would identify a table. When you reference columns belonging to the view, you use the column names used in the view definition. In the view above, for example, the column containing quantity-on-order is called *OrderQuantity*, not *OrderQty* as it is in the base table (*PurchDB.OrderItems*).

The *VendorStatistics* view can be used to quickly determine the total dollar amount of orders existing for each vendor. Because the view definition contains all the details for deriving this information, the query based on this view is quite simple:

```
 EXEC SQL SELECT  VendorNumber,
                   SUM(TotalPrice)
             INTO :VendorNumber,
                  :Sum
             FROM  PurchDB.VendorStatistics
         GROUP BY  VendorNumber
```

The query result appears as follows:

```
------------+---------------------
VENDORNUMBER|(EXPR)
------------+---------------------
 9001|             31300.00
 9002|              6555.00
 9003|              6325.00
 9004|              2850.00
 9006|              2010.00
 9008|             12460.00
 9009|              7750.00
 9010|              9180.00
 9012|             12280.00
 9013|              8270.00
 9014|              2000.00
 9015|             17550.00
```

Although you can use views in queries without restriction, you can use only some views to INSERT, UPDATE, or DELETE rows:

- You cannot INSERT, UPDATE, or DELETE using a view if the view definition contains one of the following:
  - Join operation
  - Aggregate function
  - DISTINCT option
  - GROUP BY clause
- You cannot INSERT using a view if any column of the view is computed in an arithmetic expression.

The *PurchDB.VendorStatistics* view cannot be used for any INSERT, UPDATE, or DELETE operation because it is based on a three-table join and contains a column (*TotalPrice*) derived from a multiplication operation.

Query Efficiency

Three clauses in the SELECT command have an effect on the execution speed of queries:

```
WHERE
GROUP BY
ORDER BY
```

As discussed earlier, the WHERE clause consists of one or more predicates. Predicates can be evaluated more quickly when they can be optimized by ALLBASE/SQL.

The following predicates are optimizable when all the data types within them are the same (in the case of DOUBLE PRECISION data, the precisions and scales of the different values must be the same). Note that after optimization, ALLBASE/SQL may perform an index scan to access data; an index scan improves data access speed by making use of an index on one or more of the columns in the predicate:

- WHERE *Column1 ComparisonOperator Column2* where *ComparisonOperator* is one of the following: =, >, >=, <, or <=. An index may be used if *Column1* and *Column2* are in different tables and an index exists on either column:

  ```
  WHERE PurchDB.Parts.PartNumber = PurchDB.SupplyPrice.PartNumber
  ```

- *WHERE* Column1 ComparisonOperator {*Constant* or *HostVariable*} where *ComparisonOperator* is one of the following: =, >, >=, <, or <=. An index may be used if one exists on *Column1*; however, an index may be used if a host variable appears in the predicate *only* if the comparison operator is an equal sign (=) :

  ```
  WHERE SupplyPrice = :SupplyPrice
  ```

- WHERE *Column1* BETWEEN {*Column2* or *Constant* or *HostVariable*} AND {*Column2* or *Constant* or *HostVariable*}. An index may be used if *Column1* is the *only* column name in the predicate and an index exists on it.
- WHERE *Column1* <> {*Column2* or *Constant* or *Host Variable*} Although this kind of predicate is optimizable, an index is never used:

  ```
  WHERE VendorState <> :VendorState
  ```

The lower the cluster count of an index, the greater the chance ALLBASE/SQL will use it when an appropriate index is available. *Cluster count* indicates the number of times ALLBASE/SQL has to access a *different* data page to retrieve the next row during an index scan. Refer to the *ALLBASE/SQL Database Administration Guide* for information on how to optimize the cluster count of an index.

The following predicates are not optimizable, and an index is never used:

- Predicates containing arithmetic expressions:

```
WHERE Column1 > Column2 * :HostVariable
```

- LIKE predicates:

```
WHERE Column1 LIKE :HostVariable
```

- Predicates joined by the logical operator OR:

```
WHERE Column1 = Column2
  OR Column1 > Constant
```

When a query does not contain a WHERE clause, an index is never used, because *all* rows from tables in the FROM clause containing columns in the select list qualify:

```
EXEC SQL SELECT  *
           INTO  :HostVariableList
                  .
                  .
                  .
           FROM   OwnerName.TableName
```

When an index is not used, ALLBASE/SQL performs what is known as a *serial scan* to locate rows. When a serial scan is performed instead of an index scan, the entire table is locked, regardless of the automatic locking mode of the table.

The optimization and locking ALLBASE/SQL performs for the WHERE clause in the SELECT command also applies to the WHERE clause in the UPDATE and DELETE commands.

When a query contains a GROUP BY and/or an ORDER BY clause, ALLBASE/SQL must sort rows. The time required for sorting increases as the number of qualifying rows increases.

Sorting occurs in DBEFiles associated with the SYSTEM DBEFileSet. Therefore enough file space must be available in this DBEFileSet when the query is executed to accommodate the sort operations. Guidelines on space requirements can be found in the *ALLBASE/SQL Database Administration Guide* .

Simple Data Manipulation

In simple data manipulation, you retrieve or insert single rows or update one or more rows based on a specific criterion. In most cases, the simple data manipulation technique is used to support the random retrieval and/or change of specific rows. The duration of locks can be minimized by making each data manipulation operation a separate transaction.

In the following example, if the user wants to perform a DELETE operation, the program performs the operation only if a *single row* qualifies. If no rows qualify or if more than one row qualifies, the program displays a message. Note that the host variables in this case are designed to accommodate only a single row. In addition, two of the columns may contain null values, so an indicator variable is used for these columns:

```
         EXEC SQL BEGIN DECLARE SECTION
         CHARACTER*16     PartNumber
         CHARACTER*30     PartName
         SQLIND           PartNameInd
         DOUBLE PRECISION SalesPrice
         SQLIND           SalesPriceInd
         EXEC SQL END DECLARE SECTION
         .
         .
         .
       SUBROUTINE DoQuery

 . This procedure accepts a part number from the user,
 .  then executes a query to determine whether one or
 .  more rows containing that value actually exist.

       EXEC SQL SELECT  PartNumber, PartName, SalesPrice
      1           INTO :PartNumber,
      2                :PartName :PartNameInd,
      3                :SalesPrice :SalesPriceInd
      4           FROM  PurchDB.Parts
      5          WHERE  PartNumber = :PartNumber

       IF (SQLCode .EQ. 0) THEN
         CALL DisplayDelete
       ELSEIF (SQLCode .EQ.100) THEN
         CALL WriteOut ('Row Not Found!')
       ELSEIF (SQLCode .EQ. -10002) THEN
         CALL WriteOut ('WARNING: More than one row qualifies!')
       ELSE
          CALL SQLStatusCheck
       ENDIF
       .
       .
       .
       RETURN
       END
      SUBROUTINE DisplayDelete

    . The qualifying row is displayed for the user to
    .  verify that it should be deleted before the following
    .  command is executed:

           EXEC SQL DELETE FROM PurchDB.Parts
     1                    WHERE PartNumber = :PartNumber
        .
        .
        .
      RETURN
      END
```

Chapter 7 provides more details about simple data manipulation.

Introducing the Cursor

You use a cursor to manage a query result that may contain more than one row when you want to make all the qualifying rows available to the program user. Cursors are used in sequential table processing as shown later in this chapter.

Like the cursor on a terminal screen, an ALLBASE/SQL cursor is a position indicator. It does not, however, point to a column. Rather, it points to one row in an active set. An **active set** is a query result obtained when a SELECT command associated with a cursor (defined in a DECLARE CURSOR command) is executed (using the OPEN command).

Each cursor used in a program must be declared before it is used. You use the DECLARE CURSOR command to declare a cursor. The DECLARE CURSOR command names the cursor and associates it with a particular SELECT command:

```
      EXEC SQL DECLARE Cursor1
     1           CURSOR FOR
     2           SELECT PartName,
     3                  SalesPrice
     4             FROM PurchDB.Parts
     5            WHERE PartNumber BETWEEN :LowValue AND :HighValue
     6         ORDER BY PartName
```

All cursor names within one program must be unique. You use a cursor name when you perform data manipulation operations using the cursor.

The SELECT command in the cursor declaration does not specify any *output host variables*. The SELECT command can, however, contain *input host variables*, as in the WHERE clause of the cursor declaration above.

Rows in the active set are returned to output host variables when the FETCH command is executed:

```
     EXEC SQL OPEN Cursor1
     .
     .       The OPEN command examines any input host
     .       variables and determines the active set.
             .
     EXEC SQL FETCH Cursor1 INTO OutputHostVariables

             .         The FETCH command delivers one row of
             .         the active set into output host variables.
             .
             .
```

If a *serial scan* will be used to retrieve the active set, ALLBASE/SQL locks the table(s) when the OPEN command is executed. If an *index scan* will be used, locks are placed when rows are fetched. Any locks obtained are held until the transaction terminates or the CLOSE command is executed.

Both the OPEN and the FETCH commands position the cursor:

- The OPEN command positions the cursor *before the first row* of the active set.
- The FETCH command advances the cursor to the next row of the active set and delivers that row to the output host variables.

The row at which the cursor points at any one time is called the **current row**. When a row is a current row, you can *delete* it as follows:

```
     EXEC SQL DELETE FROM PurchDB.Parts
    1                WHERE CURRENT OF Cursor1
```

When you delete the current row, the cursor remains between the row deleted and the next row in the active set until you execute the FETCH command again:

```
     EXEC SQL FETCH  Cursor1
    1           INTO :PartName :PartNameInd,
    2                :SalesPrice :SalesPriceInd
```

When a row is a current row you can update it *if* the cursor declaration contains a FOR UPDATE OF clause naming the column(s) you want to change. The following cursor, for example, can be used to update the *SalesPrice* column of the current row by using the WHERE CURRENT OF option in the UPDATE command:

```
 EXEC SQL DECLARE Cursor2
1          CURSOR FOR
2          SELECT PartName, SalesPrice
3            FROM PurchDB.Parts
4           WHERE PartNumber BETWEEN :LowValue AND :HighValue
5      FOR UPDATE OF SalesPrice
  .
  .      Because the DECLARE CURSOR command is not
  .      executed at runtime, no status checking code
  .      needs to appear here.
  .
 EXEC SQL OPEN Cursor2
  .
  .      The OPEN command examines any input host
  .      variables and determines the active set.
  .      Then the program fetches one row at a time.
  .
 EXEC SQL FETCH  Cursor2
1          INTO :PartName :PartNameInd,
2               :SalesPrice :SalesPriceInd
  .
  .      If the program user wants to change the SalesPrice
  .      of the row displayed (the current row), the UPDATE
  .      command is executed.  The new SalesPrice entered by
  .      the user is stored in an input host variable named
  .      NewSalesPrice.
         .
 EXEC SQL UPDATE PurchDB.Parts
1            SET SalesPrice = :NewSalesPrice
2          WHERE CURRENT OF Cursor2

  .      After the UPDATE command is executed, the updated
  .      row remains the current row until the FETCH command
  .      is executed again.
  .
```

The restrictions that govern deletions and updates using a view *also* govern deletions and updates using a cursor. You cannot delete or update a row using a cursor if the cursor declaration contains any of the following:

- Join operation
- Aggregate function
- DISTINCT
- GROUP BY
- ORDER BY
- UNION

After the last row in the active set has been fetched, the cursor is positioned *after the last row fetched* and the value in SQLCode is equal to 100. Therefore to retrieve *all* rows in the active set, you execute the FETCH command until SQLCode is not 0:

```
     DO WHILE (SQLCode .EQ. 0)
      EXEC SQL FETCH  Cursor3
    1          INTO :PartNumber,
    2               :PartName :PartNameInd,
    3               :SalesPrice :SalesPriceInd

      IF (SQLCode .EQ. 0) THEN
        CALL DisplayRow
      ELSEIF (SQLCA.SQLCode .EQ.100) THEN
        CALL WriteOut ('Row Not Found or No More Rows!')
       ELSE
        CALL SQLStatusCheck
      ENDIF
     END DO
```

When you are finished operating on an active set, you use the CLOSE command:

```
   EXEC SQL CLOSE Cursor3
```

When you close a cursor, the active set becomes undefined and you cannot use the cursor again unless you issue an OPEN command to reopen it. The COMMIT WORK and ROLLBACK WORK commands also close any open cursors, automatically. Figure 6-2 summarizes the effect of the cursor-related commands on the position of the cursor and on the active set.

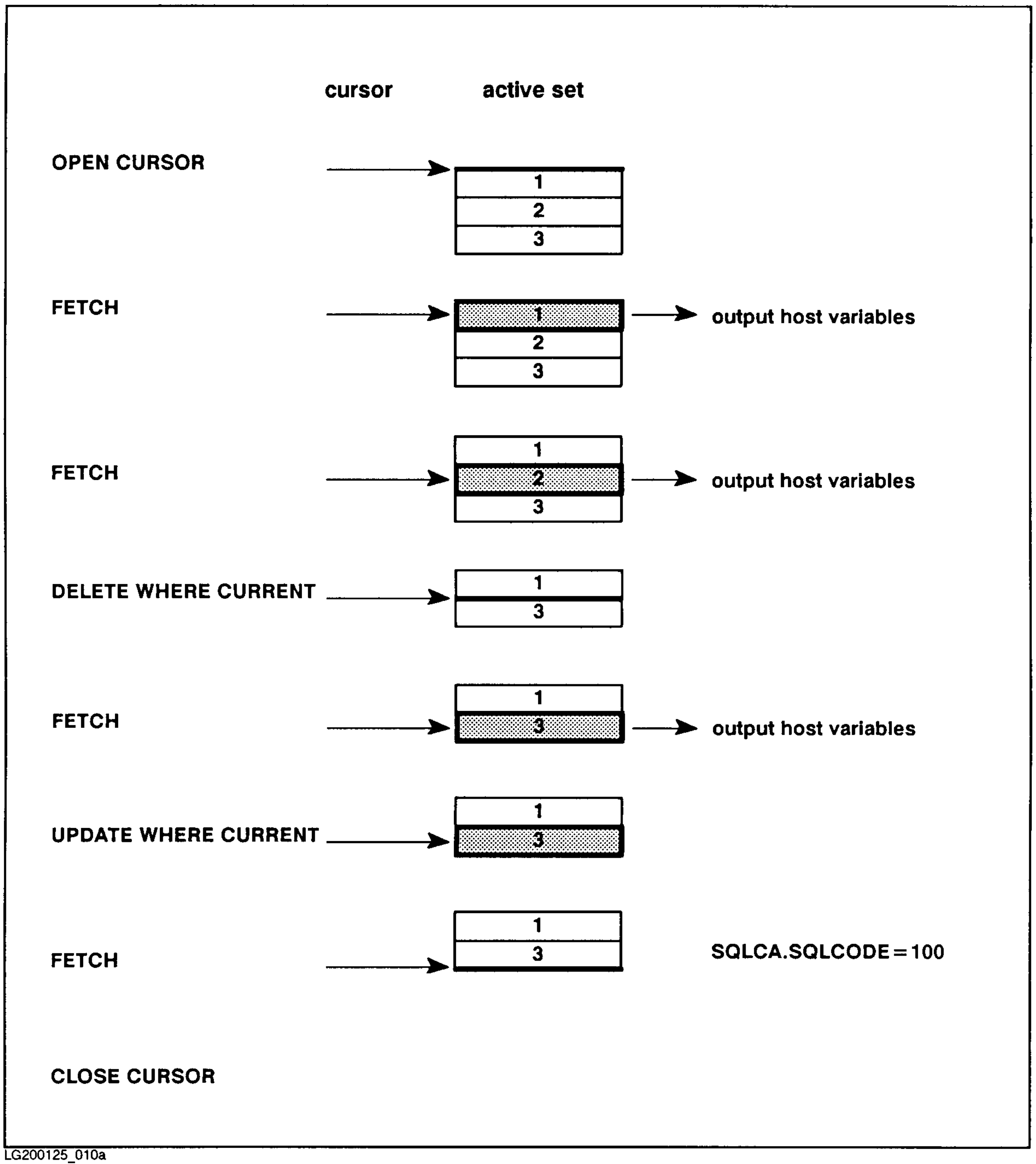


Figure 6-2. Effect of SQL Commands on Cursor and Active Sets

Sequential Table Processing

In sequential table processing, you process an active set by fetching a row at a time and optionally deleting or updating it. Sequential table processing is useful when the likelihood of row changes throughout a set of rows is high and when a program user does not need to review multiple rows to decide whether to change a specific row.

In the following example, rows for parts having the same *SalesPrice* are displayed one at a time. The program user can delete a displayed row or change its *SalesPrice*. Note that the host variable declarations are identical to those for the simple data manipulation example, since only one row at a time is fetched. Rows are fetched as long as SQLCode is equal to 0:

```
    SUBROUTINE GetActiveSet

        EXEC SQL INCLUDE SQLCA
        .
        .
        .
        CHARACTER        Response

        OK        =   0
        NotFound  = 100

        EXEC SQL BEGIN DECLARE SECTION
        CHARACTER*16     PartNumber
        CHARACTER*30     PartName
        SQLIND           PartNameInd
        DOUBLE PRECISION SalesPrice
        SQLIND           SalesPriceInd
        EXEC SQL END DECLARE SECTION
        .
        .
        .

     The cursor declared allows the user to change the SalesPrice of
     the current row.  It can also be used to delete the current row.

     EXEC SQL DECLARE PriceCursor
  1           CURSOR FOR
  2           SELECT PartNumber, PartName, SalesPrice
  3             FROM PurchDB.Parts
  4            WHERE SalesPrice = :SalesPrice
  5     FOR UPDATE OF SalesPrice
     .
     .      The program accepts a salesprice value from the user.
     .
     EXEC SQL OPEN PriceCursor

     IF (SQLCode .NE. OK) THEN
    CALL SQLStatusCheck
    CALL ReleaseDBE
  ELSE
    CALL GetRow
  ENDIF
  .
  .
  .
  RETURN
  END
  SUBROUTINE GetRow
  .
  .
  .
  DO WHILE (SQLCode .EQ. OK)
     EXEC SQL FETCH  PriceCursor
1             INTO :PartNumber,
2                  :PartName :PartNameInd,
3                  :SalesPrice :SalesPriceInd

  IF (SQLCode .EQ. OK) THEN
    CALL DisplayRow
  ELSEIF (SQLCode .EQ. NotFound) THEN
    CALL WriteOut ('No More Rows!')
  ELSE
    CALL SQLStatusCheck
  ENDIF
  END DO
  .
  .
  .
  RETURN
  END

 SUBROUTINE DisplayRow
```

. *Each row fetched is displayed. Depending on the user's response*
. *to a program prompt, the row may be deleted or its SalesPrice*
. *value changed.*

```
  IF (Response .EQ. 'D') THEN
    EXEC SQL DELETE FROM PurchDB.Parts
1             WHERE CURRENT OF PriceCursor
    .
```
. *Status checking code appears here.*

.

```
      ELSEIF (response .EQ. 'U') THEN
        .
        .   A new SalesPrice is accepted.
        .
        EXEC SQL UPDATE PurchDB.Parts
     1              SET SalesPrice = :SalesPrice
     2              WHERE CURRENT OF PriceCursor
```

```
          .
          .  Status checking code appears here.
          .
      ENDIF
      .
      .
      .
      RETURN
      END
```

More on sequential table processing can be found in Chapter 8.

Dynamic Operations

Dynamic operations offer a way to execute SQL commands that cannot be completely defined until runtime. You accept part or all of an SQL command that can be dynamically preprocessed from the user, then use one of the following techniques to preprocess and execute the command:

- You can use the PREPARE command to preprocess it, then execute it later during the same transaction using the EXECUTE command.
- You can use the EXECUTE IMMEDIATE command to preprocess *and* execute the dynamic command in one step.

FORTRAN does not support dynamic queries; only dynamic non-queries may be dynamically prepared and executed.

```
         The program accepts an SQL command from
         the user and stores it in a host variable named
         DynamicCommand.

   EXEC SQL PREPARE DynamCommand FROM :DynamicCommand
         After the command is prepared, the EXECUTE
         command is executed.

   EXEC SQL EXECUTE DynamCommand
      .
      .
      .
```

Refer to the chapter, "Using Dynamic Operations", for further explanation of the use of dynamic commands.

7

Simple Data Manipulation

Simple data manipulation is a programming technique used to SELECT or INSERT a *single* row. It can also be used to INSERT, DELETE, or UPDATE one or more rows based on a *specific criterion*. These types of data manipulation operations are considered *simple* because they can be done with SQL data manipulation commands that:

- Are not executed in conjunction with a cursor. Therefore additional SQL commands such as FETCH and OPEN are not required.
- Are not dynamically preprocessed. Therefore the amount of code required to execute them is minimized.

This chapter reviews how to use the SELECT, INSERT, DELETE, and UPDATE commands for simple data manipulation. It then examines transaction management considerations that are relevant to simple data manipulation. Finally, this chapter examines two programs that illustrate simple data manipulation.

Simple Data Manipulation Commands

The SQL commands used for simple data manipulation are:

```
SELECT
INSERT
DELETE
UPDATE
```

Refer to the *ALLBASE/SQL Reference Manual* for the complete syntax and semantics of these commands.

The SELECT Command

In simple data manipulation, you use the SELECT command to retrieve a single row, i.e., a one-row query result. The form 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)
   1           INTO :AvgSalesPrice
   2           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)
   1           INTO :AvgSalesPrice
   2           FROM  PurchDB.Parts
   3          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
   1           INTO :PartName, :SalesPrice
   2           FROM  PurchDB.Parts
   3          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:

```
   .
   .
   .
   SUBROUTINE GetRow
       MultipleRows = -10002
   .
   .
   .
```

The SELECT command is executed here.

```
         IF (SQLCode .EQ. MultipleRows) THEN
           WRITE(*,*) 'WARNING:  More than one row qualifies.'
         ENDIF
   .
   .
   .
   RETURN
   END
```

When multiple rows qualify but the receiving host variables are not in an array of records, *none* of the rows are returned.

When a column named in the WHERE clause has a unique index on it, you can omit testing for multiple-row query results if the column was defined NOT NULL. A unique index prevents the key column(s) from having duplicate values. The following index, for example, ensures that only one row will exist for any part number in *PurchDB.Parts*:

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

If a key column of a unique index *can contain a null value*, the unique index does not prevent more than one null value for that column, since each null value is considered unique. Therefore if a query contains a WHERE clause using the null predicate for such columns, multiple-row query results may occur.

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:

  This program accepts a part number from the user into a host variable named PartNumber, then retrieves a row for that part.

  ```
   EXEC SQL SELECT  PartNumber,  BinNumber
  1           INTO :PartNumber, :BinNumber
  2           FROM  PurchDB.Inventory
  3          WHERE  PartNumber = :PartNumber
  ```

  The row is displayed, and the user is asked whether they want to change the bin 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. If not, the user is prompted for another part number.

  ```
   EXEC SQL UPDATE PurchDB.Inventory
  1            SET BinNumber  = :BinNumber
  2          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 '!'
  ```

  In this example, the first and last percent signs are wildcard characters, and the percent sign after the exclamation point is the percent sign that is part of the search pattern. 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, an application can use the SELECT command. If multiple rows qualify for the SELECT operation, the UPDATE or DELETE would not be executed. Alternatively, the user could be advised that multiple rows would be affected and given a choice about whether to perform the change:

  This program prompts the user for an order number and
  a vendor part number in preparation for allowing the user
  to change the vendor part number. The following SELECT
  command determines whether more than one line item
  exists on the order for the specified vendor part number:

  ```
   EXEC SQL SELECT  ItemNumber
  1           INTO :ItemNumber
  2           FROM  PurchDB.OrderItems
  3          WHERE  OrderNumber    = :OrderNumber
  4            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)
  1           INTO :MaxOrderNumber
  2           FROM  PurchDB.Orders
  ```

  This program can increment the maximum order number by
  one, then provide the user with the new number and
  prompt for information describing the new order.

The INSERT Command

In simple data manipulation, you use the INSERT command to either insert a single row or copy one or more rows into a table from another table.

You use the following form of the INSERT command to insert a single row:

```
INSERT INTO TableName
           (ColumnNames)
     VALUES (DataValues)
```

You can omit column names when you provide values for all columns in the target table:

```
 EXEC SQL INSERT INTO   PurchDB.Parts
1             VALUES (:PartNumber,
2                     :PartName   :PartNameInd,
3                     :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, you use the following form of the INSERT command:

```
INSERT  INTO   TableName
              (ColumnName)
      SELECT   SelectList
        FROM   TableNames
       WHERE   SearchCondition1
    GROUP BY   ColumnName
      HAVING   SearchCondition2
```

Note that the SELECT command embedded in the INSERT command *cannot* contain an INTO or ORDER BY clause. In addition, any host variables used must be within the WHERE or HAVING clauses:

This program makes a copy of historical data for filled orders into the PurchDB.OldOrders table, then deletes rows for these orders from PurchDB.Orders, keeping that table minimal in size. The following INSERT command copies rows from PurchDB.Orders to PurchDB.OldOrders:

```
 EXEC SQL INSERT  INTO  PurchDB.OldOrders
1                       (OldOrder,OldVendor,OldDate)
2              SELECT   OrderNumber, VendorNumber, OrderDate
3                FROM   PurchDB.Orders
4               WHERE   OrderNumber = :OrderNumber
```

Then the DELETE command deletes rows from PurchDB.Orders:

```
 EXEC SQL DELETE FROM  PurchDB.OldOrders
1              WHERE  OrderNumber = :OrderNumber
```

The DELETE Command

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

```
DELETE FROM TableName
      WHERE SearchCondition
```

The WHERE clause specifies a *SearchCondition* that all rows satisfying to be deleted:

```
 EXEC SQL DELETE FROM PurchDB.Orders
1              WHERE OrderDate < :OrderDate
```

If the WHERE clause is omitted, *all* rows in the table are deleted.

The UPDATE Command

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

```
UPDATE TableName
   SET Columnname = :ColumnValue :ColumnValueInd
        [,...]
 WHERE SearchCondition
```

As in the case of the DELETE command, if you omit the WHERE clause, the value of any column specified is changed in *all* rows of the table.

If the WHERE clause is specified, all rows satisfying the search condition are changed:

```
EXEC SQL UPDATE PurchDB.Vendors
           SET VendorStreet  = :VendorStreet,
               VendorCity    = :VendorCity,
               VendorState   = :VendorState,
               VendorZipCode = :VendorZipCode
         WHERE VendorNumber  = :VendorNumber
```

In this example, all target columns were defined NOT NULL. If the UPDATE command is used to change the value of a column that *allows* NULL values, you use a null indicator variable directly following the variable holding the value of the column to be updated. The values that get updated at runtime depend on whether or not the program user wants to assign a null value to a column when the UPDATE command is executed:

If this program does allow the user to put a null value into column ContactName, the following UPDATE command is executed:

```
EXEC SQL UPDATE PurchDB.Vendors
           SET ContactName  = :ContactName :ContactNameInd
         WHERE VendorNumber = :VendorNumber
```

If this program does not allow the user to put a null value into the column, the following command is executed instead:

```
EXEC SQL UPDATE PurchDB.Vendors
           SET ContactName  = :ContactName
         WHERE VendorNumber = :VendorNumber
```

In the following application, the row is selected before the user enters the column data to be updated. You can achieve the same effect by using the DELETE and INSERT commands:

First retrieve all columns from the row to be updated.

```
 EXEC SQL SELECT  PartNumber,
1                 PartName,
2                 SalesPrice
3           INTO :PartNumber,
4                :PartName   :PartNameInd,
5                :SalesPrice :SalesPriceInd
6           FROM :PurchDB.Parts
7          WHERE  PartNumber = :PartNumber
```

Prompt the user for new values. If the user wants to set a column to null, set the indicator variable for that column to -1.

```
     WRITE (*,*) 'Enter new PartName (or 0 for NULL)> '
     READ(6,100) PartName
100  FORMAT (A30)
     IF (PartName .EQ. '0') THEN
       PartNameInd = -1
     ELSE
       PartNameInd = 0
     ENDIF
C
     WRITE (*,*) 'Enter new SalesPrice (or 0 for NULL)> '
     READ(6,101) SalesPrice
101  FORMAT (F10.2)
     IF (SalesPrice .EQ. 0) THEN
       SalesPriceInd = -1
     ELSE
       SalesPriceInd = 0
     ENDIF
```

After accepting new data values from the user, UPDATE the existing row.

```
 EXEC SQL UPDATE FROM PurchDB.Parts
1                SET  PartNumber = :PartNumber,
2                     PartName   = :PartName :PartNameInd,
3                     SalesPrice = :SalesPrice :SalesPriceInd
```

When an indicator variable contains a value less than 0, ALLBASE/SQL assigns a null value to that column. When the indicator variable contains a value of 0, ALLBASE/SQL assigns the data entered by the user to the column.

The following combination of DELETE and INSERT commands would have accomplished the same result, as long as all columns in the table were in the INSERT command:

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

 EXEC SQL INSERT INTO   PurchDB.Parts
1                     (PartNumber,
2                      PartName,
3                      SalesPrice)
4             VALUES (:PartNumber,
5                     :PartName   :PartNameInd,
6                     :SalesPrice :SalesPriceInd)
```

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 chapter, *Programming for Performance*.

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 (SQLCODE .LT. 0) THEN
  EXEC SQL ROLLBACK WORK;
```

Program Using SELECT, UPDATE, DELETE, and INSERT

The flow chart shown in Figure 7-1 summarizes the functionality of program forex7, which uses the four simple data manipulation commands to operate on the PurchDB.Vendors table. Forex7 uses a function menu to determine whether to execute one or more SELECT, UPDATE, DELETE, or INSERT operations. Each execution of a simple data manipulation command is done in a separate transaction.

The runtime dialog for program forex7 appears in Figure 7-2, and the source code in Figure 7-3.

The main program $\boxed{1}$ first calls function *ConnectDBE* $\boxed{3}$ to start a DBE session. This function executes the CONNECT command for the sample DBEnvironment, *PartsDBE*. The main program then displays a menu of selections. The next operation performed depends on the number entered in response to this menu:

- 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.

Select Function

Function *Select* $\boxed{10}$ prompts for a vendor number or a zero. If a zero is entered, the function menu is re-displayed. If a vendor number is entered, subroutine *BeginTransaction* $\boxed{5}$ is executed to issue the BEGIN WORK command. Then a SELECT command is executed to retrieve all data for the vendor specified from PurchDB.Vendors. The SQLCode returned is examined to determine the next action:

- If no rows qualify for the SELECT operation, a message is displayed and subroutine *CommitWork* $\boxed{6}$ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.
- If more than one row qualifies for the SELECT operation, a different message is displayed and subroutine *CommitWork* $\boxed{6}$ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.
- If the SELECT command execution results in an error condition, subroutine *SQLStatusCheck* $\boxed{2}$ is executed. This subroutine executes SQLEXPLAIN to display all error messages. If the error is serious, (less than -14024) a message is displayed and subroutine TerminateProgram (4) is called to release the DBEnvironment and terminate the entire program. If the error is not serious, subroutine *CommitWork* $\boxed{6}$ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.
- If the SELECT command can be successfully executed, subroutine *DisplayRow* $\boxed{9}$ is executed to display the row. This subroutine examines the null indicators for each of the three potentially null columns (*ContactName*, *PhoneNumber*, and *VendorRemarks*). If any null indicator contains a value less than zero, a message indicating that the value is null is displayed. After the row is completely displayed, subroutine *CommitWork* $\boxed{6}$ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.

Update Function

Function *Update* $\boxed{12}$ lets the user UPDATE the value of a column only if it contains a null value. The function prompts for a vendor number or a zero. If a zero is entered, the function menu is re-displayed. If a vendor number is entered, subroutine *BeginTransaction* $\boxed{5}$ is executed. Then a SELECT command is executed to retrieve data from *PurchDB.Vendors* for the vendor specified. The SQLCode returned is examined to determine the next action:

- If no rows qualify for the SELECT operation, a message is displayed and subroutine *CommitWork* $\boxed{6}$ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.
- If more than one row qualifies for the SELECT operation, a different message is displayed and subroutine *CommitWork* $\boxed{6}$ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.
- If the SELECT command execution results in an error condition, subroutine *SQLStatusCheck* $\boxed{2}$ is executed. Then subroutine *CommitWork* $\boxed{6}$ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.
- If the SELECT command can be successfully executed, subroutine *DisplayUpdate* $\boxed{11}$ is executed. This subroutine executes subroutine *DisplayRow* $\boxed{9}$ to display the row retrieved. Function *AnyNulls* $\boxed{8}$ is then executed to determine whether the row contains any null values. This boolean function evaluates to TRUE if the indicator variable for any of the three potentially null columns contains a non-zero value.

  If function *AnyNulls* evaluates to FALSE, a message is displayed, no UPDATE is performed, and subroutine *CommitWork* $\boxed{6}$ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.

  If function *AnyNulls* evaluates to TRUE, the null indicators are examined to determine which of them contain negative values. If the null indicator is less than zero, the column contains a null value, and the user is prompted for a new value. If the user enters a zero, the program assigns a -1 to the null indicator so that when the UPDATE command is executed, a null value is assigned to that column. If a non-zero value is entered, the program assigns a 0 to the null indicator so that the value specified is assigned to that column. After the UPDATE command is executed, subroutine *CommitWork* $\boxed{6}$ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.

Delete Function

Function *Delete* $\boxed{14}$ lets the user DELETE one row. The function prompts for a vendor number or a zero. If a zero is entered, the function menu is re-displayed. If a vendor number is entered, subroutine *BeginTransaction* $\boxed{5}$ is executed. Then a SELECT command is executed to retrieve all data for the vendor specified from PurchDB.Vendors. The SQLCode returned is examined to determine the next action:

- If no rows qualify for the SELECT operation, a message is displayed and subroutine *CommitWork* $\boxed{6}$ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.

- If more than one row qualifies for the SELECT operation, a different message is displayed and subroutine *CommitWork* $\boxed{6}$ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.
- If the SELECT command execution results in an error condition, subroutine *SQLStatusCheck* $\boxed{2}$ is executed. Then subroutine *CommitWork* $\boxed{6}$ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.
- If the SELECT command can be successfully executed, subroutine *DisplayDelete* $\boxed{13}$ is executed. This subroutine executes subroutine *DisplayRow* $\boxed{9}$ to display the row retrieved. Then the user is asked whether she wants to actually delete the row. If the user does not wish to delete, subroutine *CommitWork* $\boxed{6}$ terminates the transaction by executing the COMMIT WORK command, and the user is re-prompted for a vendor number or a zero. If the user does wish to delete, the DELETE command is executed, then subroutine *CommitWork* $\boxed{6}$ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.

Insert Function

Function *Insert* $\boxed{15}$ lets the user INSERT one row. The subroutine prompts for a vendor number or a zero. If a zero 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 zero to specify a null value for potentially null columns; to assign a null value, the program assigns a -1 to the appropriate null indicator. Subroutine BeginTransaction is executed to start a transaction, then an INSERT command is used to insert a row containing the specified values. If the INSERT operation results in an error condition, subroutine *SQLStatusCheck* $\boxed{2}$ is executed, and then subroutine *RollBackWork* $\boxed{7}$ is executed to issue the ROLLBACK WORK command. If the INSERT operation is successful, subroutine *CommitWork* $\boxed{6}$ terminates the transaction by executing the COMMIT WORK command. The user is then re-prompted for a vendor number or a zero.

When the user enters a zero in response to the function menu display, the program terminates by executing subroutine *TerminateProgram* $\boxed{4}$. This subroutine executes the RELEASE command.

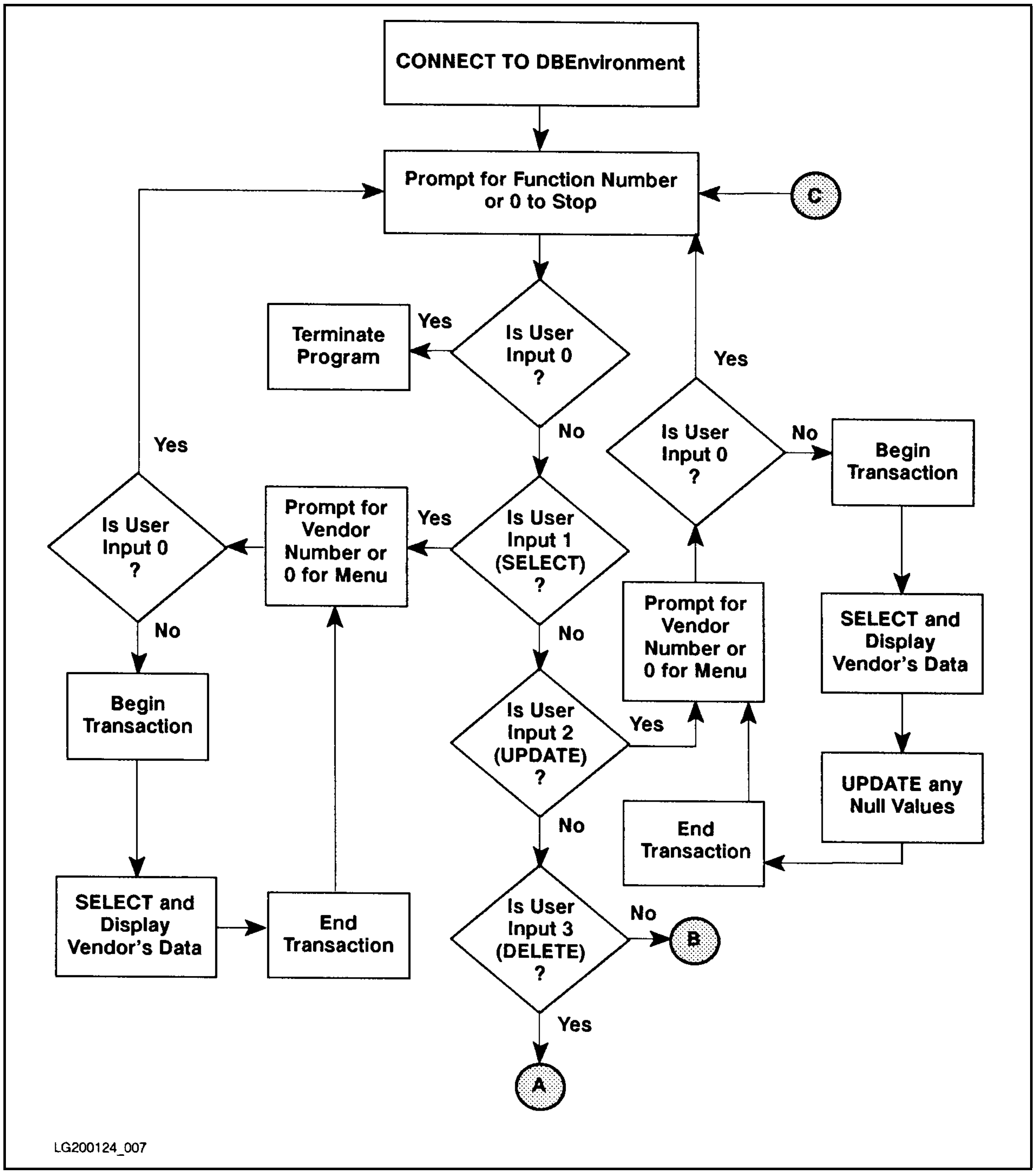

CONNECT TO DBEnvironment
Prompt for Function Number or 0 to Stop
C
Terminate Program
Yes
Is User Input 0 ?
No
Yes
Is User Input 0 ?
No
Begin Transaction
Yes
Is User Input 0 ?
No
Prompt for Vendor Number or 0 for Menu
Yes
Is User Input 1 (SELECT) ?
No
Prompt for Vendor Number or 0 for Menu
SELECT and Display Vendor's Data
Begin Transaction
Is User Input 2 (UPDATE) ?
Yes
No
End Transaction
UPDATE any Null Values
SELECT and Display Vendor's Data
End Transaction
Is User Input 3 (DELETE) ?
No
B
Yes
A
LG200124_007

Figure 7-1. Flow Chart of Program forex7

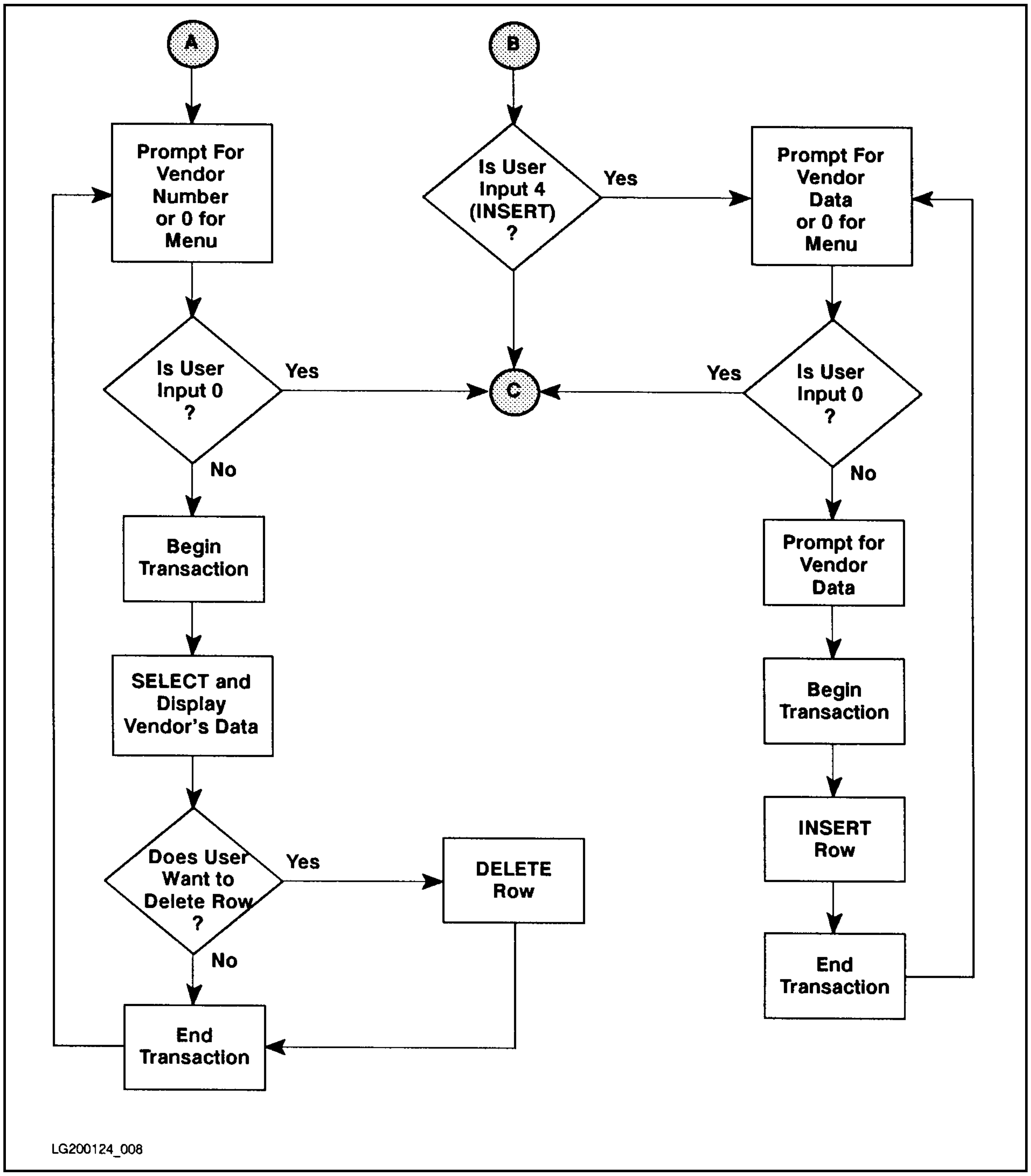


Figure 7-1. Flow Chart of Program forex7 (page 2 of 2)

```
: run forex7
 Program for Simple Data Manipulation of
   the Vendors Table -- forex7

 Event List:
   CONNECT TO PartsDBE
   Prompt for type of transaction
   Prompt for VendorNumber
   BEGIN WORK
   Display row
   Perform specified function
   COMMIT WORK or ROLLBACK WORK
   Repeat the above five steps until user enters 0
   Repeat the above seven steps until user enters 0
   RELEASE PartsDBE

 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 your choice or a 0 to STOP > 4

 Enter Vendor Number to INSERT or a 0 to STOP >  9016

 Enter Vendor Name > Wolfe Works

 Enter new ContactName (0 for NULL) > Stanley Wolfe

 Enter new PhoneNumber (0 for NULL) > 408 975 6061

 Enter new Vendor Street > 7614 Canine Way
```

Figure 7-2. Runtime Dialog of Program forex7

```
Enter new Vendor City > San Jose

Enter new Vendor State > CA

Enter new Vendor Zip Code > 90016

Enter new VendorRemarks (0 for NULL) > 0

BEGIN WORK
INSERT new row into PurchDB.Vendors
COMMIT WORK

Enter Vendor Number to INSERT or a 0 to STOP > 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 your choice or a 0 to STOP > 1

Enter Vendor Number to SELECT or a 0 to STOP > 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
```

Figure 7-2. Runtime Dialog of Program forex7 (page 2 of 4)

```
Enter Vendor Number to SELECT or a 0 to STOP > 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 your choice or a 0 to STOP > 2

Enter Vendor Number to UPDATE or a 0 to STOP > 9016

BEGIN WORK
SELECT * from PurchDB.Vendors

 VendorNumber:       9016
 VendorName:   Wolfe Works
 ContactName:  Stanley Wolfe
 PhoneNumber:  408 975 6061
 VendorStreet: 7614 Canine Way
 VendorCity:   San Jose
 VendorState:  CA
 VendorZipCode:90016
 VendorRemarks is NULL

Enter new VendorRemarks (0 for NULL) > can expedite shipments

UPDATE the PurchDB.Vendors table
COMMIT WORK

Enter Vendor Number to UPDATE or a 0 to STOP > 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 your choice or a 0 to STOP > 3
```

Figure 7-2. Runtime Dialog of Program forex7 (page 3 of 4)

```
Enter Vendor Number to DELETE or a 0 to STOP > 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 or a 0 to STOP > 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 your choice or a 0 to STOP > 0

RELEASE PartsDBE
:
```

Figure 7-2. Runtime Dialog of Program forex7 (page 4 of 4)

```
      PROGRAM forex7
C     * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C     * This program illustrates simple data manipulation.  It  *
C     * uses the UPDATE command with indicator variables to     *
C     * update any row in the Vendors table that contains null  *
C     * values.  It also uses indicator variables in            *
C     * conjunction with SELECT and INSERT. The DELETE          *
C     * command is also illustrated.                            *
C     * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

      IMPLICIT NONE

      LOGICAL*2 Done, ConnectDBE, Select, Update, Delete
      LOGICAL*2 Insert
      CHARACTER Response

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

C             (* Beginning of the Main Program *)  1

      WRITE (*,*) CHAR(27),'U'
      WRITE (*,*) 'Program for Simple Data Manipulation of Vendors
     1 - table forex7'
      WRITE (*,*) ' '
      WRITE (*,*) 'Event List:'
      WRITE (*,*) '  CONNECT TO PartsDBE'
      WRITE (*,*) '  Prompt for type of transaction'
      WRITE (*,*) '  Prompt for VendorNumber'
      WRITE (*,*) '  BEGIN WORK'
      WRITE (*,*) '  Display row'
      WRITE (*,*) '  Perform specified function'
      WRITE (*,*) '  COMMIT WORK or ROLLBACK WORK'
      WRITE (*,*) '  Repeat the above five steps until user enters 0'
      WRITE (*,*) '  Repeat the above seven steps until user enters 0'
      WRITE (*,*) '  RELEASE PartsDBE'
      WRITE (*,*) ' '
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE

```
      IF (ConnectDBE()) THEN
        Done = .FALSE.
        DO WHILE (.NOT.Done)
          WRITE (*,*) ' '
          WRITE (*,*) '1....SELECT rows from PurchDB.Vendors table'
          WRITE (*,*) '2....UPDATE rows with null values in PurchDB.Vend
     1ors table'
          WRITE (*,*) '3....DELETE rows from PurchDB.Vendors table'
          WRITE (*,*) '4....INSERT rows into PurchDB.Vendors table'
          WRITE (*,*) ' '
          WRITE (*,100)
100        FORMAT($, ' Enter your choice or a 0 to STOP > ')
          READ (*,110) Response
110       FORMAT(A1)
          IF (Response .EQ. '0') THEN
            Done = .TRUE.
          ELSEIF (Response .EQ. '1') THEN
            Done = Select()
          ELSEIF (Response .EQ. '2') THEN
            Done = Update()
          ELSEIF (Response .EQ. '3') THEN
            Done = Delete()
          ELSEIF (Response .EQ. '4') THEN
            Done = Insert()
          ELSE
            WRITE (*,*) ' Enter 0-4 only please!'
            WRITE (*,*) ' '
          ENDIF
        END DO
        CALL TerminateProgram
      ELSE
        WRITE (*,*) 'Cannot Connect to your DBEnvironment!'
      ENDIF
      STOP
      END
C              (* End of Main Program *)

C              (* Beginning of the Sub-Routines *)
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 2 of 21)

```
      SUBROUTINE SQLStatusCheck                                      2
C**** SUBROUTINE SQLStatusCheck checks status of SQL commands
C**** and print HPSQL error messages.

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

      LOGICAL*2 Abort, Check
      INTEGER   DeadLock
      PARAMETER (DeadLock =     -14024)


C             (* Begin Host Variable Declarations *)

      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*120 SQLMessage
      EXEC SQL END DECLARE SECTION

C             (* End Host Variable Declarations *)

      Abort = .FALSE.
      IF (SQLCode .LT. DeadLock) THEN
        Abort = .TRUE.
        WRITE (*,*) 'A serious error has occured!'
      ENDIF

      Check = .TRUE.
      DO WHILE (Check)

        EXEC SQL SQLEXPLAIN :SQLMessage
        WRITE(*, 100) SQLMessage
100     FORMAT(A120)
        IF (SQLCode .EQ. 0) THEN
          Check = .FALSE.
        ENDIF
      END DO

      IF (Abort) THEN
        CALL TerminateProgram
        STOP 'Program Aborted'
      END IF
      RETURN
      END
C             (* End of Subroutine SQLStatusCheck *)
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 3 of 21)

```
      LOGICAL*2 FUNCTION ConnectDBE()                          3
C**** FUNCTION to connect to PartsDBE

      INTEGER*2 OK
      PARAMETER (OK = 0)

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION

      WRITE (*,*) 'CONNECT TO PartsDBE'
      EXEC SQL CONNECT TO 'PartsDBE'
      ConnectDBE = .TRUE.
      IF (SQLCode .NE. OK) THEN
        ConnectDBE = .FALSE.
        CALL SQLStatusCheck
      ENDIF
      RETURN
      END
C             (* End of Function ConnectDBE *)

      SUBROUTINE TerminateProgram                              4
C**** SUBROUTINE  to release from PartsDBE

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION

      WRITE(*,*) ' '
      WRITE(*,*) 'RELEASE PartsDBE'
      EXEC SQL RELEASE
      RETURN
      END
C             (* End of Subroutine TerminateProgram *)
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 4 of 21)

```
      SUBROUTINE BeginTransaction                             5
C**** SUBROUTINE to begin work
      INTEGER*2 OK
      PARAMETER (OK = 0)

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION

      WRITE (*,*) ' '
      WRITE (*,*) 'BEGIN WORK'
      EXEC SQL BEGIN WORK
      IF (SQLCode .NE. OK) THEN
        CALL SQLStatusCheck
        CALL TerminateProgram
      ENDIF
      RETURN
      END
C             (* End of Subroutine BeginTransaction *)

      SUBROUTINE CommitWork                                   6
C**** SUBROUTINE to commit work
      INTEGER*2 OK
      PARAMETER (OK = 0)

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION

      WRITE(*,*) 'COMMIT WORK'
      EXEC SQL COMMIT WORK
      IF (SQLCode .NE. OK) THEN
        CALL SQLStatusCheck
        CALL TerminateProgram
      ENDIF
      RETURN
      END
C             (* End of Subroutine CommitWork *)
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 5 of 21)

```
      SUBROUTINE RollBackWork                                          7
C**** SUBROUTINE to RollBack Work
      INTEGER*2 OK
      PARAMETER (OK = 0)

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION

      WRITE(*,*) 'ROLLBACK WORK'
      EXEC SQL ROLLBACK WORK
      IF (SQLCode .NE. OK) THEN
        CALL SQLStatusCheck
        CALL TerminateProgram
      ENDIF
      RETURN
      END
C             (* End of Subroutine RollBackWork *)

      LOGICAL*2 FUNCTION AnyNulls(ContactNameInd,
     1          PhoneNumberInd, VendorRemarksInd)
C****FUNCTION to test rows for NULL values                             8

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

C             (* Begin Host Variable Declarations *)

      EXEC SQL BEGIN DECLARE SECTION
      SQLIND ContactNameInd, PhoneNumberInd, VendorRemarksInd
      EXEC SQL END DECLARE SECTION

C             (* End Host Variable Declarations *)

      IF ((ContactNameInd .EQ. 0) .AND.
     1   (PhoneNumberInd .EQ. 0) .AND.
     2   (VendorRemarksInd .EQ. 0)) THEN
C        (All columns that might be null contain non-null values)
        WRITE (*,*) 'No null values exist for this vendor.'
        WRITE (*,*) ' '
        AnyNulls = .FALSE.
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 6 of 21)

```
      ELSE
        AnyNulls = .TRUE.
      ENDIF
      RETURN
      END
C             (* End of Function AnyNulls *)

      SUBROUTINE DisplayRow (VendorNumber, VendorName, ContactName,
     1               PhoneNumber, VendorStreet, VendorCity,
     2               VendorState, VendorZipCode, VendorRemarks,
     3               ContactNameInd, PhoneNumberInd, VendorRemarksInd)
C**** SUBROUTINE to display Vendors table rows   [9]

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

C             (* Begin Host Variable Declarations *)

      EXEC SQL BEGIN DECLARE SECTION
      INTEGER*4         VendorNumber
      CHARACTER*30      VendorName
      CHARACTER*30      ContactName
      SQLIND            ContactNameInd
      CHARACTER*16      PhoneNumber
      SQLIND            PhoneNumberInd
      CHARACTER*30      VendorStreet
      CHARACTER*20      VendorCity
      CHARACTER*2       VendorState
      CHARACTER*10      VendorZipCode
      CHARACTER*50      VendorRemarks
      SQLIND            VendorRemarksInd
      CHARACTER*120     SQLMessage
      EXEC SQL END DECLARE SECTION

C             (* End Host Variable Declarations *)

      WRITE(*,*) ' '
      WRITE(*, '(''  VendorNumber: '',I10)') VendorNumber
      WRITE(*, '(''  VendorName:   '',A30)') VendorName
      IF (ContactNameInd .LT. 0) THEN
        WRITE(*,*) ' ContactName is NULL'
      ELSE
        WRITE(*, '(''  ContactName:  '',A30)') ContactName
      ENDIF
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 7 of 21)

```
      IF (PhoneNumberInd .LT. 0) THEN
        WRITE(*,*) ' PhoneNumber is NULL'
      ELSE
        WRITE(*, '(''  PhoneNumber:  '',A16)') PhoneNumber
      ENDIF
        WRITE(*, '(''  VendorStreet: '',A30)') VendorStreet
        WRITE(*, '(''  VendorCity:   '',A20)') VendorCity
        WRITE(*, '(''  VendorState:  '',A2)')  VendorState
        WRITE(*, '(''  VendorZipCode:'',A10)') VendorZipCode
      IF (VendorRemarksInd .LT. 0) THEN
        WRITE(*,*) ' VendorRemarks is NULL'
      ELSE
        WRITE(*, '(''  VendorRemarks:'',A50)') VendorRemarks
      ENDIF
      WRITE(*,*) ' '
      RETURN
      END
C             (* End of Subroutine DisplayRow *)


      LOGICAL*2 FUNCTION Select()                               10
C**** FUNCTION to select rows from PurchDB.Vendors table.

      INTEGER   NotFound,MultipleRows,OK
      LOGICAL*2 AnyNulls
      PARAMETER (NotFound = 100,
     1           MultipleRows = -10002,
     2           OK = 0)

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

C             (* Begin Host Variable Declarations *)

      EXEC SQL BEGIN DECLARE SECTION
      INTEGER*4         VendorNumber
      CHARACTER*30      VendorName
      CHARACTER*30      ContactName
      SQLIND            ContactNameInd
      CHARACTER*16      PhoneNumber
      SQLIND            PhoneNumberInd
      CHARACTER*30      VendorStreet
      CHARACTER*20      VendorCity
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 8 of 21)

```
      CHARACTER*2        VendorState
      CHARACTER*10       VendorZipCode
      CHARACTER*50       VendorRemarks
      SQLIND             VendorRemarksInd
      CHARACTER*120      SQLMessage
      EXEC SQL END DECLARE SECTION

C             (* End Host Variable Declarations *)

      Select = .FALSE.
      VendorNumber = 1
      DO WHILE (VendorNumber .NE. 0)
        WRITE (*,*) ' '
        WRITE (*,100)
100      FORMAT($, ' Enter Vendor Number to SELECT or a 0 to STOP > ')
        READ (*,110) VendorNumber
110     FORMAT(I4)
        IF (VendorNumber .NE. 0) THEN
          CALL BeginTransaction
          WRITE (*,*) ' '
          WRITE (*,*) 'SELECT * from PurchDB.Vendors'
          EXEC SQL SELECT VendorNumber,
     1                    VendorName,
     2                    ContactName,
     3                    PhoneNumber,
     4                    VendorStreet,
     5                    VendorCity,
     6                    VendorState,
     7                    VendorZipCode,
     8                    VendorRemarks
     9             INTO  :VendorNumber,
     1                   :VendorName,
     2                   :ContactName :ContactNameInd,
     3                   :PhoneNumber :PhoneNumberInd,
     4                   :VendorStreet,
     5                   :VendorCity,
     6                   :VendorState,
     7                   :VendorZipCode,
     8                   :VendorRemarks :VendorRemarksInd
     9             FROM   PurchDB.Vendors
     1             WHERE VendorNumber = :VendorNumber
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 9 of 21)

```
          IF (SQLCode .EQ. OK) THEN
            CALL
 DisplayRow(VendorNumber,VendorName,ContactName,
     1           PhoneNumber,VendorStreet,VendorCity,
     2           VendorState,VendorZipCode,VendorRemarks,
     3           ContactNameInd,PhoneNumberInd,VendorRemarksInd)
          ELSEIF (SQLCode .EQ. NotFound) THEN
            WRITE (*,*) ' '
            WRITE (*,*) 'Row not found!'
          ELSEIF (SQLCode .EQ. MultipleRows) THEN
            WRITE (*,*) ' '
            WRITE (*,*) 'WARNING: More than one row qualifies!'
          ELSE
          CALL SQLStatusCheck
          ENDIF
          CALL CommitWork
        ENDIF
      END DO

      RETURN
      END
C             (* End of Function Select *)

      SUBROUTINE DisplayUpdate(VendorNumber, VendorName, ContactName,
     1           PhoneNumber, VendorStreet, VendorCity,
     2           VendorState, VendorZipCode, VendorRemarks,
     3           ContactNameInd, PhoneNumberInd, VendorRemarksInd)
C**** SUBROUTINE to display and update                               11
C**** a row from the PurchDB.Vendors table

      INTEGER   NotFound,MultipleRows,OK
      LOGICAL*2 AnyNulls
      PARAMETER (NotFound = 100,
     1           MultipleRows = -10002,
     2           OK = 0)

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 10 of 21)

```
C             (* Begin Host Variable Declarations *)

      EXEC SQL BEGIN DECLARE SECTION
      INTEGER*4         VendorNumber
      CHARACTER*30      VendorName
      CHARACTER*30      ContactName
      SQLIND            ContactNameInd
      CHARACTER*16      PhoneNumber
      SQLIND            PhoneNumberInd
      CHARACTER*30      VendorStreet
      CHARACTER*20      VendorCity
      CHARACTER*2       VendorState
      CHARACTER*10      VendorZipCode
      CHARACTER*50      VendorRemarks
      SQLIND            VendorRemarksInd
      CHARACTER*120     SQLMessage
      EXEC SQL END DECLARE SECTION

      CALL DisplayRow(VendorNumber, VendorName, ContactName,
     1                PhoneNumber, VendorStreet, VendorCity,
     2                VendorState, VendorZipCode, VendorRemarks,
     3                ContactNameInd, PhoneNumberInd, VendorRemarksInd)
      IF (AnyNulls(ContactNameInd, PhoneNumberInd,
     1     VendorRemarksInd)) THEN
        IF (ContactNameInd .LT. 0) THEN
          WRITE(*,*)
          WRITE(*,100)
100       FORMAT($, ' Enter new ContactName (0 for NULL) > ')
          READ(*,110) ContactName
110       FORMAT (A30)
        ENDIF

        IF (PhoneNumberInd .LT. 0) THEN
          WRITE (*,*) ' '
          WRITE(*,120)
120       FORMAT($, ' Enter new PhoneNumber (0 for NULL) > ')
          READ(*,130) PhoneNumber
130       FORMAT(A16)
        ENDIF

        IF (VendorRemarksInd .LT. 0) THEN
          WRITE(*,*)
          WRITE(*,140)
140       FORMAT($, ' Enter new VendorRemarks (0 for NULL) > ')
          READ(*,150) VendorRemarks
150       FORMAT(A50)
        ENDIF
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 11 of 21)

```
          IF (ContactName .EQ. '0') THEN
            ContactNameInd = -1
          ELSE
            ContactNameInd = 0
          ENDIF

          IF (PhoneNumber .EQ. '0') THEN
            PhoneNumberInd = -1
          ELSE
            PhoneNumberInd = 0
          ENDIF

          IF (VendorRemarks .EQ. '0') THEN
            VendorRemarksInd = -1
          ELSE
            VendorRemarksInd = 0
          ENDIF

             WRITE (*,*) 'UPDATE the PurchDB.Vendors table'
          EXEC SQL UPDATE PurchDB.Vendors
     1         SET ContactName = :ContactName :ContactNameInd,
     2             PhoneNumber = :PhoneNumber :PhoneNumberInd,
     3             VendorRemarks = :VendorRemarks :VendorRemarksInd
     4       WHERE VendorNumber = :VendorNumber

          IF (SQLCode .NE. OK) THEN
            CALL SQLStatusCheck
          ENDIF

        ENDIF
        RETURN
        END
C       (End of Subroutine DisplayUpdate)


        LOGICAL*2 FUNCTION Update()                                      12
C**** FUNCTION to update rows from PurchDB.Vendors table.

        INTEGER   NotFound,MultipleRows,OK
        LOGICAL*2 AnyNulls
        PARAMETER (NotFound = 100,
     1            MultipleRows = -10002,
     2            OK = 0)
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 12 of 21)

```
C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

C             (* Begin Host Variable Declarations *)

      EXEC SQL BEGIN DECLARE SECTION
      INTEGER*4         VendorNumber
      CHARACTER*30      VendorName
      CHARACTER*30      ContactName
      SQLIND            ContactNameInd
      CHARACTER*16      PhoneNumber
      SQLIND            PhoneNumberInd
      CHARACTER*30      VendorStreet
      CHARACTER*20      VendorCity
      CHARACTER*2       VendorState
      CHARACTER*10      VendorZipCode
      CHARACTER*50      VendorRemarks
      SQLIND            VendorRemarksInd
      CHARACTER*120     SQLMessage
      EXEC SQL END DECLARE SECTION

C             (* End Host Variable Declarations *)

      Update = .FALSE.
      VendorNumber = 1
      DO WHILE (VendorNumber .NE. 0)
        WRITE (*,*) ' '
        WRITE (*,100)
100     FORMAT($, ' Enter Vendor Number to UPDATE or a 0 to STOP > ')
        READ (*,110) VendorNumber
110     FORMAT(I4)
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 13 of 21)

```
          IF (VendorNumber .NE. 0) THEN
            CALL BeginTransaction
            WRITE (*,*) ' '
            WRITE (*,*) 'SELECT * from PurchDB.Vendors'
            EXEC SQL SELECT VendorNumber,
     1                       VendorName,
     2                       ContactName,
     3                       PhoneNumber,
     4                       VendorStreet,
     5                       VendorCity,
     6                       VendorState,
     7                       VendorZipCode,
     8                       VendorRemarks
     9                INTO  :VendorNumber,
     1                      :VendorName,
     2                      :ContactName :ContactNameInd,
     3                      :PhoneNumber :PhoneNumberInd,
     4                      :VendorStreet,
     5                      :VendorCity,
     6                      :VendorState,
     7                      :VendorZipCode,
     8                      :VendorRemarks :VendorRemarksInd
     9                FROM   PurchDB.Vendors
     1                WHERE  VendorNumber = :VendorNumber
            IF (SQLCode .EQ. OK) THEN
              CALL DisplayUpdate (VendorNumber, VendorName, ContactName,
     1                 PhoneNumber, VendorStreet, VendorCity,
     2                 VendorState, VendorZipCode, VendorRemarks,
     3                 ContactNameInd, PhoneNumberInd, VendorRemarksInd)
            ELSEIF (SQLCode .EQ. NotFound) THEN
              WRITE (*,*) ' '
              WRITE (*,*) 'Row not found!'
            ELSEIF (SQLCode .EQ. MultipleRows) THEN
              WRITE(*,*) ' '
              WRITE (*,*) 'WARNING: More than one row qualifies!'
              CALL SQLStatusCheck
            ENDIF
            CALL CommitWork
          ENDIF
        END DO

        RETURN
        END
C              (* End of Function Update *)
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 14 of 21)

```
C**** SUBROUTINE to Display and Delete a row                            13
C**** from the PurchDB.Vendors table
      SUBROUTINE DisplayDelete(VendorNumber, VendorName, ContactName,
     1          PhoneNumber, VendorStreet, VendorCity,
     2          VendorState, VendorZipCode, VendorRemarks,
     3          ContactNameInd, PhoneNumberInd, VendorRemarksInd)

      CHARACTER     Response
      INTEGER       NotFound,MultipleRows,OK
      LOGICAL*2     AnyNulls
      PARAMETER    (NotFound = 100,
     1              MultipleRows = -10002,
     2              OK = 0)


C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

C             (* Begin Host Variable Declarations *)

      EXEC SQL BEGIN DECLARE SECTION
      INTEGER*4         VendorNumber
      CHARACTER*30      VendorName
      CHARACTER*30      ContactName
      SQLIND            ContactNameInd
      CHARACTER*16      PhoneNumber
      SQLIND            PhoneNumberInd
      CHARACTER*30      VendorStreet
      CHARACTER*20      VendorCity
      CHARACTER*2       VendorState
      CHARACTER*10      VendorZipCode
      CHARACTER*50      VendorRemarks
      SQLIND            VendorRemarksInd
      CHARACTER*120     SQLMessage
      EXEC SQL END DECLARE SECTION

      CALL DisplayRow(VendorNumber, VendorName, ContactName,
     1          PhoneNumber, VendorStreet, VendorCity,
     2          VendorState, VendorZipCode, VendorRemarks,
     3          ContactNameInd, PhoneNumberInd, VendorRemarksInd)>
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 15 of 21)

```
      WRITE (*,100)
100   FORMAT($, ' Is it OK to DELETE this row (N/Y)? >')
      READ (*, 110) Response
110   FORMAT(A1)
      IF ((Response .EQ. 'Y') .OR. (Response .EQ. 'y')) THEN
        WRITE (*,*) ' '
        WRITE (*,*) 'DELETE row from PurchDB.Vendors!'
        EXEC SQL DELETE FROM PurchDB.Vendors
     1                 WHERE VendorNumber = :VendorNumber
        IF (SQLCode .NE. OK) THEN
          CALL SQLStatusCheck
        ENDIF
      ELSE
        WRITE (*,*) ' '
        WRITE (*,*) 'Row not deleted from PurchDB.Vendors!'
      ENDIF
      RETURN
      END
C**** (End of Subroutine DisplayDelete)


      LOGICAL*2 FUNCTION Delete()                                 14
C**** FUNCTION to delete rows from PurchDB.Vendors table.

      INTEGER    NotFound,MultipleRows,OK
      LOGICAL*2  AnyNulls
      PARAMETER (NotFound = 100,
     1           MultipleRows = -10002,
     2           OK = 0)

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

C             (* Begin Host Variable Declarations *)

      EXEC SQL BEGIN DECLARE SECTION
      INTEGER*4        VendorNumber
      CHARACTER*30     VendorName
      CHARACTER*30     ContactName
      SQLIND           ContactNameInd
      CHARACTER*16     PhoneNumber
      SQLIND           PhoneNumberInd
      CHARACTER*30     VendorStreet
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 16 of 21)

```
      CHARACTER*20      VendorCity
      CHARACTER*2       VendorState
      CHARACTER*10      VendorZipCode
      CHARACTER*50      VendorRemarks
      SQLIND            VendorRemarksInd
      CHARACTER*120     SQLMessage
      EXEC SQL END DECLARE SECTION

C             (* End Host Variable Declarations *)

      Delete = .FALSE.
      VendorNumber = 1
      DO WHILE (VendorNumber .NE. 0)
        WRITE (*,*) ' '
        WRITE (*,100)
100     FORMAT($, ' Enter Vendor Number to DELETE or a 0 to STOP > ')
        READ (*,110) VendorNumber
110     FORMAT(I4)
        IF (VendorNumber .NE. 0) THEN
          CALL BeginTransaction
          WRITE (*,*) ' '
          WRITE (*,*) 'SELECT * from PurchDB.Vendors'
          EXEC SQL SELECT VendorNumber,
     1                   VendorName,
     2                   ContactName,
     3                   PhoneNumber,
     4                   VendorStreet,
     5                   VendorCity,
     6                   VendorState,
     7                   VendorZipCode,
     8                   VendorRemarks
     9             INTO  :VendorNumber,
     1                  :VendorName,
     2                  :ContactName :ContactNameInd,
     3                  :PhoneNumber :PhoneNumberInd,
     4                  :VendorStreet,
     5                  :VendorCity,
     6                  :VendorState,
     7                  :VendorZipCode,
     8                  :VendorRemarks :VendorRemarksInd
     9              FROM  PurchDB.Vendors
     1             WHERE  VendorNumber = :VendorNumber
          IF (SQLCode .EQ. OK) THEN
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 17 of 21)

```
         CALL DisplayDelete (VendorNumber, VendorName, ContactName,
     1            PhoneNumber, VendorStreet, VendorCity,
     2            VendorState, VendorZipCode, VendorRemarks,
     3            ContactNameInd, PhoneNumberInd, VendorRemarksInd)
       ELSEIF (SQLCode .EQ. NotFound) THEN
         WRITE (*,*) ' '
         WRITE (*,*) 'Row not found!'
       ELSEIF (SQLCode .EQ. MultipleRows) THEN
         WRITE (*,*) ' '
         WRITE (*,*) 'WARNING: More than one row qualifies!'
       ELSE
         CALL SQLStatusCheck
       ENDIF
       CALL CommitWork
      ENDIF
    END DO
    RETURN
    END
C** (End of Function Delete)


    LOGICAL*2 FUNCTION Insert()                                      15
C** FUNCTION to insert a row into the Vendors table

    INTEGER    NotFound,MultipleRows,OK
    LOGICAL*2  AnyNulls
    PARAMETER (NotFound = 100,
   1           MultipleRows = -10002,
   2           OK = 0)

C           (* Begin SQL Communication Area *)

    EXEC SQL INCLUDE SQLCA

C           (* Begin Host Variable Declarations *)
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 18 of 21)

```
      EXEC SQL BEGIN DECLARE SECTION
      INTEGER*4         VendorNumber
      CHARACTER*30      VendorName
      CHARACTER*30      ContactName
      SQLIND            ContactNameInd
      CHARACTER*16      PhoneNumber
      SQLIND            PhoneNumberInd
      CHARACTER*30      VendorStreet
      CHARACTER*20      VendorCity
      CHARACTER*2       VendorState
      CHARACTER*10      VendorZipCode
      CHARACTER*50      VendorRemarks
      SQLIND            VendorRemarksInd
      CHARACTER*120     SQLMessage
      EXEC SQL END DECLARE SECTION

C             (* End Host Variable Declarations *)

      Insert = .FALSE.
      VendorNumber = 1
      DO WHILE (VendorNumber .NE. 0)
        WRITE (*,*) ' '
        WRITE (*,100)
100     FORMAT($, ' Enter Vendor Number to INSERT or a 0 to STOP > ')
        READ (*,110) VendorNumber
110     FORMAT(I4)
        IF (VendorNumber .NE. 0) THEN
          WRITE (*,*) ' '
          WRITE (*,120)
120       FORMAT($,' Enter Vendor Name > ')
          READ (*,130) VendorName
130       FORMAT(A30)

          WRITE(*,*)
          WRITE (*,140)
140       FORMAT($,' Enter new ContactName (0 for NULL) > ')
          READ(*,150) ContactName
150       FORMAT (A30)

          IF (ContactName .EQ. '0') THEN
            ContactNameInd = -1
          ELSE
            ContactNameInd = 0
          ENDIF

          WRITE (*,*)
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 19 of 21)

```
             WRITE (*,160)
 160         FORMAT($,' Enter new PhoneNumber (0 for NULL) > ')
             READ(*,170) PhoneNumber
 170         FORMAT(A16)

             IF (PhoneNumber .EQ. '0') THEN
               PhoneNumberInd = -1
             ELSE
               PhoneNumberInd = 0
             ENDIF

             WRITE(*,*)
             WRITE (*,180)
 180         FORMAT($,' Enter new Vendor Street > ')
             READ(*,190) VendorStreet
 190         FORMAT(A30)

             WRITE(*,*)
             WRITE (*,200)
 200         FORMAT($,' Enter new Vendor City > ')
             READ(*,210) VendorCity
 210         FORMAT(A20)

             WRITE(*,*)
             WRITE (*,220)
 220         FORMAT($,' Enter new Vendor State > ')
             READ(*,230) VendorState
 230         FORMAT(A2)

             WRITE(*,*)
             WRITE (*,240)
 240         FORMAT($,' Enter new Vendor Zip Code > ')
             READ(*,250) VendorZipCode
 250         FORMAT(A10)

             WRITE(*,*)
             WRITE (*,260)
 260         FORMAT($,' Enter new VendorRemarks (0 for NULL > ')
             READ(*,270) VendorRemarks
 270         FORMAT(A50)

             IF (VendorRemarks .EQ. '0') THEN
               VendorRemarksInd = -1
             ELSE
               VendorRemarksInd = 0
             ENDIF
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 20 of 21)

```
          CALL BeginTransaction
          WRITE (*,*)'INSERT new row into PurchDB.Vendors'
          EXEC SQL INSERT
     1             INTO  PurchDB.Vendors
     2                  (VendorNumber, VendorName, ContactName,
     3                   PhoneNumber, VendorStreet, VendorCity,
     4                   VendorState, VendorZipCode, VendorRemarks)
     5          VALUES (:VendorNumber,
     6                  :VendorName,
     6                  :ContactName :ContactNameInd,
     7                  :PhoneNumber :PhoneNumberInd,
     8                  :VendorStreet,
     9                  :VendorCity,
     1                  :VendorState,
     2                  :VendorZipCode,
     3                  :VendorRemarks :VendorRemarksInd)
          IF (SQLCode .NE. OK) THEN
            CALL SQLStatusCheck
            CALL RollBackWork
          ELSE
            CALL CommitWork
          ENDIF
        ENDIF
      END DO
      RETURN
      END
C             (* End of Function Insert *)
```

Figure 7-3. Program forex7: Using INSERT, UPDATE, SELECT, and DELETE (page 21 of 21)

8

Processing with Cursors

Sequential table processing is the programming technique you use to operate on a *multiple-row query result, one row at a time*. The query result is referred to as an *active set*. You use a pointer called a *cursor* to move through the active set, retrieving a row at a time into host variables and optionally updating or deleting the row. Reporting applications may find this technique useful. Update applications such as those that periodically operate on tables not being concurrently accessed (e.g., inventory adjustments) may also find this technique useful.

This chapter reviews how to use SQL commands to perform sequential table processing. It then examines transaction management considerations that are relevant to sequential table processing. Finally, this chapter examines a program that uses this data manipulation technique.

Sequential Table Processing Commands

The SQL commands used for sequential table processing are:

- DECLARE CURSOR: defines a cursor and associates with it 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: frees up ALLBASE/SQL intermal buffer space used to handle the cursor.

Refer to the *ALLBASE/SQL Reference Manual* for the complete syntax and semantics of these commands. Ensure that all the commands listed above for any single cursor are contained within the *same transaction*.

The DECLARE CURSOR Command

The DECLARE CURSOR command names a cursor and associates with it a particular SELECT command:

```
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 clause is used when you use the UPDATE WHERE CURRENT command to update a current row. This command may offer the simplest way to update a current row, but it imposes certain restrictions on the SELECT command. Updating a current row is fully discussed later in this chapter under the UPDATE WHERE CURRENT command.

The SELECT command for cursor declarations that do not include the FOR UPDATE clause can consist of any of the SELECT command clauses *except* the INTO clause:

```
  SELECT SelectList
    FROM TableNames
   WHERE SearchCondition1
GROUP BY ColumnNames
  HAVING SearchCondition2
ORDER BY ColumnIdentifiers
```

A SELECT command associated with a cursor does not name *output host variables*, but may name *input host variables* in the select list, the WHERE clause, 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 sequential table processing, the ORDER BY clause may be useful. In the example above, 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 FORTRAN preprocessor parses this command, it stores a section in the target DBEnvironment. At runtime, the section is not executed when the DECLARE CURSOR command is encountered, but when the OPEN command is executed. Because the DECLARE CURSOR command is not executed at runtime, you do not need to perform status checking in your program following this command.

The OPEN Command

The OPEN command examines any input host variables and determines the active set:

```
OPEN CursorName
```

The following command defines the active set associated with the cursor defined earlier:

```
EXEC SQL OPEN Inventory
```

You use the FETCH command to retrieve a row at a time from the active set.

You can use the KEEP CURSOR WITH NOLOCKS option for a cursor that involves sorting, whether through the use of a DISTINCT, GROUP BY, or ORDER BY clause, or as the result of a union or a join operation. However, for kept cursors involving sorting, ALLBASE/SQL does not ensure data integrity. See the "Programming for Performance" chapter for more information on ensuring data integrity.

The FETCH Command

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:

```
  EXEC SQL FETCH  Inventory
1          INTO :PartNumber,
                :BinNumber,
2               :QtyOnHand     :QtyOnHandInd,
4               :AdjustmentQty :AdjustmentQtyInd
5
```

The first time you execute the FETCH command, the *first* row in the query result is the current row. With each subsequent execution of the FETCH command, each succeeding row in the query result becomes current. After the last row in the query result has been fetched, ALLBASE/SQL sets SQLCode to 100. ALLBASE/SQL also sets SQLCode to 100 if *no rows* qualify for the active set. You should test for an SQLCode value of 100 after each execution of the FETCH command to determine whether to re-execute this command:

```
SUBROUTINE GetARow
.
.
OK       = 0
NotFound = 100
DoFetch  = .TRUE.
DO WHILE (DoFetch)
.
.      The FETCH command appears here.
.
IF (SQLCode .EQ. OK) THEN
  CALL DisplayRow
ELSEIF (SQLCode .EQ. NotFound) THEN
  DoFetch = .FALSE.
  CALL CloseCursor
  CALL CommitWork
ELSE
  DoFetch = .FALSE.
  CALL SQLStatusCheck
  CALL CloseCursor
  CALL RollBackWork
ENDIF
END DO
RETURN
END
```

When a row is current, you can update it by using the UPDATE WHERE CURRENT command or delete it by using the DELETE WHERE CURRENT command.

The UPDATE WHERE CURRENT Command

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:

```
        UPDATE TableName
           SET ColumnName = ColumnValue
               [,...]
         WHERE CURRENT OF CursorName
```

Any columns you name in this command must also have been named in a FOR UPDATE clause in the related DECLARE CURSOR command:

```
      EXEC SQL DECLARE AdjustQtyOnHand
     1          CURSOR FOR
     2          SELECT PartNumber,
     3                 BinNumber,
     4                 QtyOnHand,
     5                 AdjustmentQty
     6            FROM PurchDB.Inventory
     7           WHERE QtyOnHand IS NOT NULL
     8             AND AdjustmentQty IS NOT NULL
     9   FOR UPDATE OF QtyOnHand,
     1                 AdjustmentQty

      EXEC SQL OPEN AdjustQtyOnHand
     .
     .  The output host variables do not need to include
     . indicator variables, because the SELECT command
     . associated with the cursor eliminates any rows having
     . null values from the active set:
     .
```

```
      EXEC SQL FETCH  AdjustQtyOnHand
     1          INTO :PartNumber,
     2               :BinNumber,
     3               :QtyOnHand,
     4               :AdjustmentQty
      .
      .
      .
      EXEC SQL UPDATE PurchDB.Inventory
     1            SET QtyOnHand     = :QtyOnHand + :AdjustmentQty,
     2                AdjustmentQty = 0
     3          WHERE CURRENT OF AdjustQtyOnHand
```

In this example, the order of the rows in the query result is not important. Therefore the SELECT command associated with cursor *AdjustQtyOnHand* does not need to contain an ORDER BY clause and the UPDATE WHERE CURRENT command can be used.

In cases where order *is* important and the ORDER BY clause must be used, you can use the form of the UPDATE command described in Chapter 6 to update values in the current row. In this case, if more than one row qualifies for the search condition in the UPDATE command, more rows than just the current row will be changed:

```
      EXEC SQL DECLARE Inventory
     1          CURSOR FOR
     2          SELECT PartNumber,
     3                 BinNumber,
     4                 QtyOnHand,
     5                 AdjustmentQty
     6            FROM PurchDB.Inventory
     7           WHERE CountCycle = :CountCycle
     8        ORDER BY BinNumber
      .
      .
      .
      EXEC SQL FETCH  Inventory
     1          INTO :PartNumber,
     2               :BinNumber,
     3               :QtyOnHand     :QtyOnHandInd
     4               :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 later to
```

```
   .    update AdjustmentQty. The value(s) entered, as well
   .    as the current PartNumber and BinNumber, are saved
   .    until all rows have been fetched and other values
   .    accepted from the user. Then one of the following
   .    UPDATE commands is executed for each UPDATE requested
   .    by the user:
     .
 EXEC SQL UPDATE PurchDB.Inventory
1            SET AdjustmentQty = :AdjustmentQty
2          WHERE PartNumber = :PartNumber
3            AND BinNumber  = :BinNumber
     .
     .
     .
 EXEC SQL UPDATE PurchDB.Inventory
1            SET QtyOnHand =     :QtyOnHand,
2                AdjustmentQty = :AdjustmentQty
3          WHERE PartNumber = :PartNumber
4            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.

The DELETE WHERE CURRENT Command

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 structure:

```
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
1          WHERE CURRENT OF AdjustQtyOnHand
```

Even though the SELECT command associated with cursor *Inventory* names only some of the columns in table *PurchDB.Inventory*, the entire current row is deleted.

After the DELETE WHERE CURRENT command is executed, there is no current row. You must re-execute the FETCH command to obtain another current row.

As in the case of the UPDATE WHERE CURRENT command, if the SELECT command associated with the cursor contains an ORDER BY clause or other components listed earlier, you can use the DELETE command to delete a row:

```
        EXEC SQL DELETE FROM PurchDB.Inventory
       1              WHERE PartNumber = :PartNumber
       2                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.

The CLOSE Command

When you no longer want to operate on the active set, you use the CLOSE command:

```
        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:

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
```

You also use the CLOSE command when you want to re-access the active set. In this case, simply re-open the cursor after executing the CLOSE command. Because locks have not been released, any changes to the rows in the active set will be those made by your program since the cursor was first opened:

Cursor Inventory is used to update information in table PurchDB.Inventory. After the last row in the active set has been fetched and its information changed, the cursor is closed:

```
EXEC SQL CLOSE Inventory
```

The cursor is then re-opened to allow the program user to review the information and optionally make some last-minute adjustments:

```
EXEC SQL OPEN Inventory
```

After the user has reviewed all rows in the active set, any changes made to the active set are made permanent as follows:

```
EXEC SQL COMMIT WORK
```

Transaction Management for Cursor Operations

The time at which ALLBASE/SQL obtains locks during cursor processing depends on whether an index scan or a sequential scan is used to retrieve the query result.

When a cursor is based on a SELECT command for which an *index scan* is used, 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 a *sequential* scan is used, 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, the lock is elevated 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 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.

Using KEEP CURSOR

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 held before the cursor position and immediately 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.

If you use the KEEP CURSOR WITH NOLOCKS option for a cursor that involves sorting, whether through the use of a DISTINCT, GROUP BY, or ORDER BY clause, or as the result of a union or a join operation, ALLBASE/SQL does not ensure data integrity.

It is your responsibility to ensure data integrity by verifying the continued existence of a row before updating it or using it as the basis for updating some other table. For an updatable cursor, you can use either the REFETCH or SELECT command to verify the continued existence of a row. For a cursor that is non-updatable, you must use the SELECT command.

A warning (DBWARN 2056) regarding the kept cursor on a sort with no locks is generated. You *must* check for this warning if you want to detect the execution of this type of cursor operation.

Figure 8-1 shows the operation of cursors when you do *not* select the KEEP CURSOR option.

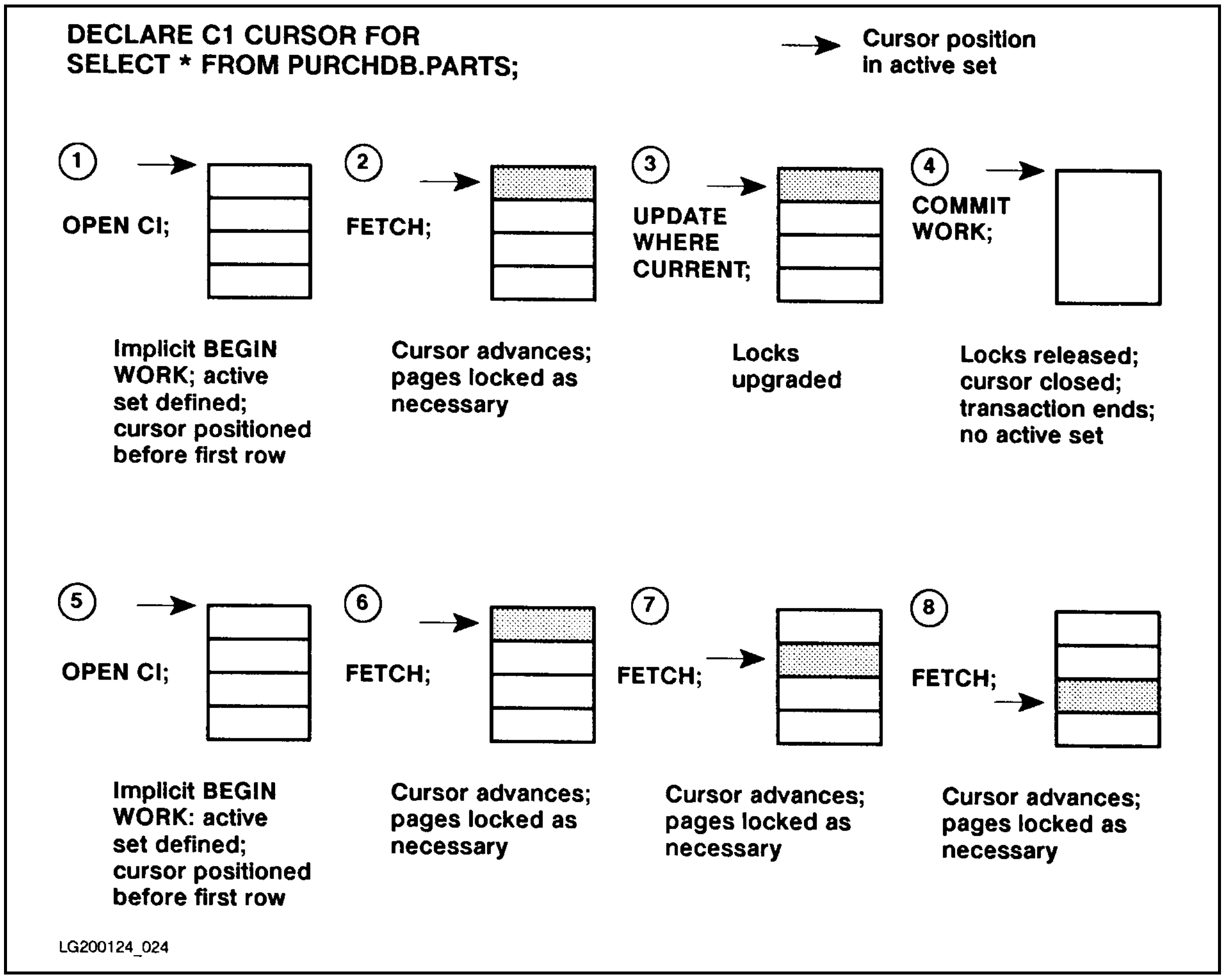

DECLARE C1 CURSOR FOR
SELECT * FROM PURCHDB.PARTS;
Cursor position in active set
1
OPEN CI;
Implicit BEGIN WORK; active set defined; cursor positioned before first row
2
FETCH;
Cursor advances; pages locked as necessary
3
UPDATE WHERE CURRENT;
Locks upgraded
4
COMMIT WORK;
Locks released; cursor closed; transaction ends; no active set
5
OPEN CI;
Implicit BEGIN WORK: active set defined; cursor positioned before first row
6
FETCH;
Cursor advances; pages locked as necessary
7
FETCH;
Cursor advances; pages locked as necessary
8
FETCH;
Cursor advances; pages locked as necessary
LG200124_024

Figure 8-1. Cursor Operation without the KEEP CURSOR Feature

After the cursor is opened, successive FETCH commands advance the cursor position. Any exclusive locks acquired along the way are retained until the transaction ends. If you have selected the Cursor Stability option in the BEGIN WORK command, 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

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 on the page that contains the current cursor position 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 and prevent a new implicit BEGIN WORK.

Figure 8-2 shows the effect of the KEEP CURSOR WITH LOCKS.

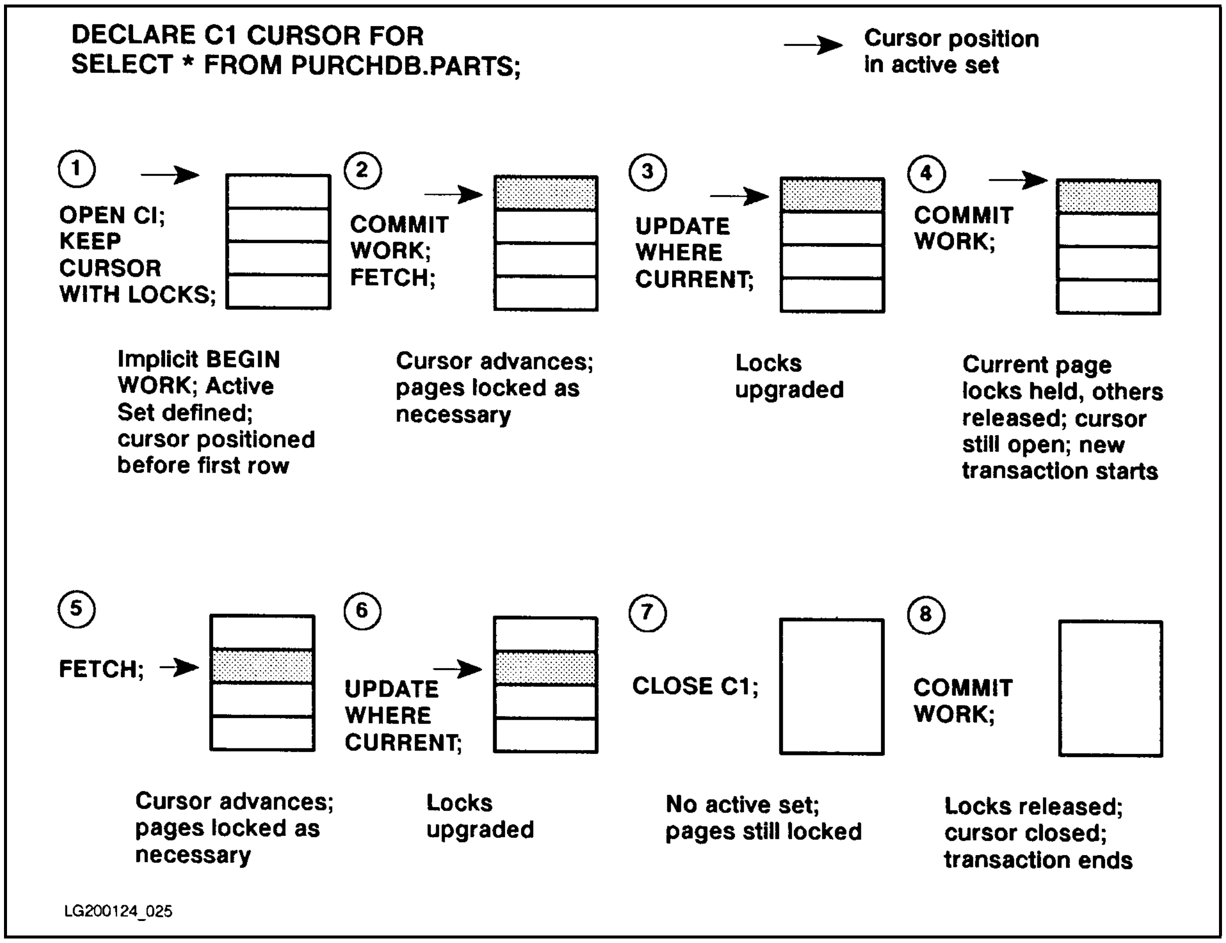

DECLARE C1 CURSOR FOR
SELECT * FROM PURCHDB.PARTS;
Cursor position in active set
1
OPEN CI; KEEP CURSOR WITH LOCKS;
Implicit BEGIN WORK; Active Set defined; cursor positioned before first row
2
COMMIT WORK; FETCH;
Cursor advances; pages locked as necessary
3
UPDATE WHERE CURRENT;
Locks upgraded
4
COMMIT WORK;
Current page locks held, others released; cursor still open; new transaction starts
5
FETCH;
Cursor advances; pages locked as necessary
6
UPDATE WHERE CURRENT;
Locks upgraded
7
CLOSE C1;
No active set; pages still locked
8
COMMIT WORK;
Locks released; cursor closed; transaction ends
LG200124_025

Figure 8-2. Cursor Operation Using KEEP CURSOR WITH LOCKS

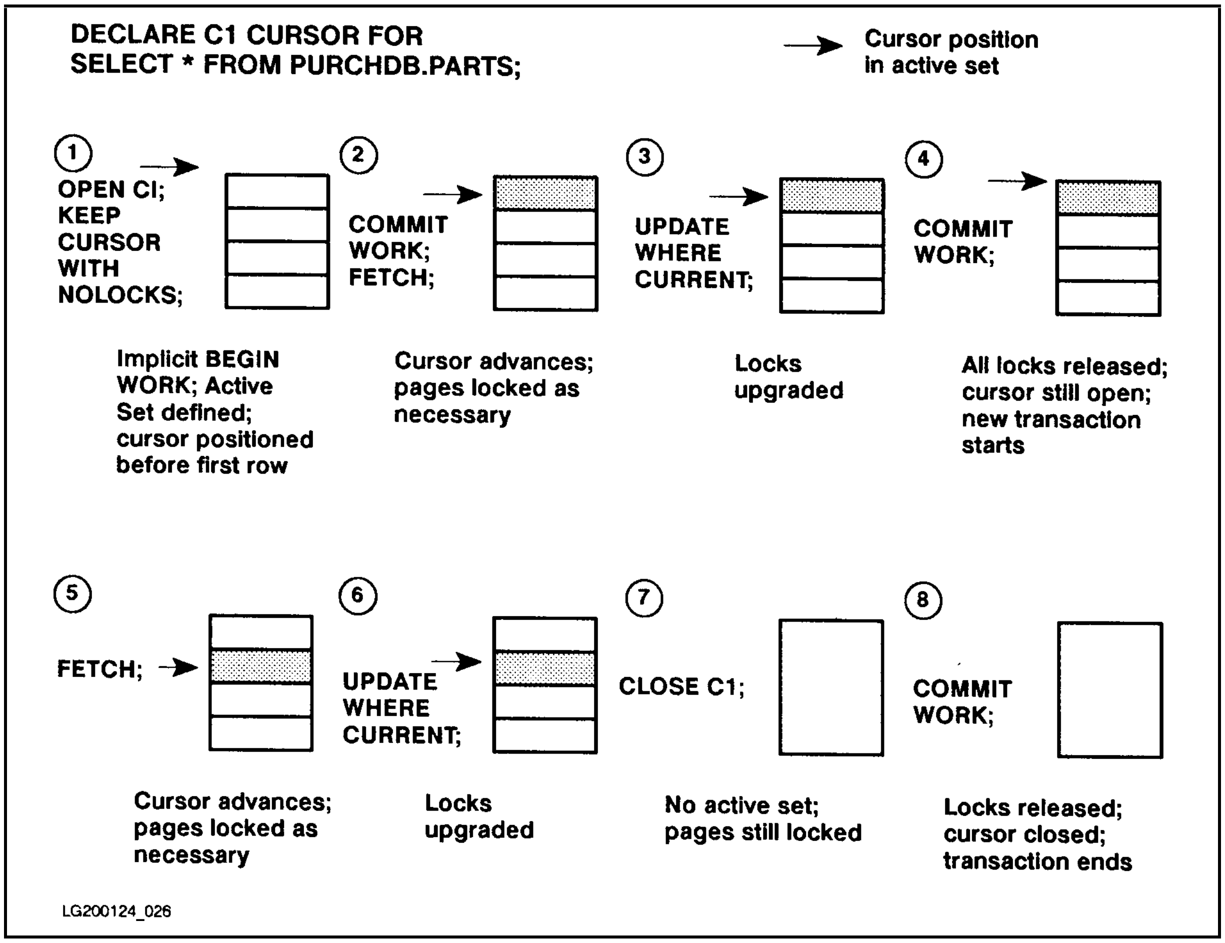
OPEN Command Using KEEP CURSOR WITH NOLOCKS

The feature has the following effects:

- A COMMIT WORK command does not close the cursor. Instead, it ends the current transaction and immediately starts another one.
- When you issue a COMMIT WORK, all locks on the page that contains the current 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.
- You cannot use the KEEP CURSOR option WITH NOLOCKS for a cursor declared as a SELECT with a DISTINCT or ORDER BY clause.

- 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 8-3 shows the effect of KEEP CURSOR WITH NOLOCKS.



DECLARE C1 CURSOR FOR
SELECT * FROM PURCHDB.PARTS;
Cursor position in active set
1
OPEN CI; KEEP CURSOR WITH NOLOCKS;
Implicit BEGIN WORK; Active Set defined; cursor positioned before first row
2
COMMIT WORK; FETCH;
Cursor advances; pages locked as necessary
3
UPDATE WHERE CURRENT;
Locks upgraded
4
COMMIT WORK;
All locks released; cursor still open; new transaction starts
5
FETCH;
Cursor advances; pages locked as necessary
6
UPDATE WHERE CURRENT;
Locks upgraded
7
CLOSE C1;
No active set; pages still locked
8
COMMIT WORK;
Locks released; cursor closed; transaction ends
LG200124_026

Figure 8-3. Cursor Operation Using KEEP CURSOR WITH NOLOCKS

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 "Controlling Performance" in the *ALLBASE/SQL Database Administration Guide.*

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, whether opened *with* or *without* the KEEP CURSOR option. Thus it is important to do a COMMIT WORK after opening a cursor with the KEEP CURSOR option.
- 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, whether opened *with* or *without* the KEEP CURSOR option. Thus it is important to do a COMMIT WORK after opening a cursor with the KEEP CURSOR option.
- 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.

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.
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.

Program Using UPDATE WHERE CURRENT

The flow chart in Figure 8-4 summarizes the functionality of program forex8. This program uses a cursor and the UPDATE WHERE CURRENT command to update column *ReceivedQty* in table *PurchDB.OrderItems*. The runtime dialog for forex8 appears in Figure 8-5, and the source code in Figure 8-6.

The main program $\boxed{1}$ first executes subroutine *DeclareCursor* $\boxed{9}$, which contains the DECLARE CURSOR command. This command is a preprocessor directive and is not executed at runtime. At runtime, subroutine *DeclareCursor* only displays the message *Declare Cursor*. The DECLARE CURSOR command defines a cursor named *OrderReview*. The cursor is associated with a SELECT command that retrieves the following columns for all rows in table *PurchDB.OrderItems* having a specific order number but no null values in column *VendPartNumber*:

```
OrderNumber (defined NOT NULL)
ItemNumber  (defined NOT NULL)
VendPartNumber
ReceivedQty
```

Cursor *OrderReview* has a FOR UPDATE clause naming column *ReceivedQty* to allow the user to change the value in this column.

Next, to establish a DBE session, program forex8 executes function *ConnectDBE* $\boxed{3}$. This function evaluates to TRUE when the CONNECT command for the sample DBEnvironment, *PartsDBE*, is successfully executed. The program then executes function *FetchUpdate* $\boxed{13}$ until the *Done* flag is set to TRUE.

FetchUpdate Function

Function *FetchUpdate* $\boxed{13}$ prompts for an order number or a 0. When the user enters a 0, *FetchUpdate* is set to FALSE, which in turn sets the *Done* flag to FALSE, and the program terminates. When the user enters an order number, the program begins a transaction by executing subroutine *BeginTransaction* $\boxed{6}$, which executes the BEGIN WORK command.

Cursor *OrderReview* is then opened by invoking function *OpenCursor* $\boxed{10}$. This function, which executes the OPEN CURSOR command, evaluates to TRUE when the command is successful.

A row at a time is retrieved and optionally updated until the *Fetch* flag is set to FALSE. 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 $\boxed{13B}$ 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 subroutine *DisplayUpdate* $\boxed{12}$ to display the current row and optionally update it.

DisplayUpdate Subroutine

Subroutine *DisplayUpdate* $\boxed{12}$ executes subroutine *DisplayRow* $\boxed{8}$ to display the current row. If column *ReceivedQty* in the current row contains a null value, the message *ReceivedQty is NULL* is displayed.

The user is then asked whether he wants to update the current *ReceivedQty* value $\boxed{12A}$. If so, the user is prompted for a new entry. Depending on the user's response, the program assigns an appropriate value to the indicator variable *ReceivedQtyInd*, and then issues the UPDATE WHERE CURRENT command $\boxed{12B}$. If the user enters a 0, the indicator variable is set to -1 and a null value is assigned to this column. If the user enters a non-zero value, the indicator variable is set to 0 and the user-entered value is assigned to the column.

The program then asks whether to FETCH another row. If so, the FETCH command is re-executed. If not, the program asks whether the user wants to make permanent any updates he may have made in the active set. To keep any row changes, the program executes subroutine *CommitWork* $\boxed{6}$, which executes the COMMIT WORK command. To undo any row changes, the program executes subroutine *RollBackWork* $\boxed{7}$, which executes the ROLLBACK WORK command.

The COMMIT WORK command is also executed when ALLBASE/SQL sets SQLCode to 100 following execution of the FETCH command. 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.

Before any COMMIT WORK or ROLLBACK WORK command is executed, cursor *OrderReview* is closed. Although the cursor is automatically closed whenever a transaction is terminated, it is good programming practice to use the CLOSE command to close open cursors prior to terminating transactions.

When the program user enters a 0 in response to the order number prompt $\boxed{13A}$, the program terminates by executing subroutine *TerminateProgram* $\boxed{4}$, which executes the RELEASE command $\boxed{2}$.

Explicit status checking is used throughout this program. After each embedded SQL command is executed, SQLCA.SQLCode is checked. If SQLCode is less than 0, the program executes subroutine *SQLStatusCheck* $\boxed{2}$, which executes the SQLEXPLAIN command.

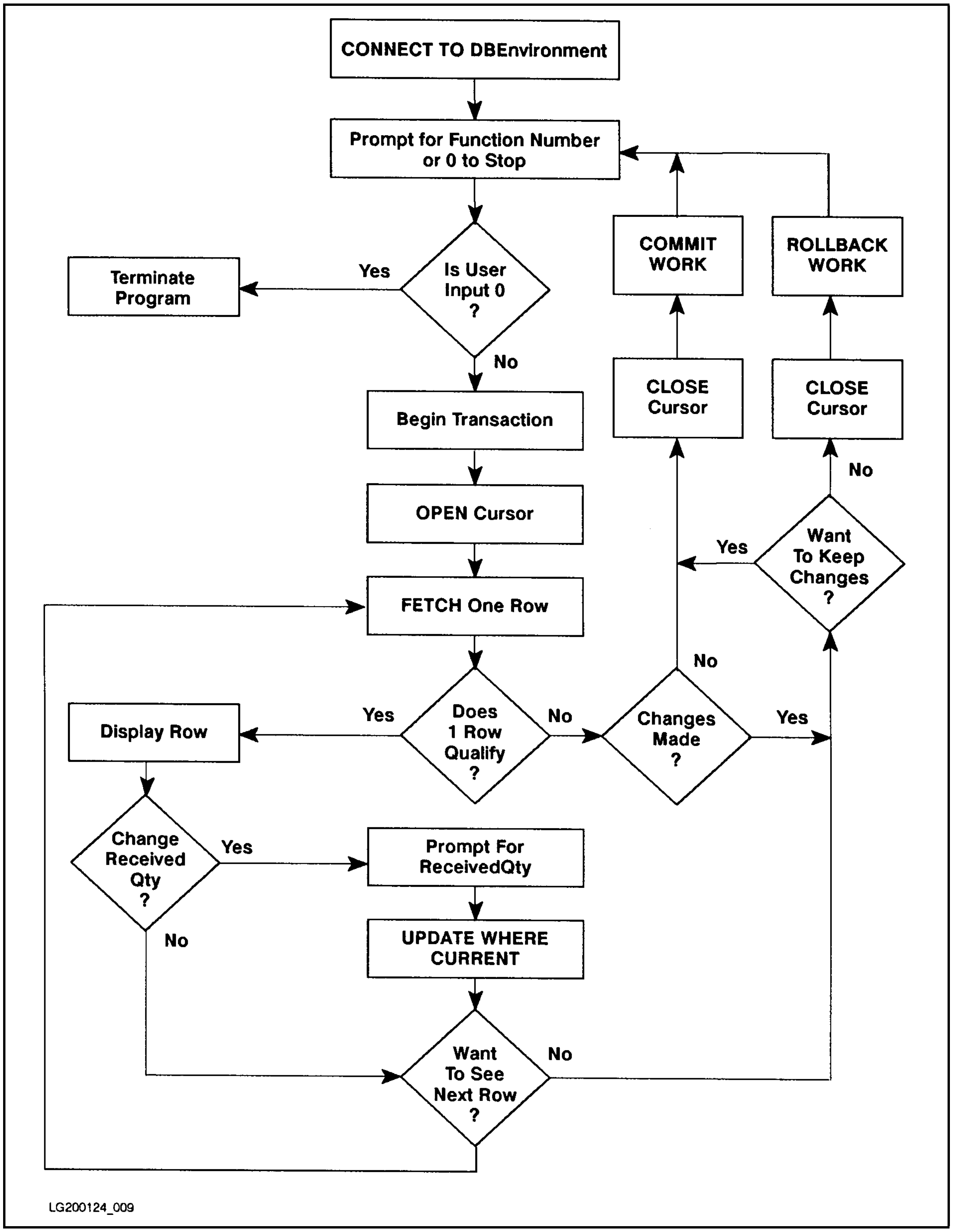


Figure 8-4. Flow Chart of Program forex8

```
:run forex8
Program to UPDATE OrderItems table via a CURSOR -- forex8

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 no more rows qualify
  CLOSE Cursor
  COMMIT WORK or ROLLBACK WORK
  Repeat the above eleven steps until user enters 0
  RELEASE PartsDBE

Declare Cursor OrderReview

CONNECT TO PartsDBE

Enter Order Number or a 0 to STOP >30520
BEGIN WORK

OPEN the Declared Cursor OrderReview

  OrderNumber:           30520
  ItemNumber:                1
  VendPartNumber:  9375
  ReceivedQty:       9

Do you want to change ReceivedQty (Y/N)? > n

Do you want to see another row (Y/N)? > y

  OrderNumber:           30520
  ItemNumber:                2
  VendPartNumber:  9105
  ReceivedQty:       3

Do you want to change ReceivedQty (Y/N)? > y

Enter New ReceivedQty or a 0 for NULL> 15

Do you want to see another row (Y/N)? > y
```

Figure 8-5. Runtime Dialog of Program forex8

```
  OrderNumber:           30520
  ItemNumber:                3
  VendPartNumber:  9135
  ReceivedQty:        3

Do you want to change ReceivedQty (Y/N)? > n

Do you want to see another row (Y/N)? > y

Row not found or no more rows!

Do you want to save your changes (Y/N)? > n

CLOSE the Declared Cursor OrderReview

ROLLBACK WORK
   No Rows Changed!

Enter Order Number or a 0 to STOP > 30510
BEGIN WORK

OPEN the Declared Cursor OrderReview

  OrderNumber:           30510
  ItemNumber:                1
  VendPartNumber:  1001
  ReceivedQty:        3

Do you want to change ReceivedQty (Y/N)? > n

Do you want to see another row (Y/N)? > n

CLOSE the Declared Cursor OrderReview

COMMIT WORK

Enter Order Number or a 0 to STOP > 0
User entered a 0

RELEASE PartsDBE

:
```

Figure 8-5. Runtime Dialog of Program forex8 (page 2 of 2)

```
      PROGRAM forex8
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C* This program illustrates the use of UPDATE WHERE CURRENT  *
C* with a Cursor to update a single row at a time.           *
C* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

      IMPLICIT NONE
      LOGICAL*2 Done, ConnectDBE, FetchUpdate

C             (* Begin SQL Communication Area *)
      EXEC SQL INCLUDE SQLCA

C             (* Beginning of the Main Program *)                 1

      WRITE (*,*) CHAR(27), 'U'
      WRITE (*,*) 'Program to UPDATE OrderItems table via a
     1CURSOR -- forex8'
      WRITE (*,*) ' '
      WRITE (*,*) 'Event List:'
      WRITE (*,*) '  CONNECT TO PartsDBE'
      WRITE (*,*) '  Prompt for Order Number'
      WRITE (*,*) '  BEGIN WORK'
      WRITE (*,*) '  OPEN Cursor'
      WRITE (*,*) '  FETCH a row'
      WRITE (*,*) '  Display the retrieved row'
      WRITE (*,*) '  Prompt for new Received Quantity'
      WRITE (*,*) '  UPDATE row within OrderItems table'
      WRITE (*,*) '  FETCH the next row, if any, with the
     1same Order Num
     1ber'
      WRITE (*,*) '  Repeat the above five steps until no
     1more rows qual
     1ify'
      WRITE (*,*) '  CLOSE Cursor'
      WRITE (*,*) '  COMMIT WORK or ROLLBACK WORK'
      WRITE (*,*) '  Repeat the above eleven steps until
     1user enters 0'
      WRITE (*,*) '  RELEASE PartsDBE'

      CALL DeclareCursor
      WRITE (*,*) ' '

      IF (ConnectDBE()) THEN
        Done = .TRUE.
        DO WHILE (Done)
          Done = FetchUpdate()
        END DO
        CALL TerminateProgram
```

Figure 8-6. Program forex8: Using UPDATE WHERE CURRENT

```
      ELSE
        WRITE (*,*) 'Cannot Connect to your DBEnvironment!'
      ENDIF
      STOP
      END
C             (* End of Main Program *)
C             (* Beginning of the Sub-Routines *)

      SUBROUTINE SQLStatusCheck                                 2
C**** SUBROUTINE SQLStatusCheck checks status of SQL commands
C**** and print HPSQL error messages.

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

      LOGICAL*2 Abort, Check
      INTEGER   DeadLock
      PARAMETER (DeadLock =     -14024)

C             (* Begin Host Variable Declarations *)

      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*120 SQLMessage
      EXEC SQL END DECLARE SECTION

C             (* End Host Variable Declarations *)

      Abort = .FALSE.
      IF (SQLCode .LT. DeadLock) THEN
        Abort = .TRUE.
        WRITE (*,*) 'A serious error has occurred.'
      ENDIF

      Check = .TRUE.
         DO WHILE (Check)
         EXEC SQL SQLEXPLAIN :SQLMessage
         WRITE(*, 100) SQLMessage
100      FORMAT(A120)
         IF (SQLCode .EQ. 0) THEN
           Check = .FALSE.
         ENDIF
      END DO

      IF (Abort) THEN
        CALL TerminateProgram
        STOP 'Program Aborted'
```

Figure 8-6. Program forex8: Using UPDATE WHERE CURRENT (page 2 of 12)

```
      END IF
      RETURN
      END
C             (* End of Subroutine SQLStatusCheck *)

      LOGICAL*2 FUNCTION ConnectDBE()                         3
C**** FUNCTION to connect to PartsDBE

      INTEGER*2  OK
      PARAMETER (OK = 0)

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION

      WRITE (*, *) 'CONNECT TO PartsDBE'

      EXEC SQL CONNECT TO 'PartsDBE'
      ConnectDBE = .TRUE.
      IF (SQLCode .NE. OK) THEN
        ConnectDBE = .FALSE.
        CALL SQLStatusCheck
      ENDIF
      RETURN
      END
C             (* End of Function ConnectDBE *)

      SUBROUTINE TerminateProgram                             4
C**** SUBROUTINE  to release from PartsDBE

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION

      WRITE(*,*) ' '
      WRITE(*,*) 'RELEASE PartsDBE'
      EXEC SQL RELEASE
      RETURN
      END
C             (* End of Subroutine TerminateProgram *)
```

Figure 8-6. Program forex8: Using UPDATE WHERE CURRENT (page 3 of 12)

```
      SUBROUTINE BeginTransaction                              5
C**** SUBROUTINE to begin work
      INTEGER*2  OK
      PARAMETER (OK = 0)

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION

      WRITE (*,*) 'BEGIN WORK'
      EXEC SQL BEGIN WORK
      IF (SQLCode .NE. OK) THEN
        CALL SQLStatusCheck
        CALL TerminateProgram
      ENDIF
      RETURN
      END
C             (* End of Subroutine BeginTransaction *)

      SUBROUTINE CommitWork                                    6
C**** SUBROUTINE to commit work
      INTEGER*2  OK
      PARAMETER (OK = 0)

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION

      WRITE(*,*) 'COMMIT WORK'
      EXEC SQL COMMIT WORK
      IF (SQLCode .NE. OK) THEN
        CALL SQLStatusCheck
        CALL TerminateProgram
      ENDIF
      RETURN
      END
C             (* End of Subroutine CommitWork *)
```

Figure 8-6. Program forex8: Using UPDATE WHERE CURRENT (page 4 of 12)

```
      SUBROUTINE RollBackWork                                          7
C**** SUBROUTINE to RollBack Work
      INTEGER*2  OK
      PARAMETER (OK = 0)

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION

      WRITE(*,*) 'ROLLBACK WORK'
      EXEC SQL ROLLBACK WORK
      IF (SQLCode .NE. OK) THEN
        CALL SQLStatusCheck
        CALL TerminateProgram
      ENDIF
      RETURN
      END
C             (* End of Subroutine RollBackWork *)

      SUBROUTINE DisplayRow (OrderNumber,ItemNumber,VendPartNumber,
     1              ReceivedQty, ReceivedQtyInd)                       8
C**** SUBROUTINE to display OrderItems table rows

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

C             (* Begin Host Variable Declarations *)

      EXEC SQL BEGIN DECLARE SECTION
      INTEGER           OrderNumber
      INTEGER           ItemNumber
      CHARACTER*16      VendPartNumber
      INTEGER           ReceivedQty
      SQLIND            ReceivedQtyInd
      CHARACTER*120     SQLMessage
      EXEC SQL END DECLARE SECTION

C             (* End Host Variable Declarations *)
```

Figure 8-6. Program forex8: Using UPDATE WHERE CURRENT (page 5 of 12)

```
      WRITE(*,*) ' '
      WRITE(*, '(''  OrderNumber:      '',I10)') OrderNumber
      WRITE(*, '(''  ItemNumber:       '',I10)') ItemNumber
      WRITE(*, '(''  VendPartNumber:   '',A16)') VendPartNumber
      IF (ReceivedQtyInd .LT. 0) THEN
        WRITE(*,*) ' ReceivedQty is NULL'
      ELSE
        WRITE(*, '(''  ReceivedQty:    '',I5)') ReceivedQty
      ENDIF
      WRITE(*,*) ' '
      RETURN
      END
C             (* End of Subroutine DisplayRow *)


      SUBROUTINE DeclareCursor                                      9
C**** SUBROUTINE to declare the Cursor

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

C             (* Begin Host Variable Declarations *)

      EXEC SQL BEGIN DECLARE SECTION
      INTEGER            OrderNumber
      INTEGER            ItemNumber
      CHARACTER*16       VendPartNumber
      INTEGER            ReceivedQty
      SQLIND             ReceivedQtyInd
      CHARACTER*120      SQLMessage
      EXEC SQL END DECLARE SECTION

C             (* End Host Variable Declarations *)

      WRITE (*,*) ' '
      WRITE (*,*) 'Declare Cursor OrderReview'
      WRITE (*,*) ' '
      EXEC SQL DECLARE OrderReview
```

Figure 8-6. Program forex8: Using UPDATE WHERE CURRENT (page 6 of 12)

```
     1          CURSOR FOR
     2          SELECT OrderNumber,
     3                 ItemNumber,
     4                 VendPartNumber,
     5                 ReceivedQty
     6            FROM PurchDB.OrderItems
     7           WHERE OrderNumber = :OrderNumber
     8             AND VendPartNumber IS NOT NULL
     9   FOR UPDATE OF ReceivedQty

      RETURN
      END
C             (* End of Subroutine DeclareCursor *)

      LOGICAL*2 FUNCTION OpenCursor(OrderNumber, ItemNumber,          10
     1                   VendPartNumber, ReceivedQty, ReceivedQtyInd)
C**** FUNCTION to open the Cursor

      INTEGER    OK
      PARAMETER (OK = 0)

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

C             (* Begin Host Variable Declarations *)

      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION

C             (* End Host Variable Declarations *)

      OpenCursor = .TRUE.
      WRITE (*,*) ' '
      WRITE (*,*) 'OPEN the Declared Cursor OrderReview'
      WRITE (*,*) ' '
      EXEC SQL OPEN OrderReview

      IF (SQLCode .NE. OK) THEN
        OpenCursor = .FALSE.
        CALL SQLStatusCheck
        CALL RollBackWork
      ENDIF
      RETURN
      END

C**** (End of Function OpenCursor)
```

Figure 8-6. Program forex8: Using UPDATE WHERE CURRENT (page 7 of 12)

```
      SUBROUTINE CloseCursor                                          11
C**** SUBROUTINE to close the Cursor

      INTEGER    OK
      PARAMETER (OK = 0)

C             (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

C             (* Begin Host Variable Declarations *)

      EXEC SQL BEGIN DECLARE SECTION
      EXEC SQL END DECLARE SECTION

C             (* End Host Variable Declarations *)

      WRITE (*,*) ' '
      WRITE (*,*) 'CLOSE the Declared Cursor OrderReview'
      WRITE (*,*) ' '
      EXEC SQL CLOSE OrderReview

      IF (SQLCode .NE. OK) THEN
        CALL SQLStatusCheck
        CALL TerminateProgram
      ENDIF
      RETURN
      END

C**** (End of Subroutine CloseCursor)


      SUBROUTINE DisplayUpdate(OrderNumber, ItemNumber,               12
     1           VendPartNumber, ReceivedQty, ReceivedQtyInd,
     2           RowCounter,Fetch)
C**** SUBROUTINE to Display and Update a row from PurchDB.OrderItems

      LOGICAL*2     Fetch
      CHARACTER     Response
      INTEGER       NotFound,MultipleRows,OK,RowCounter
      PARAMETER    (NotFound = 100,
     1              MultipleRows = -10002,
     2              OK = 0)

C             (* Begin SQL Communication Area *)
```

Figure 8-6. Program forex8: Using UPDATE WHERE CURRENT (page 8 of 12)

```
      EXEC SQL INCLUDE SQLCA

C             (* Begin Host Variable Declarations *)

      EXEC SQL BEGIN DECLARE SECTION
      INTEGER           OrderNumber
      INTEGER           ItemNumber
      CHARACTER*16      VendPartNumber
      INTEGER           ReceivedQty
      SQLIND            ReceivedQtyInd
      CHARACTER*120     SQLMessage
      EXEC SQL END DECLARE SECTION

      CALL DisplayRow(OrderNumber, ItemNumber, VendPartNumber,
     1               ReceivedQty, ReceivedQtyInd)

      WRITE (*,100)                                                 12A
100   FORMAT (/$, ' Do you want to change ReceivedQty (Y/N)? > ')
      READ (*, 110) Response
110   FORMAT(A1)
      IF ((Response .EQ. 'Y') .OR. (Response .EQ. 'y')) THEN
        WRITE (*,120)
120     FORMAT (/$,'Enter New ReceivedQty or a 0 for NULL > ')
        READ (*,130) ReceivedQty
130     FORMAT(I5)
        IF (ReceivedQty .EQ. 0) THEN
          ReceivedQtyInd = -1
        ELSE
          ReceivedQtyInd = 0
        ENDIF
        EXEC SQL UPDATE PurchDB.OrderItems                          12B
     1              SET ReceivedQty = :ReceivedQty :ReceivedQtyInd
     2            WHERE CURRENT OF OrderReview

      IF (SQLCode .NE. OK) THEN
        CALL SQLStatusCheck
        ELSE
          RowCounter = RowCounter +1
        ENDIF
      ENDIF
```

Figure 8-6. Program forex8: Using UPDATE WHERE CURRENT (page 9 of 12)

```
      WRITE (*,140)
140   FORMAT (/$, 'Do you want to see another row (Y/N)? > ')
      READ (*, 150) Response
150   FORMAT (A1)
      IF ((Response .EQ. 'N') .OR. (Response .EQ. 'n')) THEN
       IF (RowCounter .GT. 0) THEN
         WRITE (*,160)
160      FORMAT (/$, 'Do you want to save your changes (Y/N)? >')
         READ (*, 170) Response
170      FORMAT (A1)
         IF ((Response .EQ. 'N') .OR. (Response .EQ. 'n')) THEN
           CALL CloseCursor
           CALL RollBackWork
           Fetch = .FALSE.
           WRITE (*,*) 'No Row(s) Changed! '
         ELSE
           CALL CloseCursor
           CALL CommitWork
           Fetch = .FALSE.
           WRITE (*, '(''   Row(s) Changed:  '',I2)') RowCounter
         ENDIF
       ELSEIF (RowCounter .EQ. 0) THEN
         CALL CloseCursor
         CALL CommitWork
         Fetch = .FALSE.
       ENDIF
     ENDIF

   RETURN
   END
C**(End of Subroutine DisplayUpdate)

   LOGICAL*2 FUNCTION FetchUpdate()                                  13
C**FUNCTION to Fetch rows from PurchDB.OrderItems table.

   CHARACTER  Response
   LOGICAL*2  Fetch,OpenCursor
   INTEGER    NotFound,MultipleRows,OK,RowCounter
   PARAMETER (NotFound = 100,
  1           MultipleRows = -10002,
  2           OK = 0)
```

Figure 8-6. Program forex8: Using UPDATE WHERE CURRENT (page 10 of 12)

```
C            (* Begin SQL Communication Area *)

      EXEC SQL INCLUDE SQLCA

C            (* Begin Host Variable Declarations *)

      EXEC SQL BEGIN DECLARE SECTION
      INTEGER           OrderNumber
      INTEGER           ItemNumber
      CHARACTER*16      VendPartNumber
      INTEGER           ReceivedQty
      SQLIND            ReceivedQtyInd
      CHARACTER*120     SQLMessage
      EXEC SQL END DECLARE SECTION

C            (* End Host Variable Declarations *)

      RowCounter = 0
      FetchUpdate = .TRUE.
      WRITE (*,100)                                                   13A
100   FORMAT(/$, 'Enter Order Number or a 0 to stop > ')
      READ (*,110) OrderNumber
110   FORMAT(I10)
      IF (OrderNumber .NE. 0) THEN
        CALL BeginTransaction
        IF (OpenCursor(OrderNumber,ItemNumber,
     1      VendPartNumber,ReceivedQty,ReceivedQtyInd)) THEN
          Fetch = .TRUE.
          DO WHILE (Fetch)
        EXEC SQL FETCH  OrderReview                                   13B
     1           INTO  :OrderNumber,
     1                 :ItemNumber,
     2                 :VendPartNumber,
     3                 :ReceivedQty :ReceivedQtyInd

      IF (SQLCode .EQ. OK) THEN
        CALL DisplayUpdate(OrderNumber,ItemNumber,VendPartNumber,
     1       ReceivedQty, ReceivedQtyInd, RowCounter,Fetch)
      ELSEIF (SQLCode .EQ. NotFound) THEN
        Fetch = .FALSE.
        WRITE (*,*) ' '
        WRITE (*,*) 'Row not found or no more rows!'
```

Figure 8-6. Program forex8: Using UPDATE WHERE CURRENT (page 11 of 12)

```
        IF (RowCounter .GT. 0) THEN
          WRITE (*,120)
120       FORMAT (/$, 'Do you want to save your changes (Y/N)? > ' )
          READ (*,130) Response
130       FORMAT(A1)
          IF ((Response .EQ. 'N') .OR. (Response .EQ. 'n')) THEN
            CALL CloseCursor
            CALL RollBackWork
            WRITE (*,*) 'No Row(s) Changed! '
          ELSE
            CALL CloseCursor
            CALL CommitWork
          WRITE (*, '(''   Row(s) Changed:  '',I2)') RowCounter
          ENDIF
        ELSEIF (RowCounter .EQ. 0) THEN
          CALL CloseCursor
          CALL CommitWork
        ENDIF
      ELSEIF (SQLCode .EQ. MultipleRows) THEN
        Fetch = .FALSE.
        WRITE(*,*) ' '
        WRITE (*,*) 'WARNING: More than one row qualifies!'
        CALL SQLStatusCheck
      ELSE
        Fetch = .FALSE.
        CALL SQLStatusCheck
        CALL CloseCursor
        CALL RollBackWork
      ENDIF
        END DO
        ELSE
          FetchUpdate = .FALSE.
          WRITE (*,*) 'Failed to Open Cursor'
        ENDIF
      ELSE
        FetchUpdate = .FALSE.
        WRITE (*,*) 'User entered an 0'
      ENDIF

      RETURN
      END
C             (* End of Subroutine FetchUpdate *)
```

Figure 8-6. Program forex8: Using UPDATE WHERE CURRENT (page 12 of 12)

9

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.
- Programs Using Dynamic Operations.

Review of Preprocessing Events

All embedded SQL statements must be preprocessed before they can be executed. Preprocessing may be done by running the FORTRAN 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 can be preprocessed at run time:

```
BEGIN DECLARE SECTION          FETCH
CLOSE CURSOR                   INCLUDE
DECLARE CURSOR                 OPEN CURSOR
DELETE WHERE CURRENT           PREPARE
DESCRIBE                       SELECT
END DECLARE SECTION            SQLEXPLAIN
EXECUTE                        UPDATE WHERE CURRENT
EXECUTE IMMEDIATE              WHENEVER
```

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 FORTRAN preprocessor. The module remains in the system catalog until you execute the DROP MODULE command or invoke the preprocessor with the DROP 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 9-1 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.

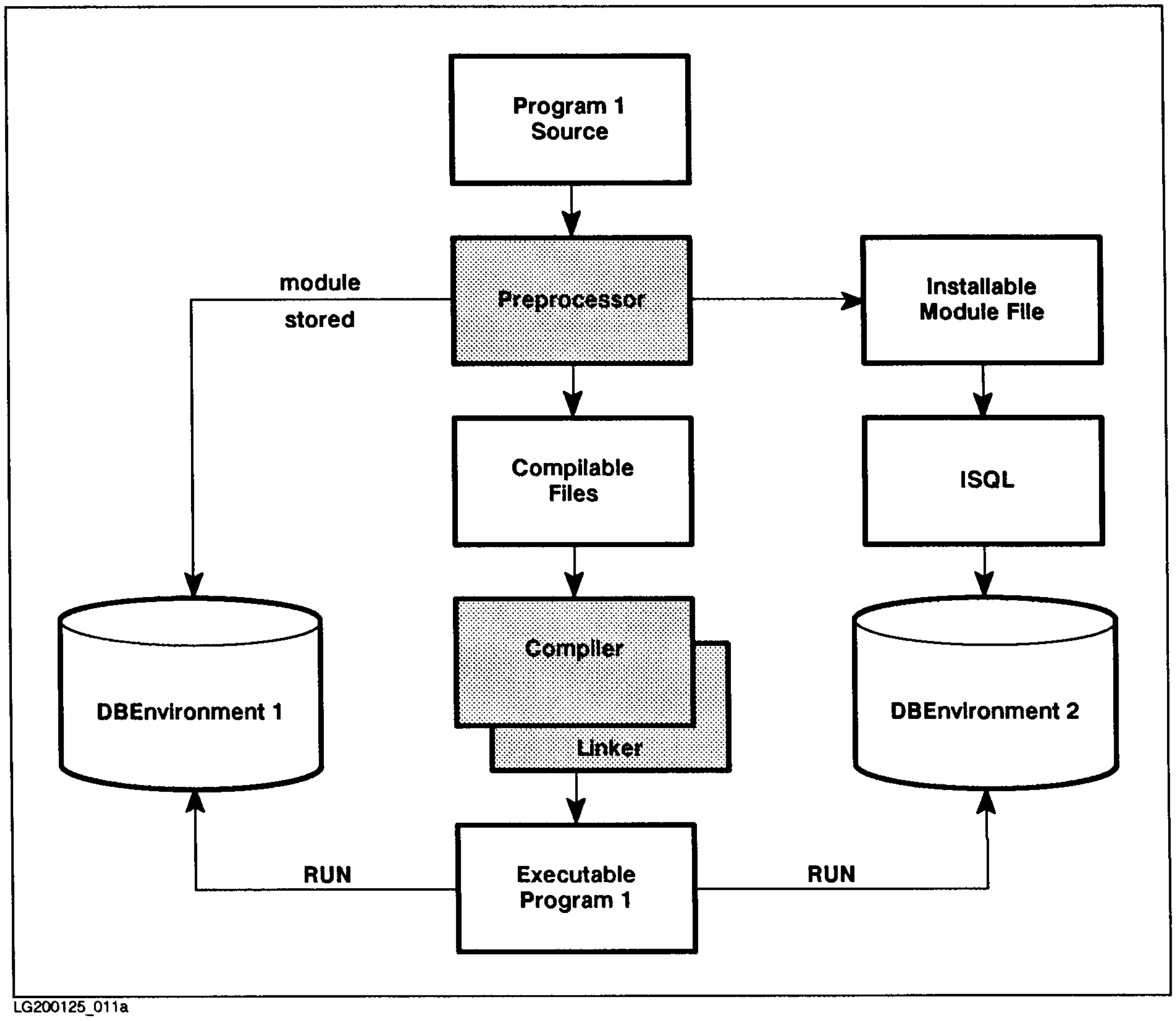


Figure 9-1. Creation and Use of a Program that has a Stored Module

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 9-2 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.

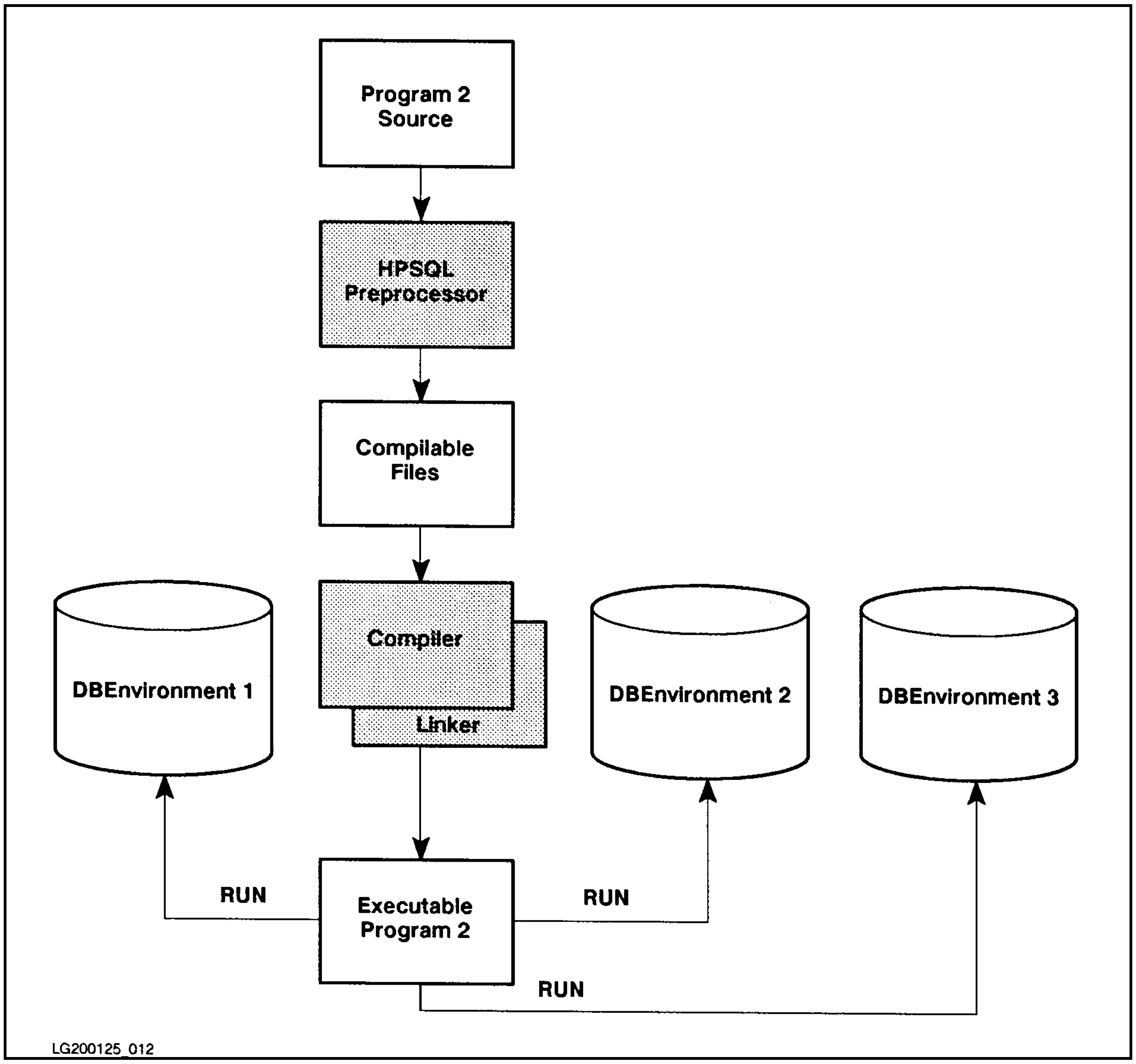


Figure 9-2. Creation and Use of a Program that has No Stored Module

Examples of Non-Dynamic and Dynamic SQL Statements

The following example shows an embedded SQL statement that is coded so as to generate a stored section before run time:

```
EXEC SQL UPDATE STATISTICS FOR TABLE PurchDB.Parts;
```

When you run the preprocessor on a source file containing this statement, a permanent section will be stored in the appropriate DBEnvironment.

The following example shows an SQL statement that is coded so as to generate a temporary section at run time:

```
DynamicCommand := 'UPDATE STATISTICS FOR TABLE PurchDB.Parts;';
EXEC SQL PREPARE MyCommand FROM :DynamicCommand;
EXEC SQL EXECUTE MyCommand;
```

In this case, the SQL statement is stored in a host variable which is passed to ALLBASE/SQL in the PREPARE statement at run time. A temporary section is then created and executed, and the section is not stored in the DBEnvironment.

Why Use Dynamic Preprocessing?

In some cases, it may *not* be desirable to preprocess an SQL command before run time:

- You may need to code an application that permits ad hoc queries requiring that SQL commands be entered by the user at run time. (ISQL is an example of an ad hoc query facility in which the command the user will submit is completely unknown at programming time.)
- You may need more specialized applications requiring SQL commands that are defined partly at programming time and partly by the user at run time. An application may, for example, perform UPDATE STATISTICS operations on tables the user specifies at run time.
- You may wish to run an application on different DBEnvironments at different times without the need to permanently store sections in those DBEnvironments.
- You may wish to code only one dynamic command (a CONNECT, for instance) and then preprocess or install the same application in several different DBEnvironments.

Passing Dynamic Commands to ALLBASE/SQL

A dynamic command is passed to ALLBASE/SQL either as a string literal or as a host variable containing a string. It must be terminated with a semicolon. The maximum length for such a string is 2048 bytes.

To pass a dynamic command that can be completely defined at programming time, you can use a delimited string:

```
  EXEC SQL PREPARE MyCommand FROM 'UPDATE STATISTICS FOR TABLE PurchDB.Parts;';
```

or

```
   EXEC SQL EXECUTE IMMEDIATE 'UPDATE STATISTICS FOR TABLE PurchDB.Parts;';
```

To pass a dynamic command that cannot be completely defined at programming time, you use a host variable declared as a string of characters:

```
   CHARACTER*2048    DynamicHostVar
   .
   .
   EXECUTE IMMEDIATE :DynamicHostVar
```

Understanding Dynamic Operations

Dynamic Non-Queries are the only type of dynamic operations available to FORTRAN programmers. Non-queries 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.

The following paragraphs examine dynamic non-queries.

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 9-3 illustrates a procedure hosting a dynamic UPDATE STATISTICS command that can be handled in this fashion.

Subroutine UpdateStatistics $\boxed{1}$ prompts the user for a table name $\boxed{2}$. The table name entered is assigned to the host variable CmdLine $\boxed{3}$ to complete the UPDATE STATISTICS command. After the command is prepared and executed $\boxed{4}$, the transaction is terminated with a COMMIT WORK command $\boxed{5}$ or a ROLLBACK WORK command $\boxed{6}$, 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.

```
      SUBROUTINE UpdateStatistics                                  1


      CHARACTER*50      TableName

      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*1024    CmdLine
      EXEC SQL END DECLARE SECTION



      DO WHILE (TableName .NE. '/')

      WRITE (*,*) 'Enter table name or a / to stop > '            2
      READ(6,100) TableName
100   FORMAT(A50)
      IF (TableName .NE. '/') THEN

      CmdLine ='UPDATE STATISTICS FOR TABLE '// TableName //  ;'   3

          EXEC SQL EXECUTE IMMEDIATE :CmdLine                     4

          IF (SQLCode .EQ. 0) THEN
            EXEC SQL COMMIT WORK                                  5
          ELSE
            EXEC SQL ROLLBACK WORK                                6
          ENDIF

      ENDIF                  (* END OF IF TABLENAME *)

      END DO

      RETURN
      END                    (* END OF UPDATESTATISTICS PROCEDURE *)
```

Figure 9-3. Procedure Hosting Dynamic Non-Query Commands

Using PREPARE and EXECUTE

Use the PREPARE command 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;

   I = MaxIterations
   DO WHILE (I .NE. 0)
   EXEC SQL EXECUTE MyNonQuery;
   I = I - 1
   END DO
```

As soon as you process a COMMIT WORK or ROLLBACK WORK command, the temporary section is deleted.

The program examined later in this chapter under "Sample Program Using PREPARE and EXECUTE" illustrates how to handle PREPARE and EXECUTE.

Programs Using Dynamic Operations

The rest of this chapter contains sample programs that illustrate the use of dynamic preprocessing techniques for queries. There are two complete programs:

- forex9a, which contains statements for using EXECUTE IMMEDIATE.
- forex9b, which contains statements for using PREPARE and EXECUTE.

For each program, there is a description of the code, a display of the runtime dialog with user input, and a listing.

Sample Program Using EXECUTE IMMEDIATE

To preprocess and execute a dynamic command in only one step, you use the EXECUTE IMMEDIATE command:

```
        EXEC SQL EXECUTE IMMEDIATE :DynamicCommand
```

Program forex9a, whose runtime dialog is shown in Figure 9-4 and whose source code is given in Figure 9-5, can be used to execute the UPDATE STATISTICS command on any table in any DBEnvironment. This program prompts for both the DBEnvironment name and the name of tables for which to execute the UPDATE STATISTICS command. The UPDATE STATISTICS command is handled by using the EXECUTE IMMEDIATE command.

The main program $\boxed{1}$ first performs function *ConnectDBE* $\boxed{2}$ to start a DBE session. *ConnectDBE* prompts for the name of a DBEnvironment $\boxed{2A}$. A READ command places the DBEnvironment name in the host variable *DBEnvironment*, and the CONNECT command is then executed. The program performs implicit status checking, which results in calls to subroutine *SQLStatusCheck* $\boxed{8}$ if an error occurs. Note that it is necessary either to include the appropriate label in each subsequent subprogram unit that follows the WHENEVER *Condition* GOTO commands or to turn implicit status checking off.

The program then performs subroutine *BeginWork* $\boxed{3}$ to begin a transaction. *BeginWork* executes a BEGIN WORK command. Function *Update* $\boxed{7}$ is then performed to execute the UPDATE STATISTICS command. *Update* declares the host variables used to hold information about the dynamic command. The static part of the UPDATE STATISTICS command is placed into the variable *Static*, and then the user is prompted for the name of the table to be updated $\boxed{7A}$. The *TableName* is then concatenated with the rest of the UPDATE STATISTICS command in *Static* and placed into the variable *Command*. The full UPDATE STATISTICS command is then preprocessed and executed with the EXECUTE IMMEDIATE command $\boxed{7B}$. At the end of the function, implicit status checking is turned off.

If function *Update* evaluates to TRUE, it prompts the user for another table name. Function *Update* terminates when *Update* is set to FALSE by the user's entering a slash in response to the prompt for a table name. When *Update* evaluates to FALSE, subroutine *CommitWork* $\boxed{4}$ is performed. *CommitWork* executes a COMMIT WORK command, then subroutine *ReleaseDBE* is performed $\boxed{5}$. *ReleaseDBE* executes a RELEASE command to terminate the DBE session. After ReleaseDBE has executed, the program terminates.

When ALLBASE/SQL returns a negative value or a value of 100 in SQLCode following the execution of the embedded SQL commands, subroutine *SQLStatusCheck* $\boxed{8}$ is performed. This subroutine writes out messages based on the values of SQLCode and SQLWARN, then calls subroutine *SQLExplain* $\boxed{6}$ to display one or more messages. Subroutine *SQLExplain* executes the SQLEXPLAIN command and prints out the error message. If an error is very serious (SQLCode < -14024), a flag named *Abort* is set, and subroutines *CommitWork* $\boxed{4}$ and *ReleaseDBE* $\boxed{5}$ are performed before the program is terminated.

```
Program to illustrate the EXECUTE IMMEDIATE command -- forex9a
Event List:
  Prompt for the DBEnvironment Name
  CONNECT TO the DBEnvironment
  BEGIN WORK
  Prompt for the table name
  EXECUTE IMMEDIATE UPDATE STATISTICS command
  COMMIT WORK
  Repeat the above 3 steps until the user enters a /
  RELEASE from DBEnvironment
  Repeat the above 8 steps until the user enters a /
  Terminate the Program

Enter DBEnvironment to CONNECT TO or a / to STOP > PartsDBE
CONNECT TO DBEnvironment
Successful CONNECT
BEGIN WORK

Enter Table Name or a / to Stop > PurchDB.Vendors
Table to Update - PurchDB.Vendors
Command - UPDATE STATISTICS FOR TABLE   PurchDB.Vendors          ;
SQL command executed successfully.

Enter Table Name or a / to Stop > System.Table
Table to Update - System.Table
Command - UPDATE STATISTICS FOR TABLE   System.Table             ;
SQL command executed successfully.

Enter Table Name or a / to Stop > PurchDB.VendorStatistics
Table to Update - PurchDB.VendorStatistics
Command - UPDATE STATISTICS FOR TABLE   PurchDB.VendorStatistics ;
HPSQL error!
Call SQLExplain
Command UPDATE STATISTICS is not allowed
     for views (PURCHDB.VENDORSTATISTICS)
SQL command not successfully executed.

Enter Table Name or a / to Stop > /
No more tables to update.
COMMIT WORK
RELEASE DBEnvironment

Enter DBEnvironment to CONNECT TO or a / to STOP > /
Terminating the Program!
```

Figure 9-4. Runtime Dialog for Program forex9a

```
      PROGRAM forex9a

C     * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C     * This program illustrates the use of SQL dynamic      *
C     * non-query commands executed from a FORTRAN program. *
C     * This program demonstrates the use of the EXECUTE     *
C     * IMMEDIATE command.                                   *
C     * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        IMPLICIT NOTE
        LOGICAL*2  Update, Test, ConnectDBE

C       (**** Begin SQL Communication Area ****)
        EXEC SQL INCLUDE SQLCA

C       (**** begin Host Variable Declarations ****)
        EXEC SQL BEGIN DECLARE SECTION
        EXEC SQL END DECLARE SECTION

C       (**** Beginning of Main Program ****)                        1
        WRITE (*,*) CHAR(27), 'U'
        WRITE (*,*) 'Program to illustrate the EXECUTE
       1IMMEDIATE command -- forex9a'
        WRITE (*,*) ' '
        WRITE (*,*) 'Event List:'
        WRITE (*,*) '  Prompt for the DBEnvironment Name:'
        WRITE (*,*) '  CONNECT TO the DBEnvironment'
        WRITE (*,*) '  BEGIN WORK'
        WRITE (*,*) '  Prompt for the table name:'
        WRITE (*,*) '  EXECUTE IMMEDIATE UPDATE STATISTICS command'
        WRITE (*,*) '  COMMIT WORK'
        WRITE (*,*) '  Repeat the above 3 steps until the
       1user enters a /'
        WRITE (*,*) '  RELEASE from DBEnvironment'
        WRITE (*,*) '  Repeat the above 8 steps until the
       1user enters a /'
        WRITE (*,*) '  Terminate the Program.'

        DO WHILE (ConnectDBE())
          CALL BeginWork
          Test = .TRUE.
          DO WHILE (Test)
            Test = Update()
          END DO
          CALL CommitWork
          CALL ReleaseDBE
```

Figure 9-5. Program forex9a: Sample Program Using EXECUTE IMMEDIATE

```
        END DO
        WRITE (*,*) 'Connect was Unsuccessful!'
C
        END
C       (* Beginning of the Subroutines *)

        LOGICAL*2 FUNCTION ConnectDBE()                            2
C       (**** Subroutine to connect to user
        entered DBEnvironment ****)

C       (* Begin Communication Area *)
        EXEC SQL INCLUDE SQLCA

C       (**** Begin Host Variable Declarations ****)
        EXEC SQL BEGIN DECLARE SECTION
        CHARACTER*80      DBEnvironment
        CHARACTER*80      SQLMessage
        EXEC SQL END DECLARE SECTION

        EXEC SQL WHENEVER SQLERROR GOTO 1000
        EXEC SQL WHENEVER SQLWARNING GOTO 1000
        EXEC SQL WHENEVER NOT FOUND GOTO 1000

        ConnectDBE = .FALSE.
        DBEnvironment = '/'

        WRITE (*,100)                                              2A
100     FORMAT (/$,'Enter DBEnvironment to CONNECT TO or a /
       1to STOP > ')
        READ (*,110) DBEnvironment
110     FORMAT(A80)

        IF (DBEnvironment .EQ. '/') THEN
          WRITE (*,*) ' '
          WRITE (*,*) 'Terminating the Program!'
          STOP
        ELSE
          WRITE (*,*) ' '
          WRITE (*,*)  'CONNECT TO DBEnvironment'
          EXEC SQL CONNECT TO :DBEnvironment
        ENDIF
        IF (SQLCode .NE. 0) THEN
          GOTO 1000
        ENDIF
        ConnectDBE = .TRUE.
        WRITE (*,*)  'Successful CONNECT'
        GOTO 2000
```

Figure 9-5. Program forex9a: Sample Program Using EXECUTE IMMEDIATE (page 2 of 6)

```
 1000   CALL SQLStatusCheck
        ConnectDBE = .FALSE.
        CALL ReleaseDBE
 2000   RETURN
        EXEC SQL WHENEVER SQLERROR CONTINUE
        EXEC SQL WHENEVER SQLWARNING CONTINUE
        EXEC SQL WHENEVER NOT FOUND CONTINUE
        END

        SUBROUTINE BeginWork                              3
 C      (**** Subroutine to Begin a Transaction ****)

 C      (* Begin Communication Area *)
        EXEC SQL INCLUDE SQLCA

 C      (**** Begin Host Variable Declarations ****)
        EXEC SQL BEGIN DECLARE SECTION
        EXEC SQL END DECLARE SECTION

        EXEC SQL WHENEVER SQLERROR GOTO 1000
        EXEC SQL WHENEVER SQLWARNING GOTO 1000
        EXEC SQL WHENEVER NOT FOUND GOTO 1000

        WRITE (*,*) 'BEGIN WORK'
        EXEC SQL BEGIN WORK
        GOTO 2000
 1000   CALL SQLStatusCheck
        CALL ReleaseDBE
 2000   RETURN
        EXEC SQL WHENEVER SQLERROR CONTINUE
        EXEC SQL WHENEVER SQLWARNING CONTINUE
        EXEC SQL WHENEVER NOT FOUND CONTINUE
        END

        SUBROUTINE CommitWork                             4
 C      (**** Subroutine to Commit Work ****)

 C      (* Begin Communication Area *)
        EXEC SQL INCLUDE SQLCA

 C      (**** Begin Host Variable Declarations ****)
        EXEC SQL BEGIN DECLARE SECTION
        EXEC SQL END DECLARE SECTION

        EXEC SQL WHENEVER SQLERROR GOTO 1000
        EXEC SQL WHENEVER SQLWARNING GOTO 1000
        EXEC SQL WHENEVER NOT FOUND GOTO 1000
```

Figure 9-5. Program forex9a: Sample Program Using EXECUTE IMMEDIATE (page 3 of 6)

```
         WRITE (*,*)  'COMMIT WORK'
         EXEC SQL COMMIT WORK
         GOTO 2000
 1000    CALL SQLStatusCheck
         CALL ReleaseDBE
 2000    RETURN
         EXEC SQL WHENEVER SQLERROR CONTINUE
         EXEC SQL WHENEVER SQLWARNING CONTINUE
         EXEC SQL WHENEVER NOT FOUND CONTINUE
         END

         SUBROUTINE ReleaseDBE                                    5
 C       (**** Subroutine to Release the DBEnvironment ****)

 C       (* Begin Communication Area *)
         EXEC SQL INCLUDE SQLCA

 C       (**** Begin Host Variable Declarations ****)
         EXEC SQL BEGIN DECLARE SECTION
         EXEC SQL END DECLARE SECTION

         WRITE (*,*) 'RELEASE DBEnvironment'
         EXEC SQL RELEASE
         RETURN
         END

         SUBROUTINE SQLExplain                                    6
 C       (**** Subroutine to CALL SQLEXPLAIN ****)

 C       (* Begin Communication Area *)
         EXEC SQL INCLUDE SQLCA

 C       (**** Begin Host Variable Declarations ****)
         EXEC SQL BEGIN DECLARE SECTION
         CHARACTER*80     SQLMessage
         EXEC SQL END DECLARE SECTION

         WRITE (*,*)  'Call SQLExplain'
         EXEC SQL SQLEXPLAIN :SQLMessage
         WRITE (*,*) ' '
         WRITE (*,*) SQLMessage
         RETURN
         END

         LOGICAL*2 FUNCTION Update()                              7
 C       (**** Function to Update the user entered tables ****)
         Static = 'UPDATE STATISTICS FOR TABLE'
```

Figure 9-5. Program forex9a: Sample Program Using EXECUTE IMMEDIATE (page 4 of 6)

```
C        (**** Begin SQL Communication Area ****)
         EXEC SQL INCLUDE SQLCA

         CHARACTER*30     Static

C        (**** Begin Host Variable Declarations ****)
         EXEC SQL BEGIN DECLARE SECTION
         CHARACTER*50     TableName
         CHARACTER*81     Command
         CHARACTER*80     SQLMessage
         EXEC SQL END DECLARE SECTION

         EXEC SQL WHENEVER SQLERROR GOTO 1000
         EXEC SQL WHENEVER SQLWARNING GOTO 1000
         EXEC SQL WHENEVER NOT FOUND GOTO 1000

        WRITE (*,100)                                          7A
100     FORMAT (/$, 'Enter Table Name or a / to Stop > ')
        READ (*,110) TableName
110     FORMAT(A50)
        IF (TableName .EQ. '/') THEN
          WRITE (*,*) 'No more tables to update.'
          Update = .FALSE.
        ELSE
          WRITE(*, '(''Table to Update - '',A25)') TableName
          Command = Static // TableName // ';'
          WRITE (*, '(''Command - '',A56)') Command
          EXEC SQL EXECUTE IMMEDIATE :Command                  7B
          WRITE (*,*) 'SQL command executed successfully.'
          Update = .TRUE.
        ENDIF
        GOTO 2000
1000    CALL SQLStatusCheck
        WRITE (*,*) 'SQL command not successfully executed.'
2000    CONTINUE
        RETURN
        EXEC SQL WHENEVER SQLERROR CONTINUE
        EXEC SQL WHENEVER SQLWARNING CONTINUE
        EXEC SQL WHENEVER NOT FOUND CONTINUE
        END

        SUBROUTINE SQLStatusCheck                              8
C       (**** Subroutine SQLStatusCheck checks status
         of SQL commands ****)
C       (**** and print HPSQL error messages. ****)
        LOGICAL*2 Abort, Check
        INTEGER MultipleRows, DeadLock, NotFound
        PARAMETER (MultipleRows = -10002,
```

Figure 9-5. Program forex9a: Sample Program Using EXECUTE IMMEDIATE (page 5 of 6)

```
     1          DeadLock =     -14024,
     2          NotFound =        100)

C      (**** Begin Communication Area ****)
       EXEC SQL INCLUDE SQLCA

C      (**** Begin Host Variable Declarations ****)
       EXEC SQL BEGIN DECLARE SECTION
       CHARACTER*80     SQLMessage
       EXEC SQL END DECLARE SECTION

       Abort = .FALSE.
       Check = .TRUE.

        IF (SQLWarn(0) .EQ. 'W') THEN
          WRITE (*,*) 'HPSQL Warning!'
        ELSEIF (SQLCode .EQ. NotFound) THEN
          WRITE (*,*) 'No record found for this PartNumber!'
        ELSEIF (SQLCode .EQ. MultipleRows) THEN
          WRITE (*,*) 'Multiple records exit for this PartNumber!'
        ELSEIF (SQLCode .EQ. DeadLock) THEN
          Abort = .TRUE.
          WRITE (*,*) 'A DEADLOCK has occurred!'
        ELSEIF (SQLCode .LT. DeadLock)THEN
          Abort = .TRUE.
          WRITE (*,*) 'Serious ALLBASE/SQL error!'
        ELSEIF (SQLCode .LT. 0) THEN
          WRITE (*,*) 'ALLBASE/SQL error!'
        ENDIF

        DO WHILE (Check)
          CALL SQLExplain

          IF (SQLCode .EQ. 0) THEN
            Check = .FALSE.
          ENDIF
        END DO

        IF (Abort) THEN
          CALL CommitWork
          CALL ReleaseDBE
        ENDIF

        RETURN
        END
```

Figure 9-5. Program forex9a: Sample Program Using EXECUTE IMMEDIATE (page 6 of 6)

Sample Program Using PREPARE and EXECUTE

To prepare a dynamic command for execution later during the current transaction, you use the PREPARE command to dynamically preprocess the command. 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 MyCommand FROM :DynamicCommand
.
.
EXEC SQL EXECUTE :DynamicCommand
```

As soon as you process a COMMIT WORK or ROLLBACK WORK command, the temporary section is deleted.

Figure 9-6 illustrates the runtime dialog for a program that uses the PREPARE and EXECUTE commands, program forex9b. The program starts a DBE session in the DBEnvironment named *PartsDBE*, then prompts for entry of an SQL command. After the user enters a command, the program displays the entered SQL command, and the command is dynamically preprocessed and executed. When the program user enters a slash (/) in response to the prompt, the transaction is committed and the program terminates. Note what happens when a SELECT command is entered.

As illustrated in Figure 9-7, the main program $\boxed{1}$ first performs a function named *ConnectDBE* $\boxed{2}$ to start a DBE session. The CONNECT command starts the session in the DBEnvironment named *PartsDBE*.

The program then performs subroutine *BeginWork* $\boxed{3}$ to start a transaction with the BEGIN WORK command. Once a transaction has been started, function *PrepareExecute* $\boxed{7}$ is performed until *Check* evaluates to FALSE.

PrepareExecute first declares a dynamic host variable *DynamicCommand*, which will hold the dynamic command to be entered by the user.

Then the user is prompted for the non-query command $\boxed{7A}$ to be dynamically prepared and executed. The entered command is then prepared $\boxed{7B}$, and if the command preparation is successful, it is executed $\boxed{7C}$. If the command was successfully executed, the user is re-prompted for another non-query command. The function terminates when *PrepareExecute* is set to FALSE by the user entering a slash (/) in response to the command prompt $\boxed{7A}$.

When *PrepareExecute* evaluates to FALSE, subroutine *CommitWork* $\boxed{4}$ is performed. This subroutine executes a COMMIT WORK command. Then subroutine *ReleaseDBE* $\boxed{5}$ executes a ROLLBACK WORK RELEASE command to terminate the DBE session. After *ReleaseDBE* has executed, the program terminates. Explicit status checking is used throughout this program. When ALLBASE/SQL returns a non-zero value in SQLCode following the execution of each embedded SQL command, subroutine *SQLStatusCheck* $\boxed{8}$ is performed. This subroutine writes out messages based on the values of SQLCode and SQLWARN, then calls *SQLExplain* $\boxed{6}$ to display one or more messages. *SQLExplain* executes the SQLEXPLAIN command and prints out the error messages. If an error is very serious (SQLCode < -14024), a flag named *Abort* is set, and subroutines *CommitWork* and *ReleaseDBE* are performed before the program is terminated.

```
 Program to illustrate the PREPARE and EXECUTE commands -- forex9b
 Event List:
   CONNECT TO PartsDBE
   BEGIN WORK
   Prompt for SQL command
   PREPARE SQL command
   EXECUTE SQL command
   Repeat the above 3 steps until the user enters a /
   COMMIT WORK
   RELEASE from DBEnvironment

CONNECT TO PartsDBE
Successful CONNECT
BEGIN WORK
Successful BEGIN

Enter an SQL non-query command or a / to stop:
>UPDATE STATISTICS FOR TABLE PurchDB.Parts;
Dynamic command to PREPARE is: UPDATE STATISTICS FOR TABLE PurchDB.Parts

PREPARE successful.
EXECUTE the command.
EXECUTE successful.

Enter an SQL non-query command or a / to stop:
>SELECT * FROM PurchDB.Parts;
Dynamic command to PREPARE is: SELECT * FROM PurchDB.Parts;

PREPARE successful.
EXECUTE the command.
HPSQL error!
Call SQLExplain

Module TEMP.FOREX9B(1) is not a procedure.  (DBERR 2752)
 Enter an SQL non-query command or a / to stop:

> /
No more commands.
COMMIT WORK
Successful COMMIT
RELEASE DBEnvironment
Successful RELEASE
```

Figure 9-6. Runtime Dialog of Program forex9b

```
C    * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C    * This program illustrates the use of SQL dynamic non-query *
C    * commands executed from a FORTRAN program.                 *
C    * This program demonstrates the use of the PREPARE and      *
C    * EXECUTE commands.                                         *
C    * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

     IMPLICIT NONE
     LOGICAL*2  PrepareExecute, Check, ConnectDBE

C       (**** Begin SQL Communication Area ****)
     EXEC SQL INCLUDE SQLCA

C       (**** Begin Host Variable Declarations ****)
        EXEC SQL BEGIN DECLARE SECTION
        EXEC SQL END DECLARE SECTION

C     (**** Beginning of Main Program ****)    [1]

        WRITE (*,*) CHAR(27), 'U'
        WRITE (*,*) 'Program to illustrate the PREPARE and EXECUTE
       1command1s -- forex9b'
        WRITE (*,*) ' '
        WRITE (*,*) 'Event List:'
        WRITE (*,*) '  CONNECT TO PartsDBE'
        WRITE (*,*) '  BEGIN WORK'
        WRITE (*,*) '  Prompt for SQL command:'
        WRITE (*,*) '  PREPARE SQL command'
        WRITE (*,*) '  EXECUTE SQL command'
        WRITE (*,*) '  Repeat the above 3 steps until the user
       1enters a /'
        WRITE (*,*) '  COMMIT WORK'
        WRITE (*,*) '  RELEASE from DBEnvironment'
        WRITE (*,*) ' '

        IF (ConnectDBE()) THEN
          CALL BeginWork
          Check = .TRUE.
          DO WHILE (Check)
            Check = PrepareExecute()
          END DO
          CALL CommitWork
          CALL ReleaseDBE
        ENDIF
        STOP
        END
```

Figure 9-7. Program forex9b: Sample Program Using PREPARE and EXECUTE

```
C       (* Beginning of the Subroutines *)

      LOGICAL*2 FUNCTION ConnectDBE()                          2
C       (**** Subroutine to CONNECT TO PartsDBE ****)

        INTEGER*2         OK
        PARAMETER (OK = 0)

C       (* Begin Communication Area *)
        EXEC SQL INCLUDE SQLCA

C       (**** Begin Host Variable Declarations ****)
        EXEC SQL BEGIN DECLARE SECTION
        EXEC SQL END DECLARE SECTION

        WRITE (*,*) 'CONNECT TO PartsDBE'
        EXEC SQL CONNECT TO 'PartsDBE'
        ConnectDBE = .TRUE.
        IF (SQLCode .NE. OK) THEN
          ConnectDBE = .FALSE.
        CALL SQLStatusCheck
        ELSE
          WRITE (*,*) 'Successful CONNECT'
        ENDIF
        RETURN
        END

        SUBROUTINE BeginWork                                   3
C       (**** Subroutine to Begin a Transaction ****)

        INTEGER*2         OK
        PARAMETER (OK = 0)

C       (* Begin Communication Area *)
        EXEC SQL INCLUDE SQLCA

C       (**** Begin Host Variable Declarations ****)
        EXEC SQL BEGIN DECLARE SECTION
        EXEC SQL END DECLARE SECTION
```

Figure 9-7. Program forex9b: Sample Program Using PREPARE and EXECUTE (page 2 of 6)

```
         WRITE (*,*) 'BEGIN WORK'
         EXEC SQL BEGIN WORK
         IF (SQLCode .NE. OK) THEN
           CALL SQLStatusCheck
           CALL ReleaseDBE
         ELSE
           WRITE (*,*) 'Successful BEGIN'
         ENDIF
         RETURN
         END
         SUBROUTINE CommitWork                           4
C        (**** Subroutine to COMMIT WORK ****)

         INTEGER*2         OK
         PARAMETER (OK = 0)

C        (* Begin Communication Area *)
         EXEC SQL INCLUDE SQLCA

C        (**** Begin Host Variable Declarations ****)
         EXEC SQL BEGIN DECLARE SECTION
         EXEC SQL END DECLARE SECTION

         WRITE (*,*) 'COMMIT WORK'
         EXEC SQL COMMIT WORK
         IF (SQLCode .NE. OK) THEN
           CALL SQLStatusCheck
           CALL ReleaseDBE
         ELSE
           WRITE (*,*) 'Successful COMMIT'
         ENDIF
         RETURN
         END

         SUBROUTINE ReleaseDBE                           5
C        (**** Subroutine to RELEASE PartsDBE ****)

         INTEGER*2         OK
         PARAMETER (OK = 0)

C        (* Begin Communication Area *)
         EXEC SQL INCLUDE SQLCA

C        (**** Begin Host Variable Declarations ****)
         EXEC SQL BEGIN DECLARE SECTION
         EXEC SQL END DECLARE SECTION
```

Figure 9-7. Program forex9b: Sample Program Using PREPARE and EXECUTE (page 3 of 6)

```
      WRITE (*,*) 'RELEASE DBEnvironment'
      EXEC SQL ROLLBACK WORK RELEASE
      IF (SQLCode .NE. OK) THEN
        CALL SQLStatusCheck
        CALL ReleaseDBE
      ELSE
        WRITE (*,*) 'Successful RELEASE'
      ENDIF
      END

      SUBROUTINE SQLExplain                                    6
C     (**** Subroutine to CALL SQLEXPLAIN ****)

C     (* Begin Communication Area *)
      EXEC SQL INCLUDE SQLCA

C     (**** Begin Host Variable Declarations ****)
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*80     SQLMessage
      EXEC SQL END DECLARE SECTION

      WRITE (*,*) 'Call SQLExplain'
      EXEC SQL SQLEXPLAIN :SQLMessage
      WRITE (*,*) ' '
      WRITE (*,100) SQLMessage
100   FORMAT(A80)
      RETURN
      END

      LOGICAL*2 FUNCTION PrepareExecute()                      7
C     (**** Function to PREPARE and EXECUTE the ****)
C     (**** user-entered command. ****)

      CHARACTER*80     CMD1
      INTEGER*2        OK
      PARAMETER (OK = 0)

C     (**** Begin SQL Communication Area ****)
      EXEC SQL INCLUDE SQLCA

C     (**** Begin Host Variable Declarations ****)
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*80     DynamicCommand
      EXEC SQL END DECLARE SECTION
```

Figure 9-7. Program forex9b: Sample Program Using PREPARE and EXECUTE (page 4 of 6)

```
       WRITE (*,100)                                                7A
 100   FORMAT(/'Enter an SQL non-query command or / to STOP '
      1 ,//$,' > ')
       READ (*,110) DynamicCommand
 110   FORMAT(A80)
       IF (DynamicCommand .EQ. '/') THEN
         WRITE (*,*) 'No more commands.'
         PrepareExecute = .FALSE.
       ELSE
         WRITE (*, 120) DynamicCommand
 120     FORMAT (/'The dynamic command to PREPARE is: '//, A80)
         EXEC SQL PREPARE CMD1 FROM :DynamicCommand                 7B
         IF (SQLCode .NE. OK) THEN
           CALL SQLStatusCheck
           WRITE (*,*) 'PREPARE failed.'
         ELSE
           WRITE (*,*) 'PREPARE successful.'
           WRITE (*,*) 'EXECUTE the command.'
           EXEC SQL EXECUTE CMD1                                    7C
             IF (SQLCode .NE. OK) THEN
               CALL SQLStatusCheck
             ELSE
               WRITE (*,*) 'EXECUTE successful.'
             ENDIF
         ENDIF
       PrepareExecute = .TRUE.
       ENDIF
       RETURN
       END

       SUBROUTINE SQLStatusCheck                                    8
 C     (**** Subroutine SQLStatusCheck checks status of SQL
      1commands ****)
 C     (**** and print HPSQL error messages. ****)

       LOGICAL*2  Abort, Check
       INTEGER    MultipleRows, DeadLock, NotFound
       PARAMETER (MultipleRows = -10002,
      1           DeadLock =     -14024,
      2           NotFound =        100)

 C     (**** Begin Communication Area ****)
       EXEC SQL INCLUDE SQLCA
```

Figure 9-7. Program forex9b: Sample Program Using PREPARE and EXECUTE (page 5 of 6)

```
C      (**** Begin Host Variable Declarations ****)
       EXEC SQL BEGIN DECLARE SECTION
       CHARACTER*80     SQLMessage
       EXEC SQL END DECLARE SECTION

       Abort = .FALSE.
       Check = .TRUE.

       IF (SQLWarn(0) .EQ. 'W') THEN
         WRITE (*,*) 'HPSQL Warning!'
       ELSEIF (SQLCode .EQ. NotFound) THEN
         WRITE (*,*) 'No record found for this PartNumber!'
       ELSEIF (SQLCode .EQ. MultipleRows) THEN
         WRITE (*,*) 'Multiple records exit for this PartNumber!'
       ELSEIF (SQLCode .EQ. DeadLock) THEN
         Abort = .TRUE.
         WRITE (*,*) 'A DEADLOCK has occurred!'
       ELSEIF (SQLCode .LT. DeadLock)THEN
         Abort = .TRUE.
         WRITE (*,*) 'Serious ALLBASE/SQL error!'
       ELSEIF (SQLCode .LT. 0) THEN
         WRITE (*,*) 'ALLBASE/SQL error!'
       ENDIF
       DO WHILE (Check)
         CALL SQLExplain

         IF (SQLCode .EQ. 0) THEN
           Check = .FALSE.
         ENDIF
       END DO

       IF (Abort) THEN
         CALL CommitWork
         CALL ReleaseDBE
       ENDIF

       RETURN
         END
```

Figure 9-7. Program forex9b: Sample Program Using PREPARE and EXECUTE (page 6 of 6)

10

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 contraints 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 greated 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 MPE/iX*.

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 10-1. Commands Used with Integrity Constraints

DDL Operations	DCL Operations	DML Operations
CREATE TABLE	GRANT REFERENCES	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 contraints 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 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 10-2. Constraint Test Matrix

DML Operations	UNIQUE or PRIMARY KEY	Referenced Table	Referencing Table
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 10-1 illustrates these contraint relationships by showing the name of each constraint and its referencing or referenced columns. Referencing columns are shaded. Referenced columns are clear white.

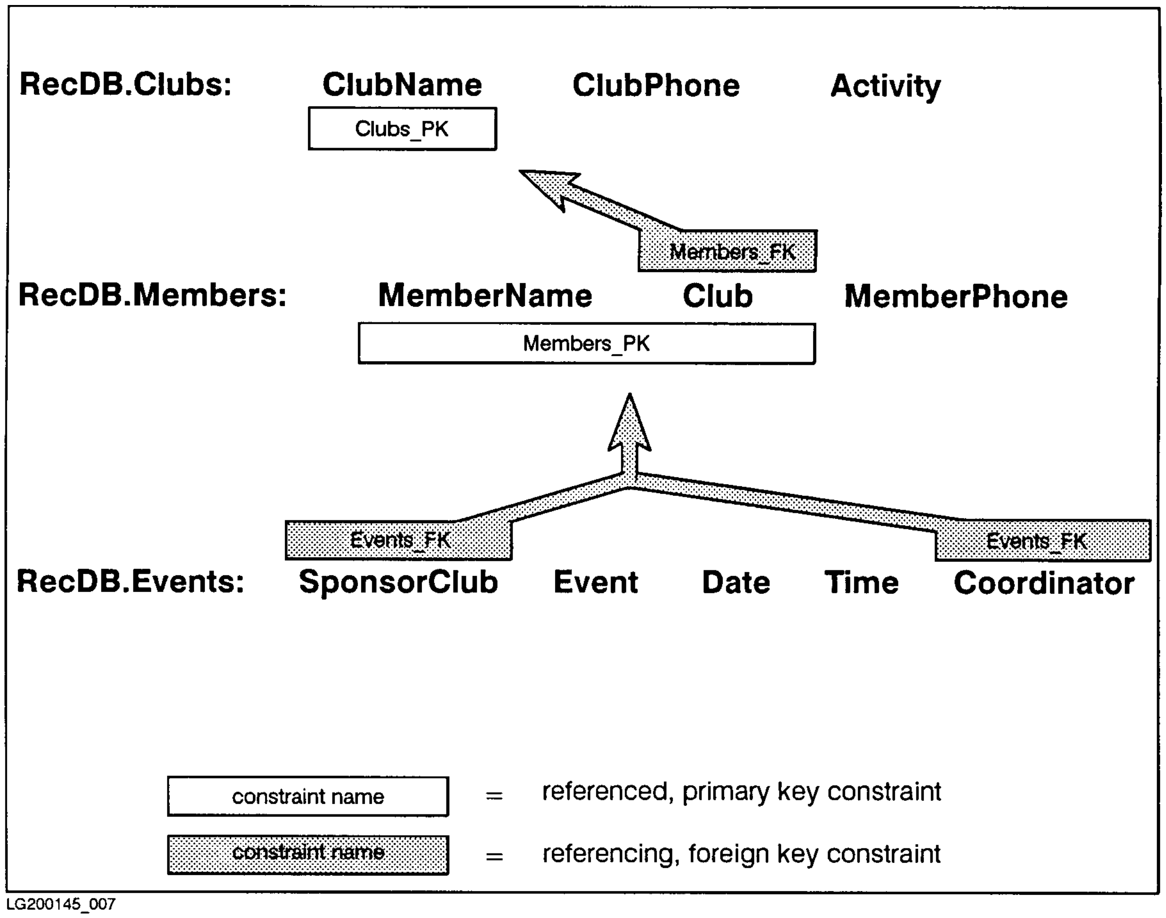


Figure 10-1. Constraints Enforced on the Recreation Database

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.")

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.

11

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 INSERT.
- Changing a LONG Column with UPDATE [WHERE CURRENT].
- Retrieving LONG Column Data with 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 11-1. Commands You Can Use with LONG Columns

DDL Operations	DML Operations
ALTER TABLE	INSERT
CREATE TABLE	UPDATE [WHERE CURRENT]
	SELECT
	FETCH
	REFETCH
	DELETE [WHERE CURRENT]

General Concepts

ALLBASE/SQL stores LONG column data in a database for later retrieval. LONG column data is not 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.

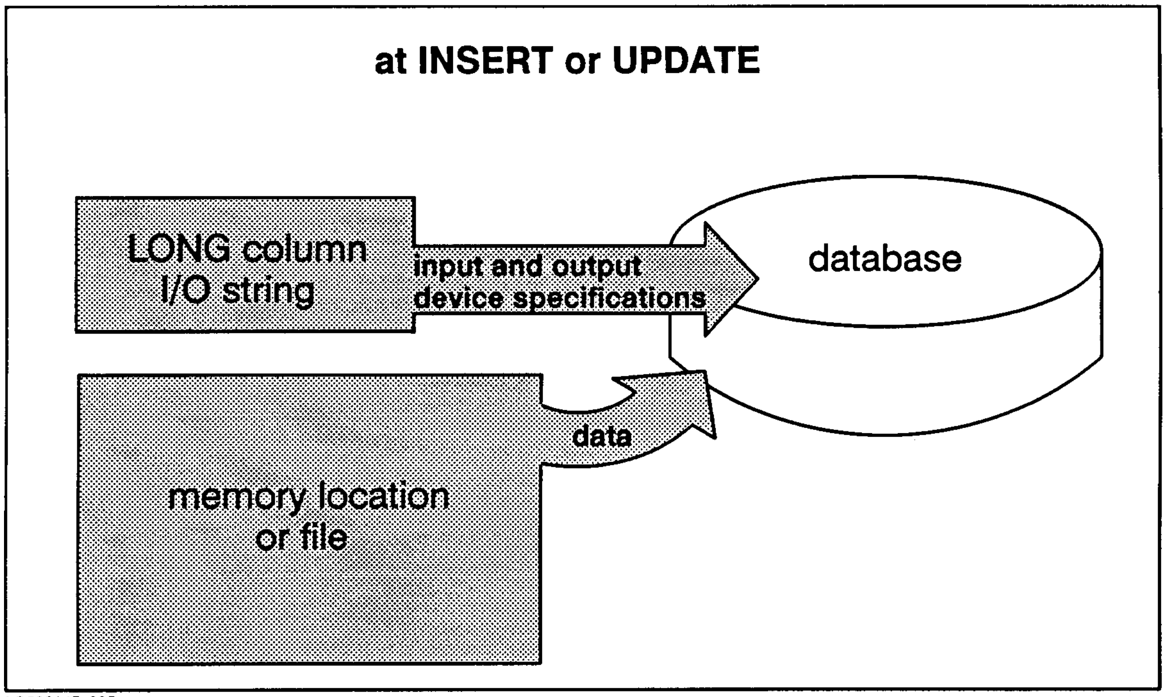
LONG column data can occupy a practically unlimited amount of space in the database, the maximum number of bytes being $2^{31}-1$ (or 2,147,483,647) per LONG column per row. Standard column data is restricted to 3996 bytes maximum.

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.

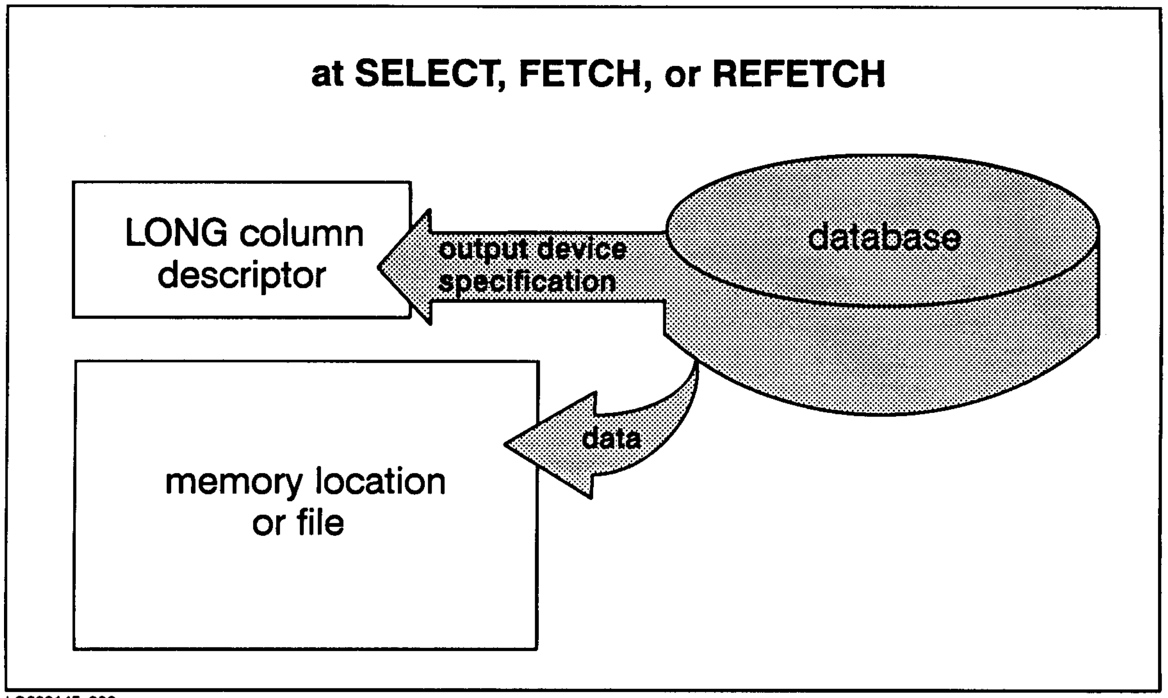
During an INSERT or UPDATE operation, you specify a **LONG column I/O string** to indicate where LONG column input data is located and where that data is to be placed when it is later selected or fetched. You indicate either an operating system file or random random heap space.

A **LONG column descriptor** (rather than the data itself) is selected or fetched into a host variable. Figure 11-1 and Figure 11-2 illustrate these concepts.



LG200145_005a

Figure 11-1. Flow of LONG Column Data and Related Information to the Database



LG200145_006a

Figure 11-2. 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 new 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.

$$(\mathit{ColumnName}\ \texttt{LONG} \left\{ \begin{array}{l} \texttt{BINARY} \\ \texttt{VARBINARY} \end{array} \right\} (\mathit{ByteSize})\ [\texttt{IN}\ \mathit{DBEFileSet}]\ \ [\texttt{NOT NULL}])\ [\texttt{,} \ldots\ ]$$

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 you 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 random heap address.

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 direct output to a random random heap address. 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 an INSERT Command

As with any column, use the INSERT command to initially put data into a LONG column. At the time of the insert, all input devices must be on the system in the locations you have specified. Should your insert operation fail, nothing is inserted, a relevant error message is returned to the program, and the transaction continues. Depending on your application, you might want to write a verification routine that reads a portion of each specified input device to make certain valid data exists prior to using the INSERT command.

The next examples are based on the PartsTable created and altered in the previous section, "Defining LONG Columns with CREATE TABLE or ALTER TABLE." Additional examples of LONG column I/O string usage are found in the *ALLBASE/SQL Reference Manual* .

Insert Using Host Variables for LONG Column I/O Strings

When inserting a single row, use a version of the LONG Column I/O String for each LONG column following the VALUES clause, as below.

```
INSERT INTO PartsTable VALUES (
            'bracket',
            200,
            :PartPictureIO,
            :PartModuleIO)
```

An example of the values that might be stored in the host variables, :PartPictureIO and :Part ModuleIO, are shown in the last two fields of a hypothetical record in the section, *Example Data File*, which 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.

```
bracket    200 0'<bracket.tools >bracket'     0'<mod88.module > mod88'       0
```

Example Data File

If your program reads the data from a data file, the file might look something like this. Note that it is limited to 80 characters per record to facilitate documentation.

```
hammer     011 0'<hammer.tools >hammer'       0'<mod11.module > mod11'       0
file       022 0'<file.tools >file'           0'<mod22.module > mod22'       0
saw        033 0'<saw.tools > saw'            0'<mod33.module > mod33'       0
wrench     044 0'<wrench.tools >wrench'       0'<mod44.module > mod44'       0
lathe      055 0'<lathe.tools >lathe'         0'<mod55.module > mod55'       0
drill      066 0'<drill.tools >drill'         0'<mod66.module > mod66'       0
pliers     077 0'<pliers.tools >pliers'       0'<mod77.module > mod77'       0
            .
            .
            .
```

Retrieving LONG Column Data with a SELECT, FETCH, or REFETCH Command

The following syntax represents the available subset when your select list includes one or more LONG columns. Remember, a LONG column can be referenced only in a select list and/or a host variable declaration.

```
                ⎧ *                              ⎫
                ⎪ [Owner.]Table.*                ⎪
SELECT [ALL]    ⎨ CorrelationName.*              ⎬ [, ... ]
                ⎩ CorrelationName.ColumnName     ⎭
[INTO HostVariableDeclaration]  FROM { [Owner.]FromTableName [CorrelationName] }
[, ... ]
```

As 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 random heap space.

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 random heap space.

Note The LONG column descriptor must be declared whether or not you access its contents in your code.

Table 11-2. LONG Column Descriptor

Description	Possible Binary Values	Byte Range
Name or Address of Output Device	File name or random heap address	1 through 44
Output Device Options	0 = no output specified 1 = overwrite 2 = append 3 = wildcard 4 = overwrite and wildcard 5 = append and wildcard	45
Output Device Type	0 = no device specified 1 = file 3 = random heap space	46
Input Device Type	0 = no device specified 1 = file 3 = random heap space	47
Reserved for Internal Use		48
Size in Bytes of LONG Column Data	1 to $2^{31}-1$ (or 2,147,483,647) per LONG column per row. Standard column data is restricted to 3996 bytes maximum.	49 through 52
Reserved for Internal Use		53 through 96

Example LONG Column Descriptor Declaration

```
C     Use this when you don't need to break down the descriptor.

      CHARACTER*96 LongDesc

C     Use this when you want to access a portion of the descriptor.

      CHARACTER*44 OutputDevName
      CHARACTER    OutDevOption
      CHARACTER    OutDevType
      CHARACTER    InDevType
      CHARACTER    UnusedA
      INTEGER      ByteLength
      CHARACTER*44 UnusedB
```

Using LONG Columns with a SELECT Command

Should an error occur before completion of the SELECT command, any operating system files written before the error occurred remain on the system, and LONG column descriptors written to a host variable array remain. It is your responsibility to remove such files as appropriate.

Using LONG Columns with a Dynamic FETCH Command

If you have the need to dynamically retrieve LONG column data, the sqlrowbuf column of the sqlda, as always, contains the address of the data buffer. However, the data buffer, rather than containing LONG column data, holds the 96-byte LONG column descriptor.

The sqltype field of the format array holds a data type ID number of 15 for a LONG BINARY column and 16 for a LONG VARBINARY column. And the sqltotallen and sqlvallen columns will always contain a value of 96 (indicating the length of the descriptor).

When a NULL is fetched as the LONG column value, no external files are created, and the associated indicator variable for the LONG column descriptor is set to −1.

Changing a LONG Column with an UPDATE [WHERE CURRENT] Command

When you issue an UPDATE command on a LONG column, you have the following options:

- Change the stored data as well as the output device name and/or options.
- Change the stored data only.
- Change the output device name and/or options only.

Specify a LONG column I/O string (discussed earlier in this chapter) following the SET clause, for each LONG column to be updated. You must specify either the input device, the output device, or both. Complete syntax with examples is found in the *ALLBASE/SQL Reference Manual* .

In the following example, the LONG column I/O string is contained in host variable PartPictureIO.

```
UPDATE PartsTable
   SET PartPicture = :PartPictureIO
 WHERE PartName = 'saw'
```

Removing LONG Column Data with a DELETE [WHERE CURRENT] Command

Syntax for the DELETE and DELETE WHERE CURRENT commands is unchanged for use with LONG columns. It is limited for the DELETE command in that a LONG column cannot be used in the WHERE clause.

In the following example, any rows in PartsTable with the PartName of hammer are deleted.

```
   DELETE FROM PartsTable WHERE PartName = 'hammer'
```

When LONG column data is deleted, the space it occupied in the DBEnvironment is released when your transaction ends. But any data file selected earlier still exists on the operating system. You may want to design a "cleanup" strategy for such files that are no longer needed.

Coding Considerations

File versus Random Heap Space

Depending on your application, you might want to use a file or random heap space as your input or output device. Random heap space may provide faster data access. Consider how much heap 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 group, 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 random heap space available for your use in relation to the size and number of LONG columns to be selected or fetched.

12

Programming with ALLBASE/SQL Functions

This chapter highlights functions available in ALLBASE/SQL. The functions return values that can be used to access, search, update, and delete data. Refer to the "Expressions" chapter of the *ALLBASE/SQL Reference Manual* for a discussion of other available ALLBASE/SQL functions. The ALLBASE/SQL functions discussed in this chapter are as follows:

- Date/Time functions.
- Tuple Identifier (TID) function.

Programming with Date/Time Functions

Seven functions can be used with date/time data types. These functions provide flexibility for inputting and retrieving date/time data from the database.

These functions can be used with a preprocessed application or with ISQL. This chapter outlines basic principles for using date/time functions in an application program. The following sections are included:

- Where Date/Time Functions Can Be Used.
- Defining and Using Host Variables with Date/Time Functions.
- Using Date/Time Input Functions.
- Using Date/Time Output Functions.
- Using the Date/Time ADD_MONTHS Function.
- Coding Considerations.
- Program Examples for Date/Time Data.

Date/time functions are used as you would use an expression. And when used in a select list, all date/time functions produce data output. Refer to the section in this chapter, "Defining and Using Host Variables with Date/Time Functions."

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 Native-3000 format.

Where Date/Time Functions Can Be Used

Use date/time functions, as you would an expression, in the DML operations listed below:

Table 12-1. Where to Use Date/Time Functions

DML Operation	Clause

INSERT [1]	VALUES
	WHERE

UPDATE or
UPDATE WHERE CURRENT
	SET
	WHERE

DELETE or DELETE WHERE CURRENT	WHERE
SELECT	Select list [2] WHERE
DECLARE	Select list [2] WHERE

[1] In the case of a 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.

[2] Input functions, TO_DATE, TO_TIME, TO_DATETIME, and TO_INTERVAL, are generally not appropriate in a select list.

Defining and Using Host Variables with Date/Time Functions

Date/time functions can be used in the way an expression is used; that is, in a select list to indicate the columns you want in the query result, in a search condition to define the set of rows to be operated on, and to define the value of a column when using the UPDATE command. (See the *ALLBASE/SQL Reference Manual* for in-depth information regarding expressions.)

Whether you use host variables or literal strings to specify the parameters of the date/time functions depends on the elements of your application and on how you are using the functions. This section focuses on the use of host variables.

You can use host variables to specify input or output format specifications. Use them as well to hold data input to and any resulting data output from the date/time functions. (Host variables cannot be used to indicate column names.)

Host variables for format specifications must be defined in your application to be compatible with ALLBASE/SQL CHAR or VARCHAR data types. The exception is the ADD_MONTHS function which requires an INTEGER compatible host variable.

As for host variables containing input and output data, define them to be CHAR or VARCHAR compatible with one exception. The TO_INTEGER function requires an INTEGER compatible host variable for its output.

Reference the chapter on defining host variables for additional information 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 12-2 shows host variable data type compatibility for date/time functions.

Table 12-2. Host Variable Data Type Compatibility for Date/Time Functions

Date/Time Function	**Input Format Specification**	**Output Format Specification**	**Input Data**	**Output Data**
TO_DATE TO_TIME TO_DATETIME TO_INTERVAL	(VAR)CHAR		(VAR)CHAR	(VAR)CHAR [1]
TO_CHAR		(VAR)CHAR		(VAR)CHAR
TO_INTEGER		(VAR)CHAR		INTEGER
ADD_MONTHS	INTEGER			(VAR)CHAR [1]

[1] Applies only when used in a select list.

Using Date/Time Input Functions

The 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:

$$\left\{\begin{array}{l}\texttt{TO\_DATETIME (}\mathit{DataValue}\ \left[\,,\mathit{FormatSpecification}\right]\texttt{)}\\ \texttt{TO\_DATE (}\mathit{DataValue}\ \left[\,,\mathit{FormatSpecification}\right]\texttt{)}\\ \texttt{TO\_TIME (}\mathit{DataValue}\ \left[\,,\mathit{FormatSpecification}\right]\texttt{)}\\ \texttt{TO\_INTERVAL (}\mathit{DataValue}\ \left[\,,\mathit{FormatSpecification}\right]\texttt{)}\end{array}\right\}$$

Input functions can be used in DML operations as shown in Table 12-1. 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 12-3.

Table 12-3. Sample of User Requested Formats for Date/Time Data

Date/Time Data Type	Desired Format Specification	Length of Format Specification in ASCII Characters
DATETIME	MM-DD-YYYY HH:MM:SS.FFF	23
DATE	MM-DD-YYYY	10
TIME	HH:MM:SS [1]	8
INTERVAL	DDDDDDD HH:MM:SS	16

[1] This is the default time data format.

You might use the following generic code examples to meet their needs.

Example Using the INSERT Command.

Your application allows users to enter data in their desired formats with a minimum of effort on your part.

```
BEGIN DECLARE SECTION
```

Declare input host variables (:BatchStamp, :BatchStamp-Format, :TestDate, :TestDate-Format, :TestStart, :LabTime, and LabTime-Format) to be compatible with data type CHAR or VARCHAR.

Declare input indicator variables (:TestDateInd 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 12-2 with one exception. In the case of a INSERT command, output functions are limited to use in the select list and the WHERE clause of a Type 2 INSERT command.

As with date/time input functions, use a host variable or a literal string to indicate a format specification. See the *ALLBASE/SQL Reference Manual* for detailed syntax.

Following is the general syntax for date/time output functions:

$$\left\{\begin{array}{l}\texttt{TO\_CHAR (}\mathit{ColumnName}\ [\texttt{,}\mathit{FormatSpecification}]\texttt{)}\\ \texttt{TO\_INTEGER (}\mathit{ColumnName}\texttt{, }\mathit{FormatSpecification}\texttt{)}\end{array}\right\}$$

Example TO_CHAR Function

The default format for the DATETIME data type specifies the year followed by the month followed by the day. The default format for the TIME data type specifies a 24-hour clock. (Refer to the *ALLBASE/SQL Reference Manual* .)

Suppose users located in Italy want to input a specified batch stamp to obtain the start and end times of the related test in 12-hour format. They will key the batch stamp in this format, "DD-MM-YYYY HH12:MM:SS:FFF AM or PM." The times returned will be in this format, "HH12:MM:SS.FFF AM or PM."

Data is located in the TestData table in the manufacturing database. (Refer to appendix C in the *ALLBASE/SQL Reference Manual* .) The following code could be used:

```
BEGIN DECLARE SECTION

Declare input host variables (:TwelveHourClockFormat, :BatchStamp,
:ItalianFormat, and :SpecifiedInput) to be compatible with data type
CHAR or VARCHAR.

Declare output host variables (:TestStart and :TestEnd) to be compatible
with data type CHAR or VARCHAR .

Declare output indicator variables (:TestStartInd and :TestEndInd).

END DECLARE SECTION
.
.
.
SELECT TO_CHAR(TestStart, :TwelveHourClock),
       TO_CHAR(TestEnd, :TwelveHourClock)
  INTO :TestStart :TestStartInd,
       :TestEnd :TestEndInd,
  FROM ManufDB.TestData
```

```
    WHERE TO_DATETIME(:BatchStamp, :ItalianFormat) = :SpecifiedInput
```

Note the use of indicator variables in the above example. Because the TO_CHAR function is used in the select list, no need exists to specify an indicator variable as part of the function.

Example TO_INTEGER Function

The TO_INTEGER format specification is mandatory and differs from that of other date/time functions in that it must consist of a single element only. See the *ALLBASE/SQL Reference Manual* for detailed format specifications.

Perhaps you are writing a management report that indicates the quarter of the year in which tests were performed. (As in the previous example, data is located in the TestData table in the manufacturing database.) You could use the following code:

```
BEGIN DECLARE SECTION

Use the ALLBASE/SQL Reference Manual  to determine your desired format
specification.  (In this case it is Q.)

Declare the input host variable, :QuarterlyFormat, to be compatible with data
types CHAR or VARCHAR.

Declare an output host variable (:TestDateQuarter)
to be compatible with data type INTEGER.  Declare other output host
variables (:BatchStamp, :LabTime, :PassQty, and :TestQty) to be
compatible with data type CHAR or VARCHAR.

Remember to declare output indicator variables (:TestDateQuarterInd,
LabTimeInd, PassQtyInd, and :TestQtyInd).

END DECLARE SECTION
.
.
.
DECLARE ReportInfo CURSOR FOR
                       SELECT BatchStamp,
                              TO_INTEGER(TestDate, :QuarterlyFormat),
                              LabTime,
                              PassQty,
                              TestQty
                         FROM ManufDB.TestData
.
.
.
     FETCH ReportInfo
      INTO ReportBuffer :BatchStamp
                        :TestDateQuarter   :TestDateQuarterInd
                        :LabTime           :LabTimeInd
                        :PassQty           :PassQtyInd
                        :TestQty           :TestQtyInd
```

Using the Date/Time ADD_MONTHS Function

This function allows you to add an integer number of months to a DATE or DATETIME column. Do so by indicating the number of months as a positive, negative, or unsigned integer value. (An unsigned value is assumed positive.) Also, you can specify the integer in a host variable of type INTEGER.

The ADD_MONTHS function can be used in both input and output operations as shown in Table 12-1.

Following is the general syntax for the ADD_MONTHS function:

{ `ADD_MONTHS (`*ColumnName*`,` *IntegerValue*`)` }

As with date/time output functions, use the ADD_MONTHS function with any DML operation listed in Table 12-2 with one exception. In the case of a 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.

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 12-4. 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 or 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 "Command Syntax" chapter in this document under the "Description" section 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.

Index