900 Series HP 3000 Computer Systems

ALLBASE/SQL Advanced Application Programming Guide

HP Part No. 36216-90100
Printed in U.S.A. 1994

First Edition
E0494

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	April 1994	36216-02A.G0.00

ALLBASE/SQL Manuals

Title	Customer Order Number
ALLBASE/NET User's Guide	36216-90031
ALLBASE/SQL Advanced Application Programming Guide	36216-90100
ALLBASE/SQL C Application Programming Guide	36216-90023
ALLBASE/SQL COBOL Application Programming Guide	36216-90006
ALLBASE/SQL Database Administration Guide	36216-90005
ALLBASE/SQL FORTRAN Application Programming Guide	36216-90030
ALLBASE/SQL Message Manual	36216-90009
ALLBASE Pascal Application Programming Guide	36216-90007
ALLBASE/SQL Performance and Monitoring Guidelines	36216-90102
ALLBASE/SQL Reference Manual	36216-90001
HP ALLBASE/QUERY User's Guide	92534-90011
HP PC API User's Guide for ALLBASE/SQL and IMAGE/SQL	36216-90104
ISQL Reference Manual for ALLBASE/SQL and IMAGE/SQL	36216-90096
Up and Running with ALLBASE/SQL	36389-90011

Preface

This ALLBASE/SQL Advanced Application Programming Guide for MPE/iX is the first edition of a new manual targeted for experienced ALLBASE/SQL application programmers using ALLBASE/SQL on HP 3000 systems. It describes product enhancements for ALLBASE/SQL Release F through the present release. This guide replaces the release specific ALLBASE/SQL application programming bulletins.

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 guide does not replace the four ALLBASE/SQL application programming guides written for C, COBOL, FORTRAN, and Pascal programmers. Indeed, if you are a new user of ALLBASE/SQL who will be assuming programmer's responsibilities, you should put this book down immediately and pick up the application programming guide for the language you use. Until you have read and absorbed your guide, much of the information included in this advanced guide will be of little use to you. This manual is intended as a source of concepts and examples for programmers using the newer and often more complex ALLBASE/SQL coding techniques.

Notable features of this new document relative to the existing ALLBASE/SQL manual set include the following:

- It is language generic. This guide provides information for ALLBASE/SQL programmers as a group; whereas, each of the language specific application programming guides is directed to programmers of a given language, C, COBOL, FORTRAN, or Pascal. This advanced guide contains code segments and programs in C, COBOL, FORTRAN, and Pascal. Some examples are duplicated in more than one language. Other examples are presented in only one language. Language generic examples are also included.
- This advanced guide contains examples based on the sample database environment, often including message numbers that might be returned to your application. It frequently refers you to the lanuguage specific application programming guides and other manuals in the ALLBASE/SQL manual set for additional information.
- Both additions and changes to functionality are presented. In the case of changes, some information regarding previous releases can be invalidated. This is specifically referenced in the advanced guide.

To facilitate information lookup, one chapter is devoted to each type of functionality. The index is task as well as reference oriented. The following topics are included in this manual:

- Chapter 1, "Using the Preprocessor," presents complete syntax for Full Preprocessing Mode, Static Conversion Mode, Syntax Checking Mode, and POSIX invocation for the C preprocessor.
- Chapter 2, "Flagging Non-Standard SQL with the FIPS Flagger," discusses ALLBASE/SQL flagging for the FIPS 127.1 standard.
- Chapter 3, "Comparing Static and Dynamic SQL," contrasts static and dynamic SQL statements and applications.
- Chapter 4, "Using Parameter Substitution in Dynamic Statements," introduces and gives examples for using dynamic parameters.
- Chapter 5, "Using Procedures in Application Programs," presents application specific features for coding with procedures.
- Chapter 6, "Using Data Integrity Features," compares the use of statement level integrity (the default) and row level integrity and discusses how to defer constraint error checking, how to use check constraints in tables and views, and how to use features of the ALTER TABLE statement.
- Chapter 7, "Transaction Management with Multiple DBEnvironment Connections," describes functionality that allows multiple, simultaneous connections to one or more DBEnvironments and use of the SET TIMEOUT statement.
- Chapter 8, "COBOL Preprocessor Enhancements," outlines two enhancements for the COBOL preprocessor: record descriptions for non-bulk queries and host variable initialization with the VALUE clause.
- Chapter 9, "Programming with Indicator Variables in Expressions," discusses the use of input indicator variables.
- Chapter 10, "Analyzing Queries with GENPLAN," describes how to use the ISQL GENPLAN statement.
- Chapter 11, "Using the VALIDATE Statement," introduces the VALIDATE statement for validating sections prior to runtime.
- Chapter 12, "Corrections to the BCDToString Example Program Routine," provides replacement pages for the BCDToString routine found in the *ALLBASE/SQL C Application Programming Guide* and in the *ALLBASE/SQL Pascal Application Programming Guide*.

Example code is based, for the most part, on the sample database environment, PartsDBE, which is a part of the ALLBASE/SQL product. (Refer to appendix C in the *ALLBASE/SQL Reference Manual* for information about the structure of PartsDBE and for listings of the sample database.)

We hope you enjoy using the document and that you will send your comments and suggestions to our attention so that the *ALLBASE/SQL Advanced Application Programming Guide* can become even more effective.

What's New in this Release

The following table highlights the new or changed functionality in this release, and shows you where each feature is documented.

New Features in ALLBASE/SQL Release G.0

Feature (Category)	Description	Documented in ...
Stored procedures (Usability)	Provides additional stored procedure functionality for application programs. Allows declaration of a procedure cursor and fetching of multiple rows within a procedure to applications. New statement: ADVANCE. Changed syntax: CLOSE, CREATE PROCEDURE, DECLARE CURSOR, DESCRIBE, EXECUTE, EXECUTE PROCEDURE, FETCH, OPEN.	*ALLBASE/SQL Reference Manual*, "SQL Statements" and "Using Procedures" in "Constraints, Procedures and Rules;" *ALLBASE/SQL Advanced Application Programming Guide*, "Using Procedures in Application Programs."
Case insensitivity (Usability)	Adds an optional attribute to the character and varchar type column attributes of tables. Allows search and compare of these columns in a case insensitive manner. Four new SQLCore data types are added. Changed syntax: ALTER TABLE, CREATE TABLE.	*ALLBASE/SQL Reference Manual*, "Comparison Predicate" in "Search Conditions," CREATE TABLE in "SQL Statements."
Support for 1023 columns (Usability)	Increases the maximum number of columns per table or view to 1023. Increases maximum sort columns and parameters in a procedure to 1023.	*ALLBASE/SQL Reference Manual*, CREATE TABLE and CREATE VIEW in "SQL Statements;" *ALLBASE/SQL Database Administration Guide*, "ALLBASE/SQL Limits" appendix.
ISQL HELP improvements (Usability)	Gives help for entire command instead of only the verb.	*ISQL Reference Manual for ALLBASE/SQL and IMAGE/SQL*, HELP in "ISQL Commands."
EXTRACT command (Usability)	Extracts modules from the database and stores them in a module file. Allows for creation of a module file at any time based on the current DBEnvironment without preprocessing. New command: EXTRACT. Changed syntax: INSTALL.	*ISQL Reference Manual for ALLBASE/SQL and IMAGE/SQL*, "Using Modules" in "Using ISQL for Database Tasks," EXTRACT, INSTALL in "ISQL Commands."

New Features in ALLBASE/SQL Release G.0 (continued)

Feature (Category)	Description	Documented in . . .
New SQLGEN GENERATE parameters (Usability)	Generates SQL statements necessary to recreate modified access plans for module sections. New syntax for GENERATE: DEFAULTSPACE, MODOPTINFO, PARTITION, PROCOPTINFO, SPACEAUTH.	*ALLBASE/SQL Database Administration Guide*, "SQLGEN Commands" appendix.
Row level locking (Usability)	Permits multiple transactions to read and update a table concurrently because locking is done at row level. Since the transaction will obtain more locks, the benefits must be weighed against the costs. (Previously documented in an addendum after F.0 release.)	*ALLBASE/SQL Reference Manual*, "Concurrency Control through Locks and Isolation Levels;" *ALLBASE/SQL Database Administration Guide*, "Effects of Page and Row Level Locking" in "Physical Design."
Increased number of users (Usability)	Removes the limitation of 240 users supported by pseudotables. (Maximum is system session limits: 2000 on HP-UX; 1700 on MPE/iX.)	*ALLBASE/SQL Database Administration Guide*, "ALLBASE/SQL Limits" appendix.
POSIX support (Usability)	Improves application portability across MPE/iX and HP-UX. Enhances the ALLBASE/SQL preprocessors to run under POSIX (Portable Operating System Interface) on MPE/iX.	*ALLBASE/SQL Advanced Application Programming Guide*, "POSIX Preprocessor Invocation" in "Using the Preprocessor."
Application thread support (Performance, Usability)	Provides the use of threads in an application. Allows ALLBASE/SQL to be used in an application threaded environment on MPE/iX. Application threads are light weight processes that share some resources and last for the duration of a transaction. Threaded applications reduce the overhead of context switching and improve the performance of OpenTP applications.	*ALLBASE/SQL Advanced Application Programming Guide*, "Using the Preprocessor."

New Features in ALLBASE/SQL Release G.0 (continued)

Feature (Category)	Description	Documented in . . .
High Availability	Provides a collection of features to keep systems available nonstop including: Partial STORE and RESTORE, Partial rollforward recovery, DBEFiles in different groups (MPE/iX), detaching and attaching database objects, CHECKPOINT host variable, changing log files, console messages logged to a file, generating fewer log records by using TRUNCATE TABLE to delete rows, and new system catalog information. See the following features for new and changed syntax.	*ALLBASE/SQL Reference Manual*, "SQL Statements;" *ALLBASE/SQL Database Administration Guide*, "Maintaining a Nonstop Production System" in "Maintenance" chapter and "SQLUtil" appendix.
Partial rollforward recovery (High Availability)	Supports partial rollforward recovery through PARTIAL option on SETUPRECOVERY. Used to recover specific DBEFiles while allowing access to other DBEFiles.	*ALLBASE/SQL Database Administration Guide*, "Backup and Recovery" chapter and SETUPRECOVERY PARTIAL in "SQLUtil" appendix.
Partial STORE and RESTORE (High Availability)	Gives more flexibility in backup and recovery strategies by allowing partial store and restore of DBEFiles, DBEFileSets or combinations of both. See "New and changed SQLUtil commands for increased availability" later in this table.	*ALLBASE/SQL Database Administration Guide*, "Backup and Recovery" chapter and "SQLUtil" appendix.
DBEFile group change on MPE/iX (High Availability)	Manages DBEFiles so they can be placed in a particular group or on a particular volume (MPE/iX). Use either CREATE DBEFILE or MOVEFILE.	*ALLBASE/SQL Reference Manual*, CREATE DBEFile in "SQL Statements;" *ALLBASE/SQL Database Administration Guide*, "Maintaining a Nonstop Production System" in "Maintenance" chapter and MOVEFILE in "SQLUtil" appendix.
Detaching and attaching database objects (High Availability)	Detaches or attaches a DBEFile or DBEFileSet from the DBEnvironment. This is useful for data that is accessed infrequently such as tables containing historical data only. New SQLUtil commands: DETACHFILE, ATTACHFILE.	*ALLBASE/SQL Database Administration Guide*, "Maintaining a Nonstop Production System" in "Maintenance" chapter and DETACHFILE, ATTACHFILE in "SQLUtil" appendix.

New Features in ALLBASE/SQL Release G.0 (continued)

Feature (Category)	Description	Documented in . . .
New and changed SQLUtil commands for increased availability (High Availability)	Adds support for high availability and System Management Intrinsics. Intended for non-stop, continuously available operations. New SQLUtil commands: ATTACHFILE, CHANGELOG, DETACHFILE, RESTORE PARTIAL, STORE PARTIAL, STOREINFO, STOREONLINE PARTIAL, WRAPDBE. Modified SQLUtil commands: MOVEFILE, RESTORE, RESTORELOG, SHOWDBE, SETUPRECOVERY, STORE, STORELOG, STOREONLINE.	*ALLBASE/SQL Database Administration Guide*, "SQLUtil" appendix.
List files on backup device (High Availability)	Lists physical names of files stored on backup device with new SQLUtil command: STOREINFO.	*ALLBASE/SQL Database Administration Guide*, "Backup and Recovery" chapter and STOREINFO in "SQLUtil" appendix.
Log file improvements (High Availability)	Allows changing log files, switching of console messages to a file, and gives advance warning for log full. Increased maximum size of a single DBE log file to 4 gigabytes. A DBEnvironment can have up to 34 log files configured. Changed syntax: CHECKPOINT. New SQLUtil command: CHANGELOG.	*ALLBASE/SQL Reference Manual*, CHECKPOINT in "SQL Statements;" *ALLBASE/SQL Database Administration Guide*, "Maintaining a Nonstop Production System" in "Maintenance" chapter, CHANGELOG in "SQLUtil" appendix, and "ALLBASE/SQL Limits" appendix.
New SET SESSION and SET TRANSACTION statements (Standards, Performance)	Provides additional flexibility and improved performance. Allows setting and changing transaction and session attributes.	*ALLBASE/SQL Reference Manual*, SET SESSION and SET TRANSACTION in "SQL Statements."
FIPS flagger (Standards)	Meets Federal Information Processing Standard (FIPS) 127.1 flagger support. Flags non-standard statement or extension. Invoked with a flagger option in the preprocessor command line or the SET FLAGGER command in ISQL. Updatability rules are different when flagger is invoked. New syntax: DECLARE CURSOR, WHENEVER. Changes to C and COBOL host variable declaration.	*ALLBASE/SQL Reference Manual*, DECLARE CURSOR in "SQL Commands" and "Standards Flagging Support" appendix; *ALLBASE/SQL Advanced Application Programming Guide*, "Flagging Non-Standard SQL with the FIPS Flagger;" *ISQL Reference Manual for ALLBASE/SQL and IMAGE/SQL*, SET in "ISQL Commands."

New Features in ALLBASE/SQL Release G.0 (continued)

Feature (Category)	Description	Documented in . . .
Optimizer enhancement (Performance)	Uses a more efficient algorithm that significantly reduces the time to generate the access plan.	*ALLBASE/SQL Performance and Monitoring Guidelines*, "Optimization" in "Basic Concepts in ALLBASE/SQL Performance."
Access plan modification (Performance)	Allows modification of access plans for stored section to optimize performance. View the plan with SYSTEM.SETOPTINFO. New statement: SETOPT.	*ALLBASE/SQL Reference Manual*, SETOPT in "SQL Statements;" *ALLBASE/SQL Database Administration Guide*, SYSTEM.SETOPINFO in "System Catalog."
Syntax added to disable access plan optimization (Performance, Usability)	Specifies that the optimization information in the module file is not to be used. Changed syntax: EXTRACT, INSTALL, VALIDATE.	*ALLBASE/SQL Reference Manual*, VALIDATE in "SQL Statements; *ISQL Reference Manual for ALLBASE/SQL and IMAGE/SQL*," EXTRACT, INSTALL in "ISQL Commands."
Application Development Concurrency (Performance, Usability)	Provides enhancements to improve preprocessing performance when simultaneously accessed by multiple users. Page or row level locking on any system base table and processing without storing sections. See the related features in this table. New SQL parameter: SET DEFAULT DBEFileSet. SQL changed syntax: ALTER TABLE, GRANT, REVOKE, UPDATE STATISTICS. ISQL changed syntax: INSTALL. Changed SYSTEM and CATALOG view. New STOREDSECT tables. Special owners HPRDBSS and STOREDSECT. Changed syntax for Full Preprocessing Mode.	*ALLBASE/SQL Reference Manual*, "Names" and "SQL Statements;" *ALLBASE/SQL Advanced Application Programming Guide*, "Using the Preprocessor;" *ISQL Reference Manual for ALLBASE/SQL and IMAGE/SQL*, "ISQL Commands;" *ALLBASE/SQL Database Administration Guide*, "Database Creation and Security" and "System Catalog."
System Catalog tables (Performance)	Provides greater concurrency by allowing users to specify table, page, or row level locking of any system table owned by STOREDSECT through the ALTER TABLE statement.	*ALLBASE/SQL Reference Manual*, "Names;" *ALLBASE/SQL Database Administration Guide*, "System Catalog."
Preprocessors (Performance)	Allows optional specification of a DBEFileSet for storage of sections. Allows preprocessing without storing sections in DBEnvironment.	*ALLBASE/SQL Advanced Application Programming Guide*, "Using the Preprocessor."

New Features in ALLBASE/SQL Release G.0 (continued)

Feature (Category)	Description	Documented in . . .
I/O performance improvement (Performance)	Optimizes I/O for initial load, index build, serial scans, internal data restructuring, file activity, pseudo mapped files and temporary files. See the following features for new and changed syntax.	*ALLBASE/SQL Reference Manual*, "SQL Statements."
TRUNCATE TABLE statement (Performance)	Deletes all rows in a specified table leaving its structure intact. Indexes, views, default values, constraints, rules defined on the table, and all authorizations are retained. TRUNCATE TABLE is faster than the DELETE statement and generates fewer logs. New statement: TRUNCATE TABLE.	*ALLBASE/SQL Reference Manual*, TRUNCATE TABLE in "SQL Statements."
New scans (Performance)	Reads tables with a new parallel sequential scan. The tables are partitioned and files are read in a round robin fashion to allow OS prefetch to be more effective. Allows the I/O for a serial scan to spread across multiple disc drives.	*ALLBASE/SQL Performance and Monitoring Guidelines*, "Using Parallel Serial Scans" in "Guidelines on Query Design."
Load performance improvement (Performance)	Improves performance with new SET and SET SESSION attributes, a new binary search algorithm, and deferred allocation of HASH pages. New attributes for SET SESSION statement: FILL, PARALLEL FILL.	*ALLBASE/SQL Reference Manual*, SET SESSION in "SQL Statements."
ISQL enhanced to improve the performance of LOADs (Performance)	Uses new parameters of the ISQL SET command to set load buffer size and message reporting. Improves load performance. Choose a procedure, command file, or new ISQL command to set constraints deferred, lock table exclusively, and set row level DML atomicity. Changed syntax: SET (see the following feature).	*ISQL Reference Manual for ALLBASE/SQL and IMAGE/SQL*, SET in "ISQL Commands."

New Features in ALLBASE/SQL Release G.0 (continued)

Feature (Category)	Description	Documented in . . .
Modified SET options (Performance)	Provides better performance for LOADs and UNLOADs. Specify buffer size, status reporting for LOAD/UNLOAD or exclusive lock for data table. AUTOSAVE row limit increased to 2147483647. New and changed SET options: LOAD_BUFFER, LOAD_ECHO, AUTOLOCK, AUTOSAVE.	*ISQL Reference Manual for ALLBASE/SQL and IMAGE/SQL*, SET in "ISQL Commands;" *ALLBASE/SQL Performance and Monitoring Guidelines*, "Initial Table Loads" in "Guidelines on Logical and Physical Design."
SQLMON (Tools)	Monitors the activity of ALLBASE/SQL DBEnvironment. Provides information on file capacity, locking, I/O, logging, tables, and indexes. Summarizes activity for entire DBEnvironment or focuses on individual sessions, programs, or database components. Provides read-only information.	*ALLBASE/SQL Performance and Monitoring Guidelines*, chapters 6-9.
Audit (Tools)	Provides a series of features to set up an audit DBEnvironment which generates audit log records that you can analyze with the new SQLAudit utility for security or administration. Includes the ability to set up partitions. See *ALLBASE/SQL Database Administration Guide* for SQLAudit commands. Modified statements: ALTER TABLE, CREATE TABLE, START DBE NEW, START DBE NEWLOG. New statements: CREATE PARTITION, DROP PARTITION, DISABLE AUDIT LOGGING, ENABLE AUDIT LOGGING, LOG COMMENT.	*ALLBASE/SQL Reference Manual*, "SQL Statements;" *ALLBASE/SQL Database Administration Guide*, "DBEnvironment Configuration and Security" chapter and "SQLAudit" appendix.
Wrapper DBEnvironments (Tools)	Creates a DBEnvironment to wrap around the log files orphaned after a hard crash of DBEnvironment. New SQLUtil command: WRAPDBE.	*ALLBASE/SQL Reference Manual*, "Wrapper DBEnvironments" in "Using ALLBASE/SQL;" *ALLBASE/SQL Database Administration Guide*, WRAPDBE in "SQLUtil."
HP PC API is now bundled with ALLBASE/SQL.	PC API is an application programming interface that allows tools written with either the GUPTA or the ODBC interface to access ALLBASE/SQL and IMAGE/SQL from a PC.	*HP PC API User's Guide for ALLBASE/SQL and IMAGE/SQL.*

New Features in ALLBASE/SQL Release G.0 (continued)

Feature (Category)	Description	Documented in . . .
Increased memory for MPE/iX (HP-UX shared memory allocation is unchanged) (Performance)	Increases memory up to 50,000 data buffer pages and 2,000 run time control block pages. Increases the limits significantly allowing allocation of enough data buffer pages to keep the entire DBEnvironment in memory if desired for performance.	*ALLBASE/SQL Reference Manual*, STARTDBE, STARTDBE NEW, and START DBE NEWLOG in "SQL Statements;" *ALLBASE/SQL Database Administration Guide*, "ALLBASE/SQL Limits" appendix.
ALLBASE/NET enhancements (Connectivity, Performance)	Improves performance of ALLBASE/NET, allows more client connections on server system, and reduces number of programs on MPE/iX.	*ALLBASE/NET User's Guide*, "Setting up ALLBASE/NET."
ALLBASE/NET commands and options for MPE/iX (Connectivity, Usability)	Adds option ARPA. Adds option NUMSERVERS to check status of listeners and number of network connections. Changed syntax: ANSTART, ANSTAT, ANSTOP. Changed NETUtil commands: ADD ALIAS, CHANGE ALIAS.	*ALLBASE/NET User's Guide*, "Setting up ALLBASE/NET" and "NETUtil Reference."
ALLBASE/NET and NetWare (Connectivity)	ALLBASE/NET listener for NetWare now works with the 3.11 version of Novell's NetWare for UNIX (HP NetWare/iX).	*ALLBASE/NET User's Guide*, "Setting up ALLBASE/NET."
Changed restrictions for executing NETUtil commands for MPE/iX (Connectivity, Usability)	Adds SM or AM (in the specified account) to MANAGER.SYS for adding, changing, or deleting users for MPE/iX.	*ALLBASE/NET User's Guide*, "Setting up ALLBASE/NET."
ARPA is only TCP/IP interface for data communication through ALLBASE/NET beginning with HP-UX 10.0 (Connectivity)	Remote database access applications that specify NS will not work if the client and/or server machine is an HP 9000 Series 700/800 running HP-UX 10.0 or greater. Server Node Name entry must be changed from NS node name to ARPA host name. For the NETUsers file, the "Client Node Name" must be changed from the NS node name to the ARPA host name. New NETUtil commands: MIGRATE USER, MIGRATE ALIAS.	*ALLBASE/NET User's Guide*, "Setting up ALLBASE/NET" and "NETUtil Reference."

Conventions

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

```
COMMAND
```

can be entered as any of the following:

```
command        Command        COMMAND
```

It cannot, however, be entered as:

```
comm           com_mand       comamnd
```

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

```
COMMAND filename
```

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

```
(filename):(filename)
```

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

```
Do you want to continue? >>  yes
```

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

```
COMMAND { ON  }
        { OFF }
```

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

```
COMMAND filename [OPTION]
```

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

```
COMMAND filename [ OPTION    ]
                 [ parameter ]
```

`[ ... ]` 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.

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

`...` 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

Tables

1

Using the Preprocessor

This chapter details complete syntax for each ALLBASE/SQL preprocessing mode, for each supported programming language (C, COBOL, FORTRAN, and Pascal). Related coding techniques are also discussed. Main topics include:

- Preprocessing in Full Preprocessing Mode.
- Preprocessing in Static Conversion Mode.
- Preprocessing in Syntax Checking Mode.
- POSIX Preprocessor Invocation.

For information about preprocessor input and output files, compiling and linking, example applications, and other preprocessor topics related to a specific language, refer to the ALLBASE/SQL application programming guide for the language you are using.

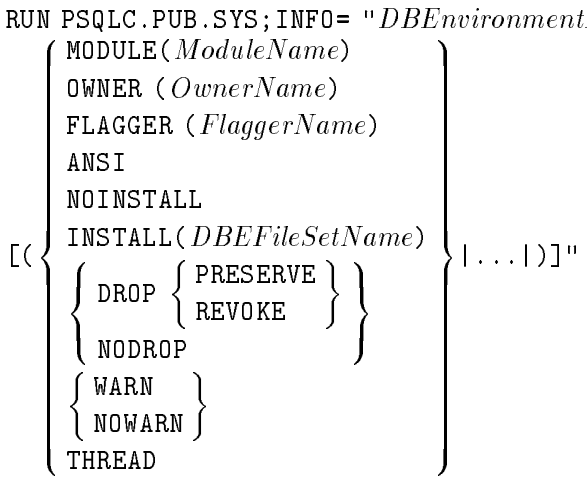
Full Preprocessing Mode

Full preprocessing mode includes the following:

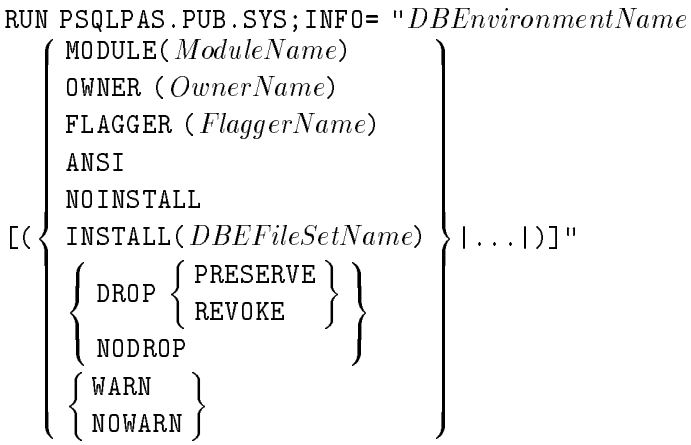
- Checking SQL syntax.
- Creating compilable output files.
- Optionally storing an installable module containing sections in a DBEnvironment. (See the NOINSTALL option below.)
- Creating a file that contains an installable copy of the module.

Full Preprocessing Mode Syntax Specification

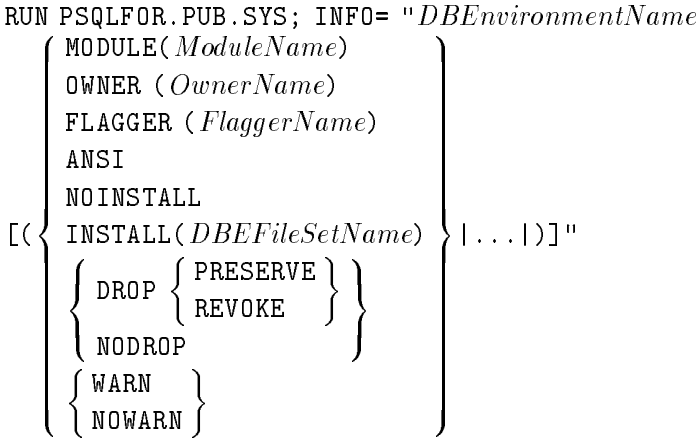
Full Preprocessing Mode for C Applications



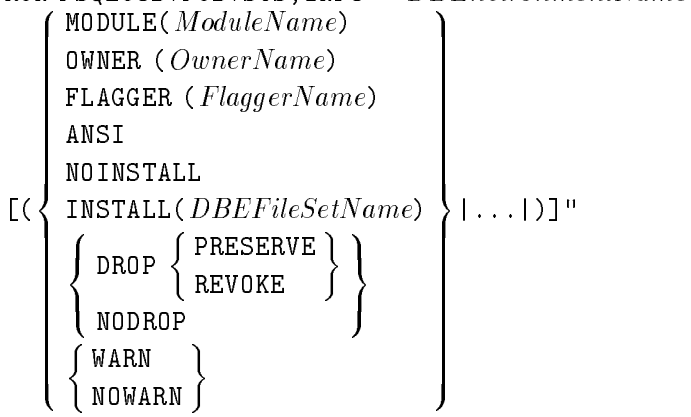
Full Preprocessing Mode for COBOL Applications



Full Preprocessing Mode for FORTRAN Applications



Full Preprocessing Mode for Pascal Applications



Parameters

DBEnvironmentName identifies the DBEnvironment in which a module is to be stored. You can use a backreference to a file defined in a file equation 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-ID (for a C application) or the *SourceFileName* (for other languages) in upper case as the module name. This module name is stored in the module file, SQLMOD.
OwnerName	associates the stored module with a *User@Account*, a class name, or a group name. You can specify an owner name for the module only if you have DBA authority in the DBEnvironment where the module is to be stored. If not specified, the owner name is your log-on *User@Account*. Any object names in SQLIN not qualified with an owner name are qualified with this *OwnerName*.
FlaggerName	is the name of the flagger being invoked. At this release FIPS127.1 is the only valid *FlaggerName*. *FlaggerName* is not case sensitive. Refer to the "Flagging Non-Standard SQL with the FIPS Flagger" chapter in this manual.
ANSI	preprocesses the application in ANSI mode. The preprocessor generates an SQLCA declaration automatically and implicit updatability is in effect. Refer to the sections "Understanding Implicit Updatability" and "Declaring the SQLCA" in this manual.
NOINSTALL	indicates that no sections are to be stored in a DBEnvironment during preprocessing. A module that can be installed in a DBEnvironment with the ISQL INSTALL command is generated. This is the only option for which a DBEnviromnent name is not required. If a DBEnvironment name is specified, a connection is established, and bind errors and warnings are generated for missing objects. If no DBEnvironment name is specified, no connection is made and no bind errors or warnings are generated.
INSTALL *DBEFileSetName*	identifies the DBEFileSet in which a module's static sections are to be stored. A *DBEFileSetName* specified in the preprocessor command line overrides any explicit specification in the IN *DBEFileSetName* clause of any ALTER TABLE, CREATE PROCEDURE, CREATE RULE, CREATE TABLE, CREATE VIEW, DECLARE CURSOR, or PREPARE statement within the preprocessed application.
`DROP`	Deletes any module currently stored in the DBEnvironment by the *ModuleName* and *OwnerName* specified in the INFO string.

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 by the *ModuleName* and *OwnerName* specified in the INFO string. If not specified, NODROP is assumed.
NOWARN	Converts preprocessor generated bind warnings to errors.
WARN	Generates the default bind warnings. If neither NOWARN or WARN is specified, the default bind warnings are generated.
THREAD	For C applications only; used to support threaded applications. When specified, the C preprocessor will not generate the #include "SQLVAR" at the beginning of SQLOUT. You must insert a #include "SQLVAR" wherever local variables are required to compile and utilize application threads successfully.

Description

- Unless you have specified the NOINSTALL option without specifying a DBEnvironment name, 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.
- Unless the NOINSTALL option is specified without specifying a DBEnvironment name, the preprocessor starts a DBE session in the DBEnvironment named in the preprocessor command by issuing a CONNECT TO '*DBEnvironmentName*' statement. If the autostart flag is OFF, the DBE session can be initiated only after a START DBE statement has been processed (by means of ISQL or an embedded SQL program). Remember that if multiple, simultaneous users run a program, you should issue the START DBE statement only once.
- 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.
- When the preprocessor's DBE session begins, ALLBASE/SQL processes a BEGIN WORK statement. When preprocessing is completed, the preprocessor submits a COMMIT WORK statement, and any sections created are committed to the system catalog. If the preprocessor detects an error in SQLIN, it processes a ROLLBACK WORK statement before terminating, and no sections are stored in the DBEnvironment. Preprocessor warnings do not prevent sections from being stored.
- Since any preprocessor DBE session initiates only one transaction, any log file space used by the session is not available for re-use until after the session terminates. You can issue a CHECKPOINT statement in ISQL *before* preprocessing to review the amount of available

log space. Refer to the *ALLBASE/SQL Database Administration Guide* for additional information on log space management, such as using the START DBE NEWLOG statement to increase the size of the log file.

- During preprocessing, system catalog pages accessed for embedded statements are locked. In multiuser mode, other DBE sessions accessing the same objects may have to wait, so there is potential for a deadlock. You have several options to use to attempt to avoid a deadlock situation.
  - Minimize competing transactions when preprocessing an application.
  - Ask your DBA to specify the system tables that utilize row level locking. Refer to the *ALLBASE/SQL Database Administration Guide* appendix, "Locks Held on the System Catalog," and to the section "Changing System Table Lock Types" for further information.
  - Use the INSTALL *(DEBFileSetName)* preprocessor parameter option to maximize transaction concurrency. This option specifies the DBEFileSet in which a module's static sections are to be stored.
- For improved runtime performance, use the UPDATE STATISTICS statement in ISQL *before* preprocessing for each table accessed in a data manipulation statement when an index on that table has been added or dropped and when data in the table is often changed.
- 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.
- When you invoke the preprocessor with the FLAGGER option, your application must contain an ANSI mode compiler directive in order for the compiler to detect non-standard statements. The following table lists the appropriate directive for each language:

Language	Directive
C	ccxl ... ;info='-Aa'
COBOL	$CONTROL STDWARN
FORTRAN	$OPTION ANSI ON$
Pascal	$standard_level 'ANSI'$

- If implicit updatability is specified (using the ANSI option), any column in a select list can be updated or deleted, so long as the FOR UPDATE clause has not been used in the SELECT statement definition. Refer to the later section, "Understanding Implicit Updatability."
- If a DBEFileSet is specified and the module owner does not have authority to store sections in that DBEFileSet, a warning is given and any sections are stored in the default DBEFileSet instead.

Authorization

To preprocess a program, you need DBA or CONNECT authority for the DBEnvironment specified in the preprocessor command line. You also need table and view authorities for the tables and views which the program will access at run time. To store sections in a specified DBEFileSet, the module owner must have SECTIONSPACE authority on the DBEFileSet.

DBEnvironment CONNECT authority can be explicitly granted. If you have DBECreator or DBA authority or module OWNER authority, you have CONNECT authority by default. SECTIONSPACE authority for other than the default DBEFileSet must be explicitly granted by a DBA. If you have DBA authority, you can issue the GRANT statement for any DBEFileSet.

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

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

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

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

Example of Full Preprocessing of a C Application

```
:FILE SQLIN=CEX2
:RUN PSQLC.PUB.SYS;INFO=&
"PartsDBE (MODULE(CEX2) OWNER(OwnerP@SomeAcct) REVOKE DROP)"

                                        WED, OCT 25, 1991,  1:38 PM
HP36216-02A.E1.02             C 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
HP32501A.07.20 EDIT/3000 FRI, OCT 27, 1991,  10:17 AM
(C)HEWLETT-PACKARD CO. 1990
/T SQLMSG;L ALL UNN
FILE UNNUMBERED

             .
             .
             .
SQLIN                = CEX2.SOMEGRP.SOMEACCT
DBEnvironment        = partsdbe
Module Name          = CEX2

        SELECT PARTNUMBER, PARTNAME, SALESPRICE INTO :PARTNUMBER, :PARTNAME,
        :SALESPRICE :SALESPRICEIND FROM PURCHDB.PARTS WHERE PARTNUMBER =
        :PARTNUMBER ;

******  ALLBASE/SQL warnings (DBWARN 10602)
******  in SQL statement ending in line 133
***     User SomeUser@SomeAcct does not have SELECT authority on PURCHDB.PARTS.
(DBERR 2301)

  1 Sections stored in DBEnvironment.

 0 ERRORS   1 WARNINGS
END OF PREPROCESSING

/
```

DBEnvironment Access in Full Preprocessing Mode

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

```
: RUN PSQLC.PUB.SYS; INFO="DBEnvironment (MODULE (ModuleName))"
```

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

ALLBASE/SQL accesses the specified DBEnvironment during preprocessing, even if your application does not use SQL statements that store sections in this DBEnvironment. Therefore, unless you specify the -N option, you must specify the name of a valid DBEnvironment. Note that a DBEnvironment name is *not required* by other preprocessing modes.

Accessing Multiple DBEnvironments in Full Preprocessing Mode

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 can use ISQL to install the installable module created by the preprocessor into each additional DBEnvironment accessed by your program.

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

For example, to preprocess several ALLBASE/SQL application programs in the same group and account and compile and link the programs later, 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
```

- Run the preprocessor for each DBEnvironment and module, as in the following example in C:

```
:RUN PSQLC.PUB.SYS;INFO="DBEnvironment1 (MODULE (ModuleName1))"
```

- After running the preprocessor, save and rename the output files so that they will not be overwritten. For example:

```
:SAVE SQLOUT
:RENAME SQLOUT, OutFile1
:SAVE SQLVAR
:RENAME SQLVAR, VarFile1
```

- When you are ready to compile the program, you must equate each output file name for a given DBEnvironment/module combination to its standard ALLBASE/SQL name, then issue the compile command, as in the following example:

```
:FILE SQLOUT = OutFile1
:FILE SQLVAR = VarFile1
    .
    .
:CCXL SQLOUT, SQLOBJ1
```

- After all applications have been complied, link their object files, as follows:

```
    .
    .
:LINK FROM=SQLOBJ1,SQLOBJ2;RL=STDRL.LIB.SYS;TO=SOMEPROG
```

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

Static Conversion Mode

The DYNAMIC preprocessor command line option provides a means of converting static SQL statements to dynamic statements with little or no change to existing source code. This is termed **static conversion processing** and provides the flexibility of running an application with either static processing or dynamic processing. In addition, an already dynamic application can be preprocessed without having to specify a DBEnvironment name in the command line.

This functionality is available for C and Pascal applications.

The following sections describe how to implement this feature. Note that additional information and examples regarding static and dynamic processing are found in the language specific ALLBASE/SQL application programming guides and in later sections in this chapter.

Static Conversion Mode Syntax Specification

Use one of the following statements to both check SQL syntax and create output files that can be processed by the compiler. All static SQL statements are converted to dynamic statements. Dynamic SQL statements remain dynamic. No module is created, and no DBEnvironment is connected to during preprocessing.

Static Conversion Mode for C Applications

```
RUN PSQLC.PUB.SYS;INFO= "(DYNAMIC [ FLAGGER (FlaggerName) ]
                                  [ ANSI                  ] )"
                                  [ THREAD                ]
```

Static Conversion Mode for Pascal Applications

```
RUN PSQLPAS.PUB.SYS;INFO= "(DYNAMIC [ FLAGGER (FlaggerName) ] )"
                                    [ ANSI                  ]
```

Parameters

DYNAMIC	indicates that all static SQL statements are to be converted to dynamic statements, dynamic statements remain dynamic, and a module is not created. DYNAMIC can be specified in upper and/or lower case.
FlaggerName	is the name of the flagger being invoked. At this release FIPS127.1 is the only valid *FlaggerName*. *FlaggerName* is not case sensitive. Refer to the "Flagging Non-Standard SQL with the FIPS Flagger" chapter in this manual.
ANSI	preprocesses the application in ANSI mode. The preprocessor generates an SQLCA declaration automatically and implicit updatability is in effect. Refer to the sections "Understanding Implicit Updatability" and "Declaring the SQLCA" in this manual.
THREAD	For C applications only; used to support threaded applications. When specified, the C preprocessor will not generate the #include "SQLVAR" line at the beginning of SQLOUT. The user must

insert a #include "SQLVAR" wherever local variables are required to compile and utilize application threads successfully.

Description

- The complete set of ALLBASE/SQL syntax can be preprocessed in static conversion mode with the following execeptions:
  - Any DECLARE CURSOR statements must be located in an executable portion of the code. In the C language, for example, you would declare the cursor within curly brackets following any data declaration statements within a given block. For Pascal, position the statement within a BEGIN/END block.
  - Static host variables in the source code (except for those used in a BULK FETCH or BULK SELECT statement) are converted to dynamic parameters. Therefore, such static host variables must adhere to the specifications for dynamic parameters. For further information, refer to the "Using Parameter Substitution in Dynamic Statements" chapter in this document.
- When you invoke the preprocessor with the FLAGGER option, your application must contain an ANSI mode compiler directive in order for the compiler to detect non-standard statements. The following table lists the appropriate directive for each language:

Language	Directive
C	ccxl ... ;info='-Aa'
COBOL	$CONTROL STDWARN
FORTRAN	$OPTION ANSI ON$
Pascal	$standard_level 'ANSI'$

Authorization

At run time, the owner of any unqualified object defaults to the user who is running the application.

Example of Static Conversion Processing of a C Application

```
RUN PSQLC.PUB.SYS;INFO="(DYNAMIC)"
```

Example of Static Conversion Processing of a Pascal Application

```
RUN PSQLPAS.PUB.SYS;INFO="(DYNAMIC)"
```

Before running the preprocessor, remember to equate SQLIN to the name of the file containing the application you want to preprocess. After running the preprocessor, save and rename any output files you do not want overwritten.

Syntax Checking Mode

In **syntax checking mode**, the preprocessor checks SQL syntax only. This mode can be useful as you develop the SQL portions of your applications. Preprocessing is quickest in this mode. In addition, you can start debugging your SQL statements before the DBEnvironment itself is in place.

Syntax Checking Mode Syntax Specification

Syntax Checking Mode for C Applications

```
:RUN PSQLC.PUB.SYS;INFO="(SYNTAX [ FLAGGER (FlaggerName) ])"
                                 [ ANSI                  ]
                                 [ THREAD                ]
```

Syntax Checking Mode for COBOL Applications

```
:RUN PSQLCOB.PUB.SYS;INFO="(SYNTAX [ FLAGGER (FlaggerName) ])"
                                   [ ANSI                  ]
```

Syntax Checking Mode for FORTRAN Applications

```
:RUN PSQLFOR.PUB.SYS;INFO="(SYNTAX [ FLAGGER (FlaggerName) ])"
                                   [ ANSI                  ]
```

Syntax Checking Mode for Pascal Applications

```
:RUN PSQLPAS.PUB.SYS;INFO="(SYNTAX [ FLAGGER (FlaggerName) ])"
                                   [ ANSI                  ]
```

Parameters

SYNTAX	causes the preprocessor to only check SQL syntax.
FlaggerName	is the name of the flagger being invoked. At this release FIPS127.1 is the only valid *FlaggerName*. *FlaggerName* is not case sensitive. Refer to the "Flagging Non-Standard SQL with the FIPS Flagger" chapter in this manual.
ANSI	preprocesses the application in ANSI mode. The preprocessor generates an SQLCA declaration automatically and implicit updatability is in effect. Refer to the sections "Understanding Implicit Updatability" and "Declaring the SQLCA" in this manual.
THREAD	For C applications only; used to support threaded applications. When specified, the C preprocessor will not generate the #include "SQLVAR" line at the beginning of SQLOUT. The user must insert a #include "SQLVAR" wherever local variables are required to compile and utilize application threads successfully.

Description

- The preprocessor does not access a DBEnvironment when it is run in this mode.
- When performing syntax only checking, the preprocessor does not convert embedded SQL statements into constructs for the programming language being used. Therefore the modified source code file does not contain preprocessor generated calls to ALLBASE/SQL external procedures.
- The include and installable module files are created, but are incomplete.
- When you invoke the preprocessor with the FLAGGER option, your application must contain an ANSI mode compiler directive in order for the compiler to detect non-standard statements. The following table lists the appropriate directive for each language:

Language	Directive
C	ccxl ... ;info='-Aa'
COBOL	$CONTROL STDWARN
FORTRAN	$OPTION ANSI ON$
Pascal	$standard_level 'ANSI'$

Authorization

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

Example of Syntax Checking of a C Application

```
:FILE SQLIN=CEX2
:RUN PSQLC.PUB.SYS;INFO="(SYNTAX)"

                                              WED, OCT 25, 1991,  1:38 PM
HP36216-02A.E1.02              C 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
HP32501A.07.20 EDIT/3000 FRI, OCT 27, 1991,  9:35 AM
(C) HEWLETT-PACKARD CO. 1985
/T SQLMSG;L ALL UNN
FILE UNNUMBERED

               .
               .
               .
SQLIN                 = CEX2.SOMEGRP.SOMEACCT


        SELECT 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 176
***     Unexpected keyword.  (DBERR 1006)

Syntax checked.

 1 ERRORS   0 WARNINGS
END OF PREPROCESSING.

/
```

The line 176 referenced in SQLMSG is the line in SQLIN where the erroneous SQL statement ends.

POSIX Preprocessor Invocation

You can invoke the C preprocessor in any preprocessing mode, using POSIX command line options as described in the following sections. The preprocessor generates output files as POSIX byte-stream files (compatible with UNIX) using the naming conventions documented in the ALLBASE/SQL application programming guides for HP-UX.

An existing DBEnvironment name can be specified in the preprocessor command line in hierarchical file system (HFS) format. An *HFS file name* is one beginning with either a '.' or a '/.' The preprocessor converts the HFS DBEnvironment name to a conventional MPE/iX style file name (*FileName.GroupName.AccountName*). In the following example, the preprocessor converts the DBEnvironment name /SYS/SAMPLEDB/PARTSDBE to PARTSDBE.SAMPLEDB.SYS.

```
POSIX> /SYS/PUB/PSQLC /SYS/SAMPLEDB/PARTSDBE -i ./Source.sql -d -m Module
```

HFS file names of up to 1024 bytes are supported.

You can also specify HFS file names in #INCLUDE statements. These files are included during preprocessing.

POSIX Full Preprocessing Mode for C Applications

```
POSIX> /SYS/PUB/PSQLC DBEnvironmentName
[-i SourceFileName                    ]
[-p ModifiedSourceFileName            ]
[-o OwnerName                         ]
[-m ModuleName                        ]
[-f FlaggerName                       ]
[-a                                   ] | ... |"
[-N                                   ]
[-n DBEFileSetName                    ]
[-d [-r]                              ]
[-W                                   ]
[-t                                   ]
```

Parameters

DBEnvironmentName	identifies the DBEnvironment in which a module is to be stored. This name must be specified in upper case.
-i *SourceFileName*	identifies the name of the input file containing the source code to be preprocessed. If not specified, ./sqlin is the default.
-p *ModifiedSourceFileName*	identifies the name for the modified source code file. If a name is not specified, the preprocessor generated code is written to a file with the same name as *SourceFileName* and with an appropriate file extension, as follows:

Language	File Name Extension
C	*SourceFileName*.c
COBOL	*SourceFileName*.cbl
FORTRAN	*SourceFileName*.f
Pascal	*SourceFileName*.p

-o *OwnerName*

associates the stored module with a *User@Account*, a class name, or a group name. You can specify an owner name for the module only if you have DBA authority in the DBEnvironment where the module is to be stored. If not specified, the owner name is your log-on *User@Account*. Any object names in *SourceFileName* not qualified with an owner name are qualified with this *OwnerName*.

-m *ModuleName*

assigns a name to the stored module. Module names must follow the rules governing ALLBASE/SQL basic names as described in the *ALLBASE/SQL Reference Manual*. If a module name is not specified, the preprocessor uses the PROGRAM-ID in upper case as the module name. This module name is stored in the module file, *ModifiedSourceFileName*.sqlm.

-f *FlaggerName*

is the name of the flagger being invoked. At this release FIPS127.1 is the only valid *FlaggerName*. *FlaggerName* is not case sensitive.

Refer to the "Flagging Non-Standard SQL with the FIPS Flagger" chapter in this manual.

-a

preprocesses the application in ANSI mode. The preprocessor generates an SQLCA declaration automatically and implicit updatability is in effect.

Refer to the sections "Understanding Implicit Updatability" and "Declaring the SQLCA" in this manual.

-N

indicates that no sections are to be stored in a DBEnvironment during preprocessing. A module that can be installed in a DBEnvironment with the ISQL INSTALL command is generated.

This is the only option for which a DBEnviromnent name is not required. If a DBEnvironment name is specified, a connection is established, and bind errors and warnings are generated for missing objects. If no DBEnvironment name is specified, no connection is made and no bind errors or warnings are generated.

-n *DBEFileSetName*

identifies the DBEFileSet in which a module's static sections are to be stored.

A *DBEFileSetName* specified in the preprocessor command line overrides any explicit specification in the IN *DBEFileSetName* clause of any ALTER TABLE, CREATE PROCEDURE, CREATE RULE, CREATE TABLE, CREATE VIEW,

DECLARE CURSOR, or PREPARE statement within the preprocessed application.

-d — deletes any module currently stored in the DBEnvironment by the *ModuleName* and *OwnerName* specified in the command string. If not specified, any module having these names is not dropped, and existing RUN authorities for that module are preserved.

-r — is specified when the program being preprocessed already has a stored module and you want to revoke existing RUN authorities for that module. The -r option cannot be specified unless -d is also specified. If the -r option is not specified, it is assumed that all existing RUN authorities for that module are to be PRESERVED.

-W — converts preprocessor generated bind warnings to errors. If -W is not specified, the default bind warnings will be generated.

-t — For C applications only; used to support threaded applications. When specified, the C preprocessor will not generate the #include "SQLVAR" line at the beginning of SQLOUT. The user must insert a #include "SQLVAR" wherever local variables are required to compile and utilize application threads successfully.

POSIX Static Conversion Mode for C Applications

```
POSIX> /SYS/PUB/PSQLC -D [ -i SourceFileName
                         | -p ModifiedSourceFileName.c
                         | -f FlaggerName
                         | -a
                         | -t ] | ... |"
```

Parameters

-D — indicates that all static SQL statements are to be converted to dynamic statements, dynamic statements remain dynamic, and a module is not created.

-i *SourceFileName* — identifies the name of the input file containing the source code to be preprocessed. If not specified, ./sqlin is the default.

-p *ModifiedSourceFileName* — identifies the name for the modified source code file. If a name is not specified, *SourceFileName*.c is the default.

-f *FlaggerName* — is the name of the flagger being invoked. At this release FIPS127.1 is the only valid *FlaggerName*. *FlaggerName* is not case sensitive.

Refer to the "Flagging Non-Standard SQL with the FIPS Flagger" chapter in this manual.

-a — preprocesses the application in ANSI mode. The preprocessorgenerates an SQLCA declaration automatically and implicit updatability is in effect. Refer to the sections

"Understanding Implicit Updatability" and "Declaring the SQLCA" in this manual.

-t — For C applications only; used to support threaded applications. When specified, the C preprocessor will not generate the #include "SQLVAR" line at the beginning of SQLOUT. The user must insert a #include "SQLVAR" wherever local variables are required to compile and utilize application threads successfully.

POSIX Syntax Checking Mode for C Applications

```
POSIX> /SYS/PUB/PSQLC -s [ -i SourceFileName
                           -p ModifiedSourceFileName
                           -f FlaggerName
                           -a
                           -t ] | ... |"
```

Parameters

-s — causes the preprocessor to only check SQL syntax.

-i *SourceFileName* — identifies the name of the input file containing the source code to be preprocessed. If not specified, ./sqlin is the default.

-p *ModifiedSourceFileName* — identifies the name for the modified source code file. If a name is not specified, *SourceFileName*.c is the default.

-f *FlaggerName* — is the name of the flagger being invoked. At this release FIPS127.1 is the only valid *FlaggerName*. *FlaggerName* is not case sensitive.

Refer to the "Flagging Non-Standard SQL with the FIPS Flagger" chapter in this manual.

-a — preprocesses the application in ANSI mode. The preprocessor generates an SQLCA declaration automatically and implicit updatability is in effect.

Refer to the sections "Understanding Implicit Updatability" and "Declaring the SQLCA" in this manual.

-t — For C applications only; used to support threaded applications. When specified, the C preprocessor will not generate the #include "SQLVAR" line at the beginning of SQLOUT. The user must insert a #include "SQLVAR" wherever local variables are required to compile and utilize application threads successfully.

2

Flagging Non-Standard SQL with the FIPS Flagger

The United States government has adopted ANSI X3.135-1989, Database Language SQL, as the database language to be used by all federal departments and agencies. This SQL standard, known as Federal Information Processing Standard 127.1 (**FIPS 127.1**), requires that all syntax and processing that does not conform to the standard be flagged. In general, flagging provides a means of identifying SQL elements that may have to be modified if an application is to be moved from a nonconforming to a conforming SQL processing environment. And it aids in writing code that is portable between various SQL databases (such as tools code).

ALLBASE/SQL provides a FIPS flagger option for use when preprocessing. Refer to the appropriate preprocessing mode, presented earlier in this chapter, for FIPS flagger syntax. This section documents non-conformant features specific to embedded SQL programming. In addition, the "Standards Flagging Support" appendix in the *ALLBASE/SQL Reference Manual* lists non-conformant syntax and features, and the *ISQL Reference Manual for ALLBASE/SQL and IMAGE/SQL* discusses setting the FIPS flagger in ISQL. The following topics are presented here:

- Coding Tips
- Setting the ANSI Compiler Directive
- Identifying Non-Standard Features
- Understanding Implicit Updatability
- Declaring the SQLCA
- Secondary References to Non-Standard SQL
- Host Variable Data Type Declarations
- Host Variable Name Length Standards

Coding Tips

It is highly recommended that you first preprocess an application without setting the flagger until no ALLBASE/SQL errors are detected. Then preprocess using the flagger to determine if there are non-standard extensions in your code.

Note that for static statements, flagger warnings are detected and generated at preprocessing time. Whereas, warnings for dynamic statements are detected at preprocessing time and generated at run time.

Setting the ANSI Compiler Directive

When you invoke the preprocessor with the FLAGGER option, your application must contain an ANSI mode compiler directive in order for the compiler to detect non-standard statements. The following table lists the appropriate directive for each language:

Table 2-1. ANSI Compiler Directives for Flagging Non-Standard Syntax

Language	Directive
C	ccxl ... ;info='-Aa'
COBOL	$CONTROL STDWARN
FORTRAN	$OPTION ANSI ON$
Pascal	$standard_level 'ANSI'$

Identifying Non-Standard Features

The following table lists ALLBASE/SQL features that do not conform to the FIPS 127.1 standard. These features do not generate a flagger warning.

Table 2-2. ALLBASE/SQL Non-Standard Programming Features

Feature	Description
updatablility processing	Refer to the following section, "Understanding Implicit Updatability."
SQLCA declaration	Refer to the following section, "Declaring the SQLCA."
host language comments	ALLBASE/SQL allows host language comments within an SQL statement. The standard allows only SQL comments.
$SQL COPY compiler directive	ALLBASE/SQL supports the COBOL COPY statement. Refer to the *ALLBASE/SQL COBOL Application Programming Guide* for usage details.
SQLCODE in FORTRAN	The standard requires a variable declaration for SQLCOD, and ALLBASE/SQL supports the standard.
SQLIND	This host variable type is not standards compliant. The standard requires that an indicator variable be declared as any exact numeric data type of scale zero, and ALLBASE/SQL supports the standard with the exception of BULK processing.

Understanding Implicit Updatability

The FIPS 127.1 standard does not allow a FOR UPDATE clause in a DECLARE CURSOR statement. In this standard, updatability of a cursor rests solely on the cursor definition. In contrast, **ALLBASE/SQL default processing** (i.e., flagging is not in effect) of a DECLARE CURSOR statement having no FOR UPDATE clause is to allow neither update nor delete on any column in the cursor definition.

When the FIPS flagger is set and a FOR UPDATE clause is not specified, standard updatability processing takes effect. That is, any column in the cursor's select list can be updated or deleted. This is known as **implicit updatability**. When implicit updatability is in effect, a performance impact may be realized as compared to ALLBASE/SQL default processing, as follows:

- An index scan can only be applied to an EQUAL predicate containing an index column. An index scan is suppressed for a predicate containing a range factor.
- More severe locks (intent update) are required. Therefore, if the cursor is to be used only for reading data, unnecessary locking overhead my be incurred. To alleviate this type of overhead, a FOR READ ONLY clause can be used when declaring the cursor.

Note that when the FIPS flagger is set and a FOR UPDATE clause *is* used in the cursor definition, a warning message is generated, and ALLBASE/SQL default processing overrides the semantics of the standard.

Declaring the SQLCA

ALLBASE/SQL default processing requires the SQLCA to be declared within an embedded program. However, FIPS 127.1 does not. Therefore, when the flagger is set, the preprocessor generates the SQLCA declaration. In this case, if the SQLCA is already declared, as would be the case in a non-flagger environment, a processing error results.

Secondary References to Non-Standard SQL

In certain circumstances(paragraph 9c of FIPS 127.1) you may choose to use a nonstandard language extension (e.g. a COMPLEX data type for a FORTRAN application). It is required that the flagger detect all direct occurrences of such extensions. However, there is no requirement to detect secondary references to a non-standard extension. Secondary references may include variables, parameters, views, or other database identifiers that do not themselves violate syntax rules, but refer to an object that is non-standard or contain a non-standard extension. ALLBASE/SQL does not flag references to non-standard objects. For example, in the following statement, the use of View1.v1 in another SQL statement would be a secondary reference to the non-standard object View1 which contains the extension COUNT(*).

```
CREATE VIEW View1(v1) AS SELECT COUNT(*) FROM Table1;
```

Host Variable Data Type Declarations

The following tables list valid ALLBASE/SQL host variable data types for each supported language. FIPS 127.1 standards compliant data types are indicated with a YES. ALLBASE/SQL extensions to the standard are indicated with a NO.

Table 2-3. ALLBASE/SQL FIPS 127.1 Compliant Data Type Declarations for C

ALLBASE/SQL DATA TYPES	FIPS 127.1 Compliant?	C DATA DECLARATIONS
CHAR(1)	NO	char *dataname*;
CHAR(n)	YES	char *dataname*[n+1];
VARCHAR(n)	NO	char *dataname*[n+1]; *
SMALLINT	YES	short *dataname*; or
	NO	short int *dataname*;
INTEGER	NO	int *dataname*; or
	NO	long int *dataname*; or
	YES	long *dataname*;
REAL	YES	float *dataname*;
FLOAT(1..24)	NO	float *dataname*;
FLOAT(25..53)	NO	double *dataname*;
DOUBLE PRECISION	YES	double *dataname*;
BINARY	NO	sqlbinary *dataname*;
	NO	sqlbinary *dataname*[n];
VARBINARY	NO	sqlvarbinary *dataname*[m]; **
DECIMAL	NO	double *dataname*;
DATE	NO	char *dataname*[11];
TIME	NO	char *dataname*[9];
DATETIME	NO	char *dataname*[24];
INTERVAL	NO	char *dataname*[21];

* This declaration is for non-dynamic statements only. Refer to the chapter, "Using Dynamic Operations," in the *ALLBASE/SQL C Application Programming Guide* for a description of how to use VARCHAR dynamically.
** See the "BINARY Data" section in the *ALLBASE/SQL C Application Programming Guide* for the calculation of m.

Table 2-4.
ALLBASE/SQL FIPS 127.1 Compliant Data Type Declarations for COBOL

SQL DATA TYPES	FIPS 127.1 Compliant?	COBOL DATA DESCRIPTION ENTRIES
CHAR(n)	YES	01 *DATA-NAME* PIC X(*n*).
VARCHAR(n)	NO	01 *GROUP-NAME*. 49 *LENGTH-NAME* PIC S9(9) COMP. 49 *VALUE-NAME* PIC X(*n*).
BINARY	NO	01 *DATA-NAME* PIC X(*n*).
VARBINARY(n)	NO	01 *GROUP-NAME*. 49 *LENGTH-NAME* PIC S9(9) COMP. 49 *VALUE-NAME* PIC X(*n*).
SMALLINT	NO	01 *DATA-NAME* PIC S9(4) COMP.
INTEGER	YES	01 *DATA-NAME* PIC S9(9) COMP.
FLOAT	NO	01 *DATA-NAME* PIC S9(p-s)V9(s) COMP-3.
DECIMAL(p,s)	NO	01 *DATA-NAME* PIC S9(p-s)V9(s) COMP-3. or
	YES	01 *DATA-NAME* PIC S9(p-s)V9(s) [USAGE IS] DISPLAY [SIGN IS] LEADING SEPARATE [CHARACTER].
DATE	NO	01 *DATA-NAME* PIC X(10). *
TIME	NO	01 *DATA-NAME* PIC X(8). *
DATETIME	NO	01 *DATA-NAME* PIC X(23). *
INTERVAL	NO	01 *DATA-NAME* PIC X(20). *

* Applies to default format specification only.

Table 2-5.
ALLBASE/SQL FIPS 127.1 Compliant Data Type Declarations for FORTRAN

SQL DATA TYPES	FIPS 127.1 Compliant?	FORTRAN DATA DECLARATIONS
CHAR(1)	NO	`CHARACTER` *DataName*
CHAR(n)	YES	`CHARACTER*n` *DataName*
VARCHAR(n)	NO	`CHARACTER*n` *DataName* *
SMALLINT	NO	`INTEGER*2` *DataName*
INTEGER	YES	`INTEGER` *DataName*
REAL	YES	`REAL` *DataName* `or`
	NO	`REAL*4` *DataName*
FLOAT(1..24)	NO	`REAL` *DataName* `or`
	NO	`REAL*4` *DataName*
FLOAT(1..53)	NO	`DOUBLE PRECISION` *DataName* `or`
	NO	`REAL*8` *DataName*
DOUBLE PRECISION	YES	`DOUBLE PRECISION` *DataName* `or`
	NO	`REAL*8` *DataName*
BINARY	NO	`CHARACTER` *DataName* `or`
	NO	`CHARACTER*n` *DataName*
VARBINARY	NO	`CHARACTER*n` *DataName*
DECIMAL	NO	`DOUBLE PRECISION` *DataName* `or`
	NO	`REAL*8` *DataName*
DATE	NO	`CHARACTER*10` *DataName*
TIME	NO	`CHARACTER*8` *DataName*
DATETIME	NO	`CHARACTER*23` *DataName*
INTERVAL	NO	`CHARACTER*20` *DataName*

* This declaration is for non-dynamic statements only.

Table 2-6.
ALLBASE/SQL FIPS 127.1 Compliant Data Type Declarations for Pascal

SQL DATA TYPES	FIPS 127.1 Compliant?	PASCAL TYPE DESCRIPTION
CHAR(1)	NO	*DataName* : char;
CHAR(n)	NO	*DataName* : array [1..*n*] of char; or
	YES	*DataName* : packed array [1..*n*] of char;
VARCHAR(*n*)	NO	*DataName* : string[*n*];
BINARY(1)	NO	*DataName* : char;
BINARY(n)	NO	*DataName* : array [1..*n*] of char;or
	NO	*DataName* : packed array [1..*n*] of char;
VARBINARY(*n*)	NO	*DataName* : string[*n*];
SMALLINT	NO	*DataName* : smallint;
INTEGER	YES	*DataName* : integer;
REAL	YES	*DataName* : real;
FLOAT	NO	*DataName* : longreal;
DECIMAL	NO	*DataName* : longreal;
DATE	NO	*DataName* : packed array[1..10] of char; *
TIME	NO	*DataName* : packed array[1..8] of char; *
DATETIME	NO	*DataName* : packed array[1..23] of char; *
INTERVAL	NO	*DataName* : packed array[1..20] of char; *

* Applies to default format specification only.

Host Variable Name Length Standards

The following table lists the standards compliant length for a host variable name in each supported language and the maximum ALLBASE/SQL limit.

Table 2-7. FIPS 127.1 Compliant Host Variable Name Lengths

Language	FIPS 127.1 Allowed Byte Length	ALLBASE/SQL Maximum Bytes
C	31 to infinity	20
COBOL	1 to 30	20
FORTRAN	1 to 32	20
Pascal	compiler limit: implementor defined	20

3

Comparing Static and Dynamic SQL

An ALLBASE/SQL application can contain both static and dynamic SQL statements. A **static statement** involves the preparation and storing of a section at preprocessing time and the execution of that stored section at run time. A **dynamic statement** involves the preparation and execution of a section at runtime. Some statements do not require a section, and they are also classified as dynamic.

Each type of statement has advantages and disadvantages as listed below:

- A static statement performs more efficiently than the equivalent dynamic statement.
- In order to execute a static statement, a program module (containing a stored section for the statement) must be installed in each DBEnvironment in which the statement is to run. A dynamically preprocessed statement is portable and can be run in any DBEnvironment without installing a program module.
- Dynamic statements may be more complex to code than are static statements.

Note that all objects referenced by a statement, whether it is static or dynamic, must be present in the DBEnvironment in which the application is running.

The following are dynamic statements:

- PREPARE
- DECLARE CURSOR when used with a *CommandName*
- DESCRIBE
- EXECUTE
- EXECUTE IMMEDIATE
- FETCH when used with the USING keyword
- any statement that does not require a stored section for execution

Any statement that is *not* coded as a parameter of one of the above statements and requires a stored section in order to execute is a static statement.

Comparing Static and Dynamic Applications

A **static application** is one that is preprocessed in full preprocessing mode and contains at least one static SQL statement. This means that a module is generated at preprocessing time, and this module must be installed in any DBEnvironment in which the application is running.

An application containing only dynamic SQL statements is termed a **dynamic application**. Such an application does not require a module. It can be preprocessed in either full preprocessing mode or in static conversion mode.

Suppose you are coding a user interface to the PurchDB.Parts database. While there is only one way of creating a static application, there are two ways of creating a dynamic application. They are as follows:

1. At coding time, by embedding only dynamic statements (as defined in the previous section), you are assured of a dynamic application. Such an application can be preprocessed in either full preprocessing mode or static conversion mode. With static conversion mode, a DBEnvironment name need not be specified on the command line.
2. At preprocessing time, by choosing static conversion mode, your resulting application is dynamic. If your source code contains static statements, they are converted to dynamic ones. Any dynamic statements are unchanged and remain dynamic.

Coding an Application that can be Either Static or Dynamic

Suppose you are coding a user interface to the PurchDB.Parts database. You know at coding time that you want the user to have the option of selecting information from each table in the database environment. You also know the exact format of each select statement. In addition, you want the user to be able to enter an ad hoc query, one that meets their particular needs at run time, for any of the tables in the database. The displayed menu might look something like the following:

```
                Menu to Select Parts Information
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

1. PARTS          3. ORDERS          5. VENDORS          7. REPORTS
2. INVENTORY      4. ORDER ITEMS     6. SUPPLY PRICE     8. AD HOC QUERY
```

You could code seven static SELECT statements and a dynamic select statement (for the ad hoc query). Depending on your requirements, you could preprocess this application in full preprocessing mode or static conversion mode. Full preprocessing could enhance performance when users most often choose one of the static SELECT statements. Static conversion processing would result in a portable application, ideal for use in a distributed database environment.

Converting a Static Application to a Dynamic Application

There are two ways of converting a static application to a dynamic application. One method is to change the source code to contain the required dynamic SQL statements. (The ALLBASE/SQL application programming guides provide complete information about coding with dynamic SQL statements.) The second method is to preprocess the application using static conversion mode. The following paragraphs discuss use of the second method with an application originally designed to use full preprocessing mode.

By choosing the DYNAMIC option at preprocessing time, you avoid having to make major changes to your existing source code. You also maintain the flexibility of preprocessing the source code to be either static or dynamic simply by changing the preprocessor command line options.

Unlike other preprocessing modes in which any DECLARE CURSOR statements are commented out, static preprocessing mode converts DECLARE CURSOR statements to executable code. Therefore, if your existing code contains such a statement in a non-executable portion of the code, you must move the statement to an executable portion of the application. In the C language, for example, you would declare the cursor within curly brackets following any data declaration statements within a given block. For Pascal, position the statement within a BEGIN/END block.

Another consideration is that host variables used in static statements (except for BULK FETCH and BULK SELECT statements) must adhere to the restrictions for dynamic parameters. See the chapter in this document titled "Using Parameter Substitution in Dynamic Statements" for further details.

Enhancing Performance

By default, ALLBASE/SQL does authorization checks on a dynamic query each time it is executed. By setting the authorize once per session flag to ON, you insure that authorization checks on dynamic queries are performed only the first time the query is executed during a given user session.

You can set this flag by means of the SQLUtil ALTDBE command described in the *ALLBASE/SQL Database Administration Guide*.

4

Using Parameter Substitution in Dynamic Statements

When your application uses dynamic processing, parameter substitution offers added flexibility and improved performance. Although you can use this technique in any dynamic processing application involving prepared sections, it could be most useful for applications where the same SQL statement type must be re-executed multiple times using a different set of parameter values each time.

A statement containing dynamic parameters must be dynamically preprocessed at run time by using the PREPARE statement. The dynamic section created can then be executed as many times as required within a given transaction with the option of assigning a different set of dynamic parameter values for each execution and without the overhead of preprocessing each time input values change.

The following paragraphs define dynamic parameters, discuss implementation methods, and provide examples. Topics include:

- Understanding Dynamic Parameters.
- Where to Use Dynamic Parameters.
- Programming with Dynamic Parameters.
- Using Default Data Types with Dynamic Parameters.

Understanding Dynamic Parameters

A **dynamic parameter** has the following characteristics:

- It is an input value to the database or an input or output parameter to or from a procedure.
- It is specified as a question mark within a string in a prepared statement in your application.
- Its datatype is determined based on its use in the prepared statement.
- You assign its value at run time via a host variable or a data buffer array.
- It is replaced by its assigned value when the OPEN or EXECUTE statement executes.

For example, the following statement specifying two dynamic parameters could be put into a string in your program:

```
UPDATE PurchDB.Parts SET SalesPrice = ? WHERE PartNumber = ?
```

The string itself can be used as a parameter of the PREPARE statement, or it can be assigned to a host variable that is a parameter of the PREPARE statement, as shown in the following sections.

Examples in C of Preparing a Statement with Dynamic Parameters

The following example uses a string as a parameter of the PREPARE statement:

```
EXEC SQL PREPARE CMD1 FROM 'INSERT INTO PurchDB.Parts (PartNumber,PartName)
          VALUES (?,?);';
```

In the following example, a host variable is used:

In the declare section, declare a character array host variable large enough to hold the string plus one byte for a delimiting ASCII 0:

```
EXEC SQL BEGIN DECLARE SECTION;
⋮
char DynamicCmdLine[81];
⋮
EXEC SQL END DECLARE SECTION;
⋮
```

Assign the string to the host variable:

```
strcpy(DynamicCmdLine,"INSERT INTO PurchDB.Parts (PartNumber, PartName)");
strcpy(tmpstr, " VALUES (?,?);");
strcat(DynamicCmdLine,tmpstr);
```

Prepare the statement:

```
EXEC SQL PREPARE CMD1 FROM :DynamicCmdLine;
```

Examples in COBOL of Preparing a Statement with Dynamic Parameters

The following example uses a string as a parameter of the PREPARE statement:

```
       EXEC SQL PREPARE CMD1 FROM "INSERT INTO PurchDB.Parts (PartNumbe
      -        r, PartName") VALUES (?,?);"
       END-EXEC.
```

In the following example, a host variable is used:

In the declare section, declare a host variable large enough to hold the string:

```
       EXEC SQL BEGIN DECLARE SECTION END-EXEC.
       ⋮
       01  DYNAMICCMD             PIC X(80).
       ⋮
       EXEC SQL END DECLARE SECTION END-EXEC.
       ⋮
```

Assign the string to the host variable:

```
       MOVE "INSERT INTO PurchDB.Parts (PartNumber, PartName)" TO TEMP1.
       MOVE " VALUES (?,?);" TO TEMP2.
```

```
STRING TEMP1 DELIMITED BY SIZE
       TEMP2 DELIMITED BY SIZE
       INTO DYNAMICCMD.
```

Prepare the statement:

```
EXEC SQL PREPARE CMD1 FROM :DYNAMICCMD END-EXEC.
```

Examples in FORTRAN of Preparing a Statement with Dynamic Parameters

The following example uses a string as a parameter of the PREPARE statement:

```
      EXEC SQL PREPARE CMD1 FROM 'INSERT INTO PurchDB.Parts
     1                (PartNumber,PartName) VALUES (?,?);'
```

In the following example, a host variable is used:

In the declare section, declare a host variable large enough to hold the string:

```
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*80     DynamicCommand
      EXEC SQL END DECLARE SECTION
      :
```

Assign the string to the host variable:

```
      DynamicCommand  = 'INSERT INTO PurchDB.Parts (PartNumber,PartName)
     1                   VALUES (?,?)'
```

Prepare the statement:

```
      EXEC SQL PREPARE CMD1 FROM :DynamicCommand
```

Examples in Pascal of Preparing a Statement with Dynamic Parameters

The following example uses a string as a parameter of the PREPARE statement:

```
EXEC SQL PREPARE CMD FROM 'INSERT INTO PurchDB.Parts (PartNumber,PartName)
      VALUES (?,?);';
```

In the following example, a host variable is used:

In the declare section, declare a host variable large enough to hold the string:

```
EXEC SQL BEGIN DECLARE SECTION;
⋮
DynamicCmdLine:string[80];
⋮
EXEC SQL END DECLARE SECTION;
⋮
```

Assign the string to the host variable:

```
DynamicCmdLine := 'INSERT INTO PurchDB.Parts (PartNumber, PartName)'+
                  ' VALUES (?,?);';
```

Prepare the statement:

```
EXEC SQL PREPARE CMD1 FROM :DynamicCmdLine;
```

Where to Use Dynamic Parameters

Use a dynamic parameter as you would a constant in an expression in the DML operations listed below:

Table 4-1. Where to Use Dynamic Parameters

DML Operation	Clause
INSERT	VALUES WHERE, HAVING [1]
BULK INSERT	VALUES
UPDATE	SET WHERE
DELETE	WHERE
SELECT	WHERE, HAVING
EXECUTE	INPUT, OUTPUT

[1] In the case of an INSERT statement, dynamic parameters can be used in the WHERE or HAVING clause of a Type 2 INSERT.

See the *ALLBASE/SQL Reference Manual* chapter, "Expressions," for more information regarding constants in expressions.

Restrictions

The examples below are shown to clarify the conditions under which dynamic parameters *cannot* be used. The following locations are *not* valid:

- In any select list.
- In any statement that is not dynamically preprocessed with the PREPARE statement.
- As both operands of a single, arithmetic operator or comparison operator. The following example is not valid:

```
SELECT * FROM PurchDB.Parts
        WHERE SalesPrice > (? * ?)
```

- As the operand of a minus sign or a null predicate. The following examples are not valid:

```
INSERT INTO PurchDB.OrderItems VALUES (-?,?,?)

UPDATE PurchDB.Parts
   SET SalesPrice = ?
 WHERE ItemNo = ?
    OR (ItemNo IS NULL AND ? IS NULL)
```

- As the entire argument of an aggregate function. The following example is not valid:

```
SELECT * FROM PurchDB.Orders
     GROUP BY PartNumber
       HAVING MAX(?) > 543
```

- As the parameter of a NULL predicate. For example, the following is not valid:

```
SELECT * FROM PurchDB.Parts
        WHERE ItemNo = ?
           OR (ItemNo IS NULL AND ? IS NULL)
```

- As both the first and second expressions of a BETWEEN predicate. The following example is not valid:

```
SELECT * FROM PurchDB.Orders
        WHERE ? BETWEEN ? AND 100
```

- As both the expression and the first parameter of an IN predicate. The following example is not valid:

```
SELECT * FROM PurchDB.Orders
        WHERE ? IN (?, 4000, 5000, 6000)
```

Programming with Dynamic Parameters

Depending on the purpose of your application, there is a broad spectrum of scenarios in which dynamic parameters could be useful. You might know almost all the elements of a statement at coding time, including the statement type and what dynamic parameters are required. At the opposite extreme, a program might be required to handle a completely unknown SQL statement containing dynamic parameters. Generally speaking, the less you know about a dynamic statement at coding time, the more coding you must do to verify the statement's content at run time.

The two basic methods of assigning dynamic parameter values involve use of either host variables or ALLBASE/SQL data structures and a data buffer. To use host variables, you must at least know the exact format of your SQL statement, although you need not know the specific data values of dynamic parameters. To use data structures and a data buffer, you do not need to know the exact format of your SQL statement.

Table 4-2. Dynamic Parameter Functionality by Programming Language

Language	Dynamic Parameter Data Assignment via Host Variables	Dynamic Parameter Data Assignment via a Data Buffer	Dynamic Parameters in a BULK INSERT Statement
C	yes	yes	yes
COBOL	yes	no	yes (with host variables)
FORTRAN	yes	no	no
Pascal	yes	yes	yes

Host variables are available for C, COBOL, FORTRAN, and Pascal applications. Data buffers are available for C and Pascal applications only. In addition, dynamic parameters within a BULK INSERT statement require special syntax and are discussed separately. BULK INSERT functionality is available for C, COBOL, and Pascal applications. The following subsections discuss each basic coding method:

- Using Host Variables to Process Dynamic Parameters.
- Using Data Structures and a Data Buffer to Process Dynamic Parameters
- Using a BULK INSERT Statement with Dynamic Parameters.

Using Host Variables to Process Dynamic Parameters

When you know at coding time the data type and format of each dynamic parameter in a dynamic statement, you have the choice of using either a host variable or a data buffer to provide dynamic parameter input at run time. This section details the use of host variables with non-bulk statements. (The next section discusses the data buffer technique.) The functionality described in this section is available for C, COBOL, FORTRAN, and Pascal programs.

Suppose you are coding an interactive user application. It involves mapping a user's menu choice to a partially known statement, then prompting for and accepting dynamic parameter values for data whose format is known at coding time. The following pseudocode illustrates this scenario.

⋮

Accept a variable indicating which of a set of statements the user has chosen.

Prepare the dynamic command for this statement:

```
PREPARE CMD FROM 'UPDATE PurchDB.Parts SET SalesPrice = ? WHERE PartNumber = ?;'
```

Prompt the user for values for the SalesPrice and PartNumber columns.

Execute the dynamic command using host variables to provide dynamic parameter values:

```
EXECUTE CMD USING :SalesPrice, :PartNumber;
```

You could now loop back to prompt the user for additional values for SalesPrice and PartNumber. Note that the dynamic command does not have to be prepared again.

Using Data Structures and a Data Buffer to Process Dynamic Parameters

If at coding time you don't know the data types of all dynamic parameters in the prepared statement, you must use two ALLBASE/SQL data structures and a data buffer to obtain the default data types and pass dynamic parameter input to the database. These data structures are the same as those used for dynamic output processing:

- sqlda_type data structure.
- sqlfmts_type data structure.
- data buffer.

The following discussion points out how to use these structures for input data. Here the term **input data** means dynamic parameter data, and **output data** means select list data. When a prepared statement is described for *both* input *and* output data, you must define one set of data structures for input data and another set for output data. Refer to the chapter "Using Dynamic Operations" in the *ALLBASE/SQL C Application Programming Guide* or the *ALLBASE/SQL Pascal Application Programming Guide* for a detailed description and example of how to use dynamic data structures for output.

Using the SQLDA for Input

To use an SQLDA structure for input, you prepare the dynamic command, then use the INPUT option with the DESCRIBE statement:

```
DESCRIBE INPUT DynamicCommand INTO SQLDA
```

In place of SQLDA, you could name any data structure of type *sqlda_type*.

When the DESCRIBE statement executes, whether for input data or output data, the values in a given format array *must* be consistent with the values in its related data buffer. Refer to Table 4-6 and Table 4-7 for a detailed description of the format array.

One difference between the use of the SQLDA for input data versus output data involves the Sqln and Sqld fields. Sqln is set by your program prior to issuing the DESCRIBE statement and represents the maximum number of 48 byte format array records allowed by the program. When using the DESCRIBE OUTPUT specification, you tell ALLBASE/SQL to load each format record with information for each select list item in the currently prepared statement (if it is a query). Using the DESCRIBE INPUT specification indicates that you want ALLBASE/SQL to load each format array record with information for each dynamic parameter in the currently prepared statement. Therefore, following execution of the DESCRIBE statement, Sqld represents either the number of select list items output or the number of dynamic parameters input.

When you describe dynamic parameters for input, the Sqlindlen field in the format array *always* equals two, even if it relates to a table column that does not allow nulls. Therefore, you must allow two bytes for this field in the corresponding input data buffer. By contrast, when you describe output for a column that was defined as not null, the Sqlindlen field is set to 0 to indicate no bytes are allocated in the corresponding data buffer for information relating to null values.

Caution ALLBASE/SQL reads the data buffer based on its related format array. When you have described input, be sure the data you load into the data buffer corresponds to the information in its related format array.

Using the Data Buffer for Input

When you are describing data for input, it is your program's responsibility to load the data buffer with input values for each dynamic parameter based on information in the related format array. (This is unlike describing data for output, where ALLBASE/SQL loads the specified data buffer when data is fetched.) Following is a list of possible coding steps:

1. Define any host variables to be used to pass a command string to ALLBASE/SQL via the PREPARE and DESCRIBE statements.
2. Define any necessary sqlda_type structures (also called descriptor areas) for holding information about a given command string. Sqlda_type structures are used to communicate information regarding a specific SQL statement between this program and the database to which it is connected. Information is transferred when the DESCRIBE statement executes and when the FETCH statement executes.

   Remember that the INCLUDE SQLDA statement generates one sqlda_type data structure named sqlda. So, if you need more than one such structure, you must code a declaration for each.

If you know nothing about an SQL statement until run time, define an sqlda_type structure for output to determine if the statement is a query or not. If it is a query, ALLBASE/SQL loads the related sqlformat_type structure with the format of the query result (one 48 byte element per select list item). You must also define an sqlda_type structure for input in case the statement contains dynamic parameters. In this case, ALLBASE/SQL loads the related sqlformat_type structure with the format of the dynamic parameters (one 48 byte element per dynamic parameter).

3. Define an sqlformat_type structure (also called a format array) for each required data buffer.

   Information is transferred to the format array when the DESCRIBE statement executes.

4. Define any necessary data buffers.

   Each data buffer must correspond to an sqlda_type structure and an sqlformat_type structure.

   If the statement is a query, your program needs a data buffer to hold the query results generated by the FETCH statement. If the statement contains dynamic parameters, your program needs a data buffer into which it loads the values of those dynamic parameters. The dynamic parameter values are transmitted to the database by means of an OPEN or an EXECUTE statement.

5. Use the PREPARE statement to preprocess the dynamic statement.

6. Set the appropriate sqlda_type fields. See Table 4-4 and Table 4-5.

   Remember, when you describe input for a non-bulk statement (a statement that processes just one row), sqlnrow must always be equal to one prior to issuing the OPEN or EXECUTE statement.

7. Use the DESCRIBE statement (with the optional OUTPUT specification) to determine the statement type and its format if it is a query. Information goes to the specified sqlda_type and sqlformat_type data structures.

   You must use DESCRIBE OUTPUT if, at coding time, the composition of your prepared statement is completely unknown or if you know it is a query but you do not know its exact format and content.

8. Use the DESCRIBE statement with the INPUT specification to determine the number of dynamic parameters in the prepared statement and the default data type and format of each. Your application obtains this information via the specified sqlda_type and sqlformat_type data structures. (The "Using Default Data Types with Dynamic Parameters" section later in this chapter contains detailed information about default data types and default data formats for dynamic parameters.)

9. Load the input data buffer with dynamic parameter values based on information provided by the DESCRIBE INPUT statement.

10. If the prepared statement is a query, use a DECLARE statement to associate it with a cursor.

    Use an OPEN statement to put qualifying rows of the query into the data buffer you have defined for output. Specify the USING DESCRIPTOR clause of the OPEN statement to pass in dynamic parameter values.

    In a loop, use a FETCH USING DESCRIPTOR statement to process each row.

11. Close the cursor and commit work.
12. If the prepared statement is not a query, use the EXECUTE statement with the USING clause to pass in dynamic parameter values.

Table 4-3. ALLBASE/SQL Data Type Byte Alignment

Format Array sqltype Field	Data Type	Series 700 and 800 Byte Alignment	Series 300 and 400 Byte Alignment
0	INTEGER	4	2
0	SMALLINT	2	2
1	BINARY	1	1
2	CHAR	1	1
3	VARCHAR	4	2
4	DOUBLE PRECISION	8	2
4	FLOAT (4 bytes)	4	2
4	FLOAT (8 bytes)	8	2
4	REAL	4	2
5	DECIMAL	4	2
5	NUMERIC	4	2
6	TID	4	2
10	DATE	1	1
11	TIME	1	1
12	DATETIME	1	1
13	INTERVAL	1	1
14	VARBINARY	4	2
15	LONG BINARY	1	1
16	LONG VARBINARY	1	1

Table 4-4. Setting SQLDA Fields for Output and for Input in C

Field Name	Field Description	C Data Type	You Set Before DESCRIBE or ADVANCE	You Set Before OPEN or EXECUTE USING INPUT	You Set Before FETCH or EXECUTE USING OUTPUT	ALLBASE/SQL Sets at DESCRIBE or ADVANCE	ALLBASE/SQL Sets at FETCH or EXECUTE USING OUTPUT
sqldaid	reserved	char[8]					
sqlmproc	number of multiple row result sets inside a procedure	short				IOR	
sqloparm	number of output dynamic parameters in a dynamically prepared EXECUTE PROCEDURE statement	short				O	
sqln	number of format array elements (for output, one record per select list item to a maximum of 1024; for input, one record per dynamic parameter to a maximum of 255)	int	IOR				
sqld	for output, number of columns in query result (0 if non-query or EXECUTE PROCEDURE); for input, number of input dynamic parameters in the prepared statement	int				IOR	
sqlfmtarr	address of format array	int	IOR			IOR [2]	
sqlnrow	number of rows in the data buffer [1]	int		I[1]	O[1]		
sqlrrow	number of rows put into the data buffer	int					O
sqlrowlen	number of bytes in each row	int				IOR	
sqlbuflen	number of bytes in the data buffer	int		I	O		
sqlrowbuf	address of data buffer	int		I	O		

I Used for input.

O Used for output.

R Used for DESCRIBE RESULT and ADVANCE.

[1] When you describe for output, use sqlnrow to specify the number of rows to fetch into the data buffer. When you describe for input, use sqlnrow to specify the number of rows you have loaded into the data buffer (Always set to one for a non-bulk statement.).

[2] Data is loaded into the format array when a DESCRIBE or ADVANCE statement executes.

Table 4-5. Setting SQLDA Fields for Output and for Input in Pascal

Field Name	Field Description	C Data Type	You Set Before DESCRIBE or ADVANCE	You Set Before OPEN or EXECUTE USING INPUT	You Set Before FETCH or EXECUTE USING OUTPUT	ALLBASE/SQL Sets at DESCRIBE or ADVANCE	ALLBASE/SQL Sets at FETCH or EXECUTE USING OUTPUT
sqldaid	reserved	char[8]					
sqlmproc	number of multiple row result sets inside a procedure	short				IOR	
sqloparm	number of output dynamic parameters in a dynamically prepared EXECUTE PROCEDURE statement	smallint				O	
sqln	number of format array elements (for output, one record per select list item to a maximum of 1024; for input, one record per dynamic parameter to a maximum of 255)	integer	IOR				
sqld	for output, number of columns in query result (0 if non-query or EXECUTE PROCEDURE); for input, number of input dynamic parameters in the prepared statement	integer				IOR	
sqlfmtarr	address of format array	integer	IOR			IOR [2]	
sqlnrow	number of rows in the data buffer [1]	integer		I[1]	O[1]		
sqlrrow	number of rows put into the data buffer	integer					O
sqlrowlen	number of bytes in each row	integer				IOR	
sqlbuflen	number of bytes in the data buffer	integer		I	O		
sqlrowbuf	address of data buffer	integer		I	O		

I Used for input.

O Used for output.

R Used for DESCRIBE RESULT and ADVANCE.

[1] When you describe for output, use sqlnrow to specify the number of rows to fetch into the data buffer. When you describe for input, use sqlnrow to specify the number of rows you have loaded into the data buffer (Always set to one for a non-bulk statement.).

[2] Data is loaded into the format array when a DESCRIBE or ADVANCE statement executes.

Table 4-6. Fields in a Format Array Record in C

Field Name	Meaning of Field	C Data Type
sqlnty	reserved; always set to 111	short
sqltype	data type of column: 0 = SMALLINT or INTEGER 1 = BINARY 2 = CHAR* 3 = VARCHAR* 4 = FLOAT 5 = DECIMAL 8 = NATIVE CHAR * 9 = NATIVE VARCHAR * 10 = DATE* 11 = TIME* 12 = DATETIME* 13 = INTERVAL* 14 = VARBINARY 15 = LONG BINARY 16 = LONG VARBINARY 19 = case insensitive CHAR* 20 = case insensitive VARCHAR* 21 = case insensitive NATIVE CHAR* 22 = case insensitive NATIVE VARCHAR* * Native CHAR or VARCHAR is what SQLCore uses internally when a CHAR or VARCHAR column is defined with a LANG = ColumnLanguageName clause. They possess the same characteristics as the related types CHAR and VARCHAR, except that data stored in native columns will be sorted, compared, or truncated using local language rules. Native, character, and Date/Time types are compatible with regular character types.	short
sqlprec	precision of DECIMAL data	short
sqlscale	scale of DECIMAL data	short
sqltotallen	byte sum of sqlvallen, sqlindlen, indicator alignment bytes, and next data value alignment bytes	int
sqlvallen	number of bytes in data value, including a 4-byte prefix containing actual length of VARCHAR data	int
sqlindlen	number of bytes null indicator occupies in the data buffer for output: 0 bytes: column defined NOT NULL 2 bytes: column allows null values for input: always 2 bytes	int
sqlvof	byte offset of value from the beginning of a row	int

Table 4-6. Fields in a Format Array Record in C (continued)

Field Name	Meaning of Field	C Data Type
sqlnof	byte offset of null indicator from the beginning of a row, dependent on the value of sqlindlen	int
sqlname	defined name of column or, for computed expression, EXPR	char[20]

Table 4-7. Fields in a Format Array Record in Pascal

Field Name	Meaning of Field	Pascal Data Type
sqlnty	reserved; always set to 111	SmallInt

sqltype	data type of column:
	`0 = SMALLINT or INTEGER`
	`1 = BINARY`
	`2 = CHAR*`
	`3 = VARCHAR*`
	`4 = FLOAT`
	`5 = DECIMAL`
	`8 = NATIVE CHAR *`
	`9 = NATIVE VARCHAR *`
	`10 = DATE*`
	`11 = TIME*`
	`12 = DATETIME*`
	`13 = INTERVAL*`
	`14 = VARBINARY`
	`15 = LONG BINARY`
	`16 = LONG VARBINARY`
	`19 = case insensitive CHAR*`
	`20 = case insensitive VARCHAR*`
	`21 = case insensitive NATIVE CHAR*`
	`22 = case insensitive NATIVE VARCHAR*`
	* Native CHAR or VARCHAR is what SQLCore uses internally when a CHAR or VARCHAR column is defined with a LANG = ColumnLanguageName clause. They possess the same characteristics as the related types CHAR and VARCHAR, except that data stored in native columns will be sorted, compared, or truncated using local language rules. Native, character, and Date/Time types are compatible with regular character types.
		SmallInt

sqlprec	precision of DECIMAL data	SmallInt
sqlscale	scale of DECIMAL data	SmallInt
sqltotallen	byte sum of sqlvallen, sqlindlen, indicator alignment bytes, and next data value alignment bytes	Integer
sqlvallen	number of bytes in data value, including a 4-byte prefix containing actual length of VARCHAR data	Integer

sqlindlen	number of bytes null indicator occupies in the data buffer:
	`for output:`
	`0 bytes: column defined NOT NULL`
	`2 bytes: column allows null values`
	`for input: always 2 bytes`
		Integer

sqlvof	byte offset of value from the beginning of a row	Integer

Table 4-7. Fields in a Format Array Record in Pascal (continued)

Field Name	Meaning of Field	Pascal Data Type
sqlnof	byte offset of null indicator from the beginning of a row, dependent on the value of sqlindlen	Integer
sqlname	defined name of column or, for computed expression, EXPR	Packed Array [1..20] of char

Example in C Using Output and Input Data Buffers

Suppose you have designed an application that builds a SELECT statement. A user can enter any valid DBEnvironment name, table name, column name, and a column value to be used as a filter in the WHERE clause. The application builds the appropriate query and displays the query result.

Your application prepares this SELECT statement and describes it for output. It also describes the statement for input so that ALLBASE/SQL can determine a default data type and format for the user entered column value. Note that the "Using Default Data Types with Dynamic Parameters" section later in this chapter contains detailed information about default data types and default data formats for dynamic parameters.

The following C pseudocode outlines the above scenario with emphasis on ALLBASE/SQL programming for dynamic parameter substitution. The functionality is available for the C and Pascal languages.

```
⋮

#define  NbrFmtRecords    255
#define  MaxDataBuff     2500
#define  MaxColSize      3996
#define  MaxName           20
#define  MaxStr           132
#define  SQLINT             0
#define  SQLCHAR            2
#define  OK                 0
#define  TRUE               1
#define  FALSE              0
```

The ConvertType union structure is used to convert the SearchValue before it is assigned to DataBufferIn.

```
typedef union ct {
  char           CharData[MaxColSize];
  char           VarCharData[MaxColSize];
  int            IntegerData;
  short          SmallIntData;
  float          FloatData;
  double         DecimalData;
  } ConvertType;
```

Host variables are declared as follows:
DynamicCommand contains the dynamic SELECT statement. SQLMessage holds messages returned by the SQLEXPLAIN statement. SearchValue, entered by the user, is the value searched for by the SELECT statement.

```
EXEC SQL BEGIN DECLARE SECTION;
   char      DynamicCommand[1023];
   char      SQLMessage[133];
```

```
   char      SearchValue[1024];
EXEC SQL END DECLARE SECTION;
```

Declare the SQL communications area.

```
EXEC SQL INCLUDE SQLCA;
```

The sqldain record contains information about the sqlfmtsin format array and the DataBufferIn variable.

```
sqlda_type sqldain;
```

The sqldaout record contains information about the sqlfmtsout format array and the DataBufferOut variable.

```
sqlda_type sqldaout;
```

The sqlfmtsin format array describes the dynamic parameters in the WHERE clause of the SELECT statement. Each record in the array describes one dynamic parameter. Since this program specifies a single dynamic parameter, only the first record in the array, sqlfmtsin[0], will be checked.

```
sqlformat_type sqlfmtsin[NbrFmtRecords];
```

The sqlfmtsout format array describes the columns in the select list of the SELECT statement. Each record in the array describes one column.

```
sqlformat_type sqlfmtsout[NbrFmtRecords];
```

DBEName contains the user specified database environment name.

```
char DBEName[MaxName];
```

TableName contains the user specified table name of the SELECT statment.

```
char TableName[MaxName];
```

ColName contains the user specified column name of the SELECT statment.

```
char ColName[MaxName];
```

DataBufferIn contains the value of the dynamic parameter, in this case the value of the column in the WHERE clause of the SELECT statement.

```
char DataBufferIn[MaxDataBuff];
```

DataBufferOut contains the row values retrieved by the SELECT statement.

```
char DataBufferOut[MaxDataBuff];

⋮

/*****************************************************************/
main()
/*****************************************************************/
{
```

Prompt the user for the database environment used in the CONNECT statement.

```
sprintf (DBEName,"");
sprintf (DynamicCommand,"");
Prompt ("DBEnvironment name",DBEName);
```

After prompting the user for the table name and the column name, move the SELECT statement into the DynamicCommand variable. The dynamic parameter, represented by the question mark, is not specified until after the PREPARE and DESCRIBE statements.

```
if (ConnectDBE()) {
   Prompt ("Table Name",TableName);
   while (strlen(TableName)!=0) {
       Prompt ("Column Name",ColName);
       sprintf (DynamicCommand,"SELECT * FROM %s WHERE %s = ?;",
               TableName, ColName);
       Prepare();
       Prompt ("Table Name",TableName);
      } /* end while */
   ReleaseDBE();
   } /* end if */
else
   printf("\nError: Cannot Connect to %s",DBEName);

}  /* End of Main Program */

/*****************************************************************/
int Prepare()
/*****************************************************************/
{
```

Before the PREPARE statement, the input and output descriptor fields must be set up.

The sqldain.sqln variable is assigned the number of records in the sqlfmtsin array and the sqldain.sqlfmtarr variable is assigned the address of the

sqlfmtsin array.

```
sqldain.sqln      = NbrFmtRecords;
sqldain.sqlfmtarr = sqlfmtsin;
```

The sqldaout.sqln variable is assigned the number of records in the sqlfmtsout array and the sqldaout.sqlfmtarr variable is assigned the address of the sqlfmtsout array.

```
sqldaout.sqln      = NbrFmtRecords;
sqldaout.sqlfmtarr = sqlfmtsout;

if (BeginTransaction()) {

   Prepare the dynamic SELECT statement.  At this point the value of the
   dynamic parameter is still undefined.

   EXEC SQL PREPARE CMD1 FROM :DynamicCommand;

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

      The DESCRIBE statement gets information about the statement that was
      dynamically preprocessed by the PREPARE statement.

      Here dynamic parameter information is obtained:

      EXEC SQL DESCRIBE INPUT CMD1 INTO sqldain;
      if (sqlca.sqlcode != OK) {
          SQLStatusCheck();
          EndTransaction();
          }
      else

      Here query result information is obtained:

      EXEC SQL DESCRIBE CMD1 INTO sqldaout;
      if (sqlca.sqlcode != OK) {
          SQLStatusCheck();
          EndTransaction();
          }
      else

         Fetch();
```

```
      }

  }  /* End if BeginTransaction */

}  /* End of Prepare function */

/*********************************************************************/
int Fetch()
/*********************************************************************/
{

short       i;
ConvertType ConvertedSearch;
```

Declare the cursor for the SELECT statement.

```
EXEC SQL DECLARE CURSOR1 CURSOR FOR CMD1;
```

Prompt the user for the search value, which will be assigned to the dynamic parameter in the WHERE clause of the SELECT statement. The sqlfmtsin[0].sqlname variable contains the column name in the WHERE clause of the SELECT statement.

```
Prompt (sqlfmtsin[0].sqlname,SearchValue);
```

Set up the input descriptor fields of the sqldain record before opening the cursor.

The sqldain.sqlnrow variable is assigned the number of rows in DataBufferIn, that is, the number of dynamic parameters specified.

```
sqldain.sqlnrow   = 1;
```

The sqldain.sqlbuflen variable is assigned the number of bytes in DataBufferIn.

```
sqldain.sqlbuflen = MaxDataBuff;
```

The sqldain.sqlrowbuf variable is assigned the address of DataBufferIn.

```
sqldain.sqlrowbuf = (int) DataBufferIn;
```

Since the search value entered by the user is a character string, it must be converted to the format of the column in the WHERE clause of the dynamic SELECT statement. The SearchValue is first assigned to the ConvertedSearch record, and then assigned to DataBufferIn.

Check the value of sqlfmtsin[0].sqltype to determine the data type of the column in the WHERE clause.

```
if (sqlfmtsin[0].sqltype == SQLINT) {
   INT or SMALLINT columns generate the same data type value in sqltype, but
   must be distinguished because they have different lengths.  If sqlvallen
   is equal to the size of an integer variable, then the data type is INT.
   if (sqlfmtsin[0].sqlvallen == sizeof(ConvertedSearch.IntegerData))
      SQL INT data type.
      ConvertedSearch.IntegerData =  atoi(SearchValue);
   else
      Otherwise, the column data type is SMALLINT.
      ConvertedSearch.SmallIntData =  atoi(SearchValue);
   }
else if (sqlfmtsin[0].sqltype == SQLCHAR) {
   Otherwise, the column data type is CHAR.
   for (i = 0;i < strlen(SearchValue); i++)
       ConvertedSearch.CharData[i] = SearchValue[i];
   for (i = strlen(SearchValue); i < sqlfmtsin[0].sqlvallen; i++)
       ConvertedSearch.CharData[i] = ' ';
   }
else
   printf ("Error: Conversion routine unavailable for that data type.\n");
```

Move the ConvertedSearch data to DataBufferIn.

```
StrMove (sqlfmtsin[0].sqlvallen,ConvertedSearch.CharData,0,DataBufferIn,0);
```

Assign zero to the two-byte, null field value that must follow the search value data in DataBufferIn.

```
ConvertedSearch.SmallIntData = 0;
StrMove (2,ConvertedSearch.CharData,0,DataBufferIn,sqlfmtsin[0].sqlvallen);
```

Open the cursor, using the input description record sqldain.

```
EXEC SQL OPEN CURSOR1 USING SQL DESCRIPTOR sqldain;

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

Set up the output descriptor fields of the sqldaout record before performing the fetch.

The sqldaout.sqlbuflen variable is assigned the number of bytes in DataBufferOut.

```
sqldaout.sqlbuflen = MaxDataBuff;
```

The sqldaout.sqlnrow variable is assigned the number of rows in DataBufferOut, that is, the number of rows to fetch.

```
sqldaout.sqlnrow   = ((sqldaout.sqlbuflen) / (sqldaout.sqlrowlen));
```

The sqldaout.sqlrowbuf variable is assigned the address of DataBufferOut.

```
sqldaout.sqlrowbuf = (int) DataBufferOut;
```

Fetch rows into DataBufferOut until no more rows are found.

```
do {
   EXEC SQL FETCH CURSOR1 USING SQL DESCRIPTOR sqldaout;

   if (sqlca.sqlcode == 100)
      printf ("Warning: No more rows qualify for this operation\n");
   else if (sqlca.sqlcode != 0)
      SQLStatusCheck();
   else
      The DisplaySelect function parses DataBufferOut and displays the data.
      See program cex10a in the ALLBASE/SQL C Application Programming Guide
      for a full listing of the DisplaySelect function.
      DisplaySelect();

  } while (sqlca.sqlcode == 0);
```

Close the cursor and end the transaction.

```
EXEC SQL CLOSE CURSOR1;

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

EndTransaction();

}  /* End of Fetch function */

/*****************************************************************/
int StrMove (n,source,subs,dest,subd)
/*****************************************************************/

int n, subs, subd;
char source[], dest[];

{
int i = 1;
```

Move n number of bytes from source, starting at source[subs] to dest, starting at dest[subd].

```
while (i++ <= n)
   dest[subd++] = source[subs++];

}  /* End of StrMove function */

/*********************************************************************/
int Prompt (displaystr,inputstr)
/*********************************************************************/

char *displaystr, *inputstr;

{

printf("Enter %s: ",displaystr);
gets(inputstr);

} /* End of Prompt function */

⋮
```

You could enhance the above pseudocode by coding an application for the following scenario. After the display, offer the user these choices:

- Enter another value for the same column.
- Enter another table name, column name, and column value for the same DBEnvironment.
- Exit the application.

Since the column value is passed by means of a dynamic parameter, if the user chooses to enter another column value for the same column, you can improve performance by reusing the already prepared stored section for the given SELECT statement.

Each time the user enters another value for the same SELECT statement, your application does the following:

- Loads the new value into the input data buffer.
- Opens the dynamic cursor.
- Fetches all qualifying rows.
- Closes the cursor.

Using a BULK INSERT Statement with Dynamic Parameters

Suppose you are writing an application that inserts multiple rows of data. At coding time, you know the format of the BULK INSERT statement, and you know which parameters in the statement will differ for each row (the dynamic parameters).

The application must run in many DBEnvironments, and you want it to achieve maximum performance. For portability, you decide on dynamic statements. For maximum performance, you decide to use parameter substitution and the BULK INSERT statement. To minimize your coding time, you use host variables, rather than a data buffer.

The following pseudocode examples illustrate this scenario for C, COBOL, and Pascal applications.

Note When host variables are used, EXECUTE statement syntax differs for a BULK INSERT statement and an INSERT statement.

Example in C Using a BULK INSERT

```
   ⋮
   boolean   OrdersOk;
   boolean   ConnectDBE();

⋮
    sqlca_type sqlca;              /* SQL communication area.              */
```

Define a host variable array to hold dynamic parameter values. Be sure each host variable data type matches (or is compatible with) its ALLBASE/SQL default data type:

```
    EXEC SQL BEGIN DECLARE SECTION;

    struct   {
        int      NewOrderNumber;
        int      NewVendorNumber;
        sqlind   NewVendorNumberInd;
        char     NewOrderDate[11];  /*   Add a byte for end of char array.   */
        sqlind   NewOrderDateInd;
      } NewOrders[25];
```

If the dynamic parameter represents data for a column that can contain nulls, and it is possible that input data will contain null values, be sure to define a null indicator host variable immediately following the related host variable.

If you are using other than the default values for the starting index and number of rows in the array, define host variables for these as well:

```
    short   StartIndex;
    short   NumberOfRows;          /* Maximum possible rows to bulk insert.  */

    int     OrderNumber;           /* Host variables for input data.         */
    int     VendorNumber;
    sqlind  VendorNumberInd;
    char    OrderDate[11];
    sqlind  OrderDateInd;

    ⋮

    char    SQLMessage[133];       /* Add a byte for end of char array.      */

    EXEC SQL END DECLARE SECTION;
```

```
main() {                              /*Specify main functions.                */

if (ConnectDBE()) {                   /* If the application is successfully    */
                                      /* connected to a DBEnvironment, proceed. */
   OrdersOk = TRUE;

   BeginTransaction();
   PrepareIt();
   CreateOrders();
   InsertNew();

if (OrdersOk)                         /* If there were no errors in processing, */
   CommitWork();                      /* data is committed to the database.     */

TerminateProgram();

       }                              /* End if.                                */
}                                     /* End of main program.                   */
```

⋮

Use the PREPARE statement to preprocess the dynamic statement, in this case, from a string:

```
int PrepareIt(){

EXEC SQL PREPARE CMD from 'BULK INSERT INTO PurchDB.Orders VALUES (?,?,?);';


switch (sqlca.sqlcode){               /* Check for processing errors.           */
       case     OK:      break;

           default:      SQLStatusCheck();
                         RollBackWork();
                         OrdersOk = FALSE;

       }                              /* End switch.                            */
}                                     /* End function PrepareIt.                */
```

Load up to 25 rows of new orders for the BULK INSERT. This data could originate from an interactive user or from a file:

```
int CreateOrders()
{

int i = 0;                            /* Define and initialize an index to move */
                                      /* through array elements.                */
NumberOfRows = 25;
StartIndex   = 1;
```

Count rows as they are loaded into the NewOrders array up to a maximum of 25:

```
for (i = 0; i <= NumberOfRows; i++){
```

Read a file record or accept a row of data from the user into the appropriate host variables.

Load host variable data into the bulk insert array.

```
NewOrders[i].NewOrderNumber = OrderNumber;
NewOrders[i].NewVendorNumber = VendorNumber;
NewOrders[i].NewVendorNumberInd = VendorNumberInd;
strcpy (NewOrders[i].NewOrderDate,OrderDate);
NewOrders[i].NewOrderDateInd = OrderDateInd;

    }                           /* End for.                              */
}                               /* End of function CreateOrders.         */
```

Execute the prepared CMD command specifying the array where data for the BULK INSERT is located:

```
int InsertNew(){

EXEC SQL EXECUTE CMD USING :NewOrders, :StartIndex, :NumberOfRows;

switch (sqlca.sqlcode){            /* Check for processing errors.       */
       case     OK:      break;

           default:      SQLStatusCheck();
                         RollBackWork();
                         OrdersOk = FALSE;
       }                            /* End switch.                       */
}                                   /* End of function InsertNew.        */
```

Example in COBOL Using a BULK INSERT

```
       ⋮

       WORKING-STORAGE SECTION.

      * SQL communication area. *
       EXEC SQL INCLUDE SQLCA END-EXEC.

      * Host variables for input data. *
       EXEC SQL BEGIN DECLARE SECTION END-EXEC.
       01  ORDERNUMBER             PIC S9(9) COMP.
       01  VENDORNUMBER            PIC S9(9) COMP.
       01  VENDORNUMBERIND         SQLIND.
       01  ORDERDATE               PIC X(8).
       01  ORDERDATEIND            SQLIND.
```

Define a host variable array to hold dynamic parameter values. Be sure each host variable data type matches (or is compatible with) its ALLBASE/SQL default data type:

```
       01  NEWORDERS.
         05  EACH-ROW OCCURS 25 TIMES.
           10  NEWORDERNUMBER           PIC S9(9) COMP.
           10  NEWVENDORNUMBER          PIC S9(9) COMP.
           10  NEWVENDORNUMBERIND       SQLIND.
           10  NEWORDERDATE             PIC X(8).
           10  NEWORDERDATEIND          SQLIND.
```

If the dynamic parameter represents data for a column that can contain nulls, and it is possible that input data will contain null values, be sure to define a null indicator host variable immediately following the related host variable.

If you are using other than the default values for the starting index and number of rows in the array, define host variables for these as well:

```
       01  STARTINDEX              PIC S9(4) COMP.

      * Maximum possible rows to bulk insert. *
       01  NUMBEROFROWS            PIC S9(4) COMP.

       01  SQLMESSAGE              PIC X(132).
       EXEC SQL END DECLARE SECTION END-EXEC.

       PROCEDURE DIVISION.

       A100-MAIN.
```

```
        PERFORM A200-CONNECT-DBENVIRONMENT THRU A200-EXIT.

        PERFORM B100-PREPARE-IT THRU B100-EXIT.

        PERFORM C100-CREATE-ORDERS THRU C100-EXIT
             UNTIL DONE.

        PERFORM D100-BULK-INSERT THRU D100-EXIT.

        PERFORM A500-TERMINATE-PROGRAM THRU A500-EXIT.

    A100-EXIT.
        EXIT.

    .
    .
    .
```

Use the PREPARE statement to preprocess the dynamic statement, in this case, from a string:

```
    B100-PREPARE-IT.

        MOVE 1 to I.
        MOVE SPACES TO DONE-FLAG.
        MOVE SPACES TO NEWORDERS.

        PERFORM A300-BEGIN-TRANSACTION THRU A300-EXIT.

        EXEC SQL
          PREPARE CMD from
          'BULK INSERT INTO PurchDB.Orders VALUES (?, ?, ?);'
        END-EXEC.
```

Check for processing errors. Display any messages, and either commit or roll back the transaction:

```
        IF SQLCODE = OK
           PERFORM A400-COMMIT-WORK THRU A400-EXIT
        ELSE
           PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT
           PERFORM A450-ROLLBACK-WORK THRU A450-EXIT.

    B100-EXIT.
        EXIT.
```

Load up to 25 rows of new orders for the BULK INSERT. This data could originate

from an interactive user or from a file (In this case, it is an interactive user.):

```
C100-CREATE-ORDERS.

    DISPLAY ' '.
    DISPLAY 'You can specify as many as 25 line items.'.
    DISPLAY ' '.

       MOVE ' Order Number> ' TO PROMPT-USER
       DISPLAY " "
       WRITE PROMPT-USER
       ACCEPT NEWORDERNUMBER(I)

       MOVE ' Vendor Number> ' TO PROMPT-USER
       DISPLAY " "
       WRITE PROMPT-USER
       ACCEPT NEWVENDORNUMBER(I)

       MOVE ' Order Date (YYYYMMDD)> ' TO PROMPT-USER
       DISPLAY " "
       WRITE PROMPT-USER
       MOVE SPACES TO NEWORDERDATE(I)
       ACCEPT NEWORDERDATE(I)

       IF I = 25
         MOVE "X" TO DONE-FLAG
         GO TO C100-EXIT
       ELSE
         PERFORM C200-MORE-LINES THRU C200-EXIT.

C100-EXIT.
    EXIT.

C200-MORE-LINES.

         DISPLAY ' '
         MOVE 'Do you want to specify another line item (Y/N)?> '
           TO PROMPT-USER
         MOVE SPACE TO RESPONSE-ALPHA
         DISPLAY " "
         WRITE PROMPT-USER
         ACCEPT RESPONSE-ALPHA.

    IF RESPONSE-ALPHA NOT = "Y"
    AND RESPONSE-ALPHA NOT = "y"
       MOVE "X" TO DONE-FLAG
       GO TO C200-EXIT
```

```
        ELSE
           COMPUTE I = I + 1.

    C200-EXIT.
        EXIT.
```

Execute the prepared CMD command specifying the array where data for the BULK INSERT is located:

```
    D100-BULK-INSERT.

        DISPLAY ' '.

        MOVE I TO NUMBEROFROWS.
        MOVE 1 TO STARTINDEX.

        MOVE 1 to I.

        DISPLAY 'BULK INSERT INTO PurchDB.OrderItems'.

        EXEC SQL
          EXECUTE CMD USING :NEWORDERS, :STARTINDEX, :NUMBEROFROWS
        END-EXEC.
```

Check for processing errors. Display any messages, and either commit or roll back the transaction:

```
        IF SQLCODE = OK
           PERFORM A400-COMMIT-WORK THRU A400-EXIT
        ELSE
           PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT
           PERFORM A450-ROLLBACK-WORK THRU A450-EXIT.

    D100-EXIT.
        EXIT.
```

⋮

Example in Pascal Using a BULK INSERT

```
⋮
```

Define a host variable array to hold dynamic parameter values. Be sure each host variable data type matches (or is compatible with) its ALLBASE/SQL default data type:

```
EXEC SQL BEGIN DECLARE SECTION;

NewOrders              : array[1..25]
  of record
    NewOrderNumber     : integer;
    NewVendorNumber    : integer;
    NewVendorNumberInd : sqlind;
    NewOrderDate       : packed array[1..10] of char;
    NewOrderDateInd    : sqlind;
  end;
```

If the dynamic parameter represents data for a column that can contain nulls, and it is possible that input data will contain null values, be sure to define a null indicator host variable immediately following the related host variable.

If you are using other than the default values for the starting index and number of rows in the array, define host variables for these as well:

```
StartIndex             : SmallInt;
NumberOfRows           : SmallInt;      (* Maximum possible rows to bulk *)
                                        (* insert. *)

OrderNumber            : integer;       (* Host variables for user input.*)
VendorNumber           : integer;
VendorNumberInd        : sqlind;
OrderDate              : packed array[1..10] of char;
OrderDateInd           : sqlind;

⋮

SQLMessage             : packed array[1..132] of char;

EXEC SQL END DECLARE SECTION;

sqlca                  : SQLCA_type;    (* SQL Communication Area *)

OrdersOK               : boolean;

⋮
```

Use the PREPARE statement to preprocess the dynamic statement, in this case, from a string:

```
procedure PrepareIt;
begin

EXEC SQL PREPARE CMD from 'BULK INSERT INTO PurchDB.Orders VALUES (?,?,?);';

if SQLCA.SQLCODE  OK then         (* Check for processing errors.          *)
  begin
  SQLStatusCheck;
  RollBackWork;
  OrdersOK := FALSE;
  end;

end;                               (* End PrepareIt Procedure.              *)
```

Load up to 25 rows of new orders for the BULK INSERT. This data could originate from an interactive user or from a file:

```
procedure CreateOrders;

var
i:integer;

begin

NumberOfRows := 25;
StartIndex   := 1;
```

Count rows as they are loaded into the NewOrders array up to a maximum of 25:

```
for i := 1 to NumberOfRows do
   begin

   Read a file record or accept a row of data from the user into
   the appropriate host variables.

   Load host variable data into the bulk insert array.

   NewOrders[i].NewOrderNumber := OrderNumber;
   NewOrders[i].NewVendorNumber := VendorNumber;
   NewOrders[i].NewVendorNumberInd := VendorNumberInd;
   NewOrders[i].NewOrderDate := OrderDate;
   NewOrders[i].NewOrderDateInd := OrderDateInd;
```

```
   end;                              (* End for.                              *)
end;                                 (* End procedure CreateOrders.           *)
```

Execute the prepared CMD command specifying the array where data for the BULK INSERT is located:

```
procedure InsertNew;
begin

EXEC SQL EXECUTE CMD USING :NewOrders, :StartIndex, :NumberOfRows;

if SQLCA.SQLCODE  OK then            (* Check for processing errors.          *)
  begin
  SQLStatusCheck;
  RollBackWork;
  OrdersOK := FALSE;
  end;
end;                                  (* End of procedure InsertNew.          *)

⋮

begin                                 (* Begin the program.                    *)

if ConnectDBE then                    (* If the application is successfully    *)
                                      (* connected to a DBEnvironment, proceed. *)
   begin

   OrdersOK := TRUE;

   BeginTransaction;
   PrepareIt;
   CreateOrders;
   InsertNew;

if OrdersOK then                      (* If there were no errors in processing, *)
   CommitWork;                        (* data is committed to the database.    *)
   end;

TerminateProgram;

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

Using Default Data Types with Dynamic Parameters

When the PREPARE statement executes, ALLBASE/SQL assigns a **default data type** to any dynamic parameter in the dynamic section that is created. Depending on how you provide dynamic parameter data to the database, one of the following occurs. If you are using the DESCRIBE INPUT statement, the default data type information is loaded into the related format array. If you are using a host variable, the data type of the host variable is compared to the default, and a conversion is done if possible. (These methods of assigning data are discussed in the "Programming with with Dynamic Parameters" section in this document.) The following topics are discussed in this section:

- How ALLBASE/SQL Derives a Default Data Type.
- Dynamic Parameter Formats.
- Conversion of Actual Data Types to Default Data Types.
- Data Overflow and Truncation.

How ALLBASE/SQL Derives a Default Data Type

The following explains how the default data type of a dynamic parameter is derived by ALLBASE/SQL:

- When the parameter is an operand of an arithmetic operator or a comparison operator, its data type is assumed to be that of the other operand. In the following example, the dynamic parameter is assumed to be an integer because SalesPrice is defined as an integer column in the database:

```
UPDATE PurchDB.Parts
       SET SalesPrice = (SalesPrice * ?)
```

- When the parameter is the second and/or third operand in a BETWEEN predicate, its data type is assumed to be that of the first operand. The assumed data type of both dynamic parameters in the following example is decimal with a precision of six and a scale of two, because this is the data type of the SalesPrice column in the database.

```
SELECT * FROM PurchDB.Parts
         WHERE SalesPrice BETWEEN ? AND ?
```

- When the parameter is any value in an IN predicate, its data type is assumed to be that of the expression. In the following example, the dynamic parameters are assumed to be integers, because OrderNumber is defined as an integer in the database:

```
SELECT * FROM PurchDB.Orders
         WHERE OrderNumber IN (?, ?, ?)
```

- When the parameter is the pattern value in a LIKE predicate, it is assumed to be of character data type. The default length is based on the other operand of the LIKE predicate. In the following example, the default length of the dynamic parameter is 16 since PartNumber is defined as a 16 byte character column.

```
SELECT * FROM PurchDB.Parts WHERE PartNumber LIKE ?
```

- When the parameter is a parameter in the SET clause of an UPDATE statement, its data type is assumed to be that of the update column. The assumed data type of the dynamic parameter in the following example is decimal with a precision of six and a scale of two, because this is the column definition of SalesPrice in the database.

```
UPDATE PurchDB.Parts
       SET SalesPrice = ?
       WHERE PartNumber = '12345'
```

- Used in the VALUES clause of an INSERT statement, its data type is assumed to be that of the inserted column. In the example below, both dynamic parameters are assumed to be of character data type because both PartNumber and PartName are defined as such in the database.

```
INSERT INTO PurchDB.Parts (PartNumber, PartName)
       VALUES(?,?)
```

Note The following examples are syntactically correct but have no or little value semantically. They illustrate additional places where dynamic parameters can be used in accord with standard SQL.

- When the parameter is the first operand in a BETWEEN predicate, its data type is assumed to be that of the second operand. In the following example, the dynamic parameter is assumed to be an integer because 1000 is an integer:

```
SELECT * FROM PurchDB.Orders
         WHERE ? BETWEEN 1000 AND 2000
```

- When the parameter is used as the first and third operands in a BETWEEN predicate, its data type is assumed to be that of the second operand. For example, both dynamic parameters in the following example are assumed to be integer because 1000 is an integer.

```
SELECT * FROM PurchDB.Orders
         WHERE ? BETWEEN 1000 AND ?
```

- When the parameter is the expression in an IN predicate, its data type is assumed to be that of the first value or that of the result column of a subquery. In the following example, the data type of the dynamic parameter is assumed to be integer, because 30507 is an integer value.

```
SELECT DISTINCT * FROM PurchDB.Orders
         WHERE ? IN (30507, 30517, 30518)
```

  In the following example, the data type of the dynamic parameter is assumed to be that of the result column, OrderNumber, which is defined in the database as integer.

```
SELECT * FROM PurchDB.Orders
         WHERE ? IN (SELECT OrderNumber FROM PurchDB.Orders
                        WHERE OrderNumber BETWEEN 1000 AND 2000)
```

- When the parameter is the expression in a quantified predicate, its data type is assumed to be that of the result column of the subquery. In the following example, the data type of the dynamic parameter is assumed to be integer, because that is the column definition of the OrderNumber column.

```
SELECT * FROM PurchDB.Orders
 WHERE ? = ANY (SELECT OrderNumber FROM PurchDB.Orders WHERE OrderNumber >= 500)
```

Note When a dynamic parameter is used in a non-assignment operation and the default data type is determined to be REAL, ALLBASE/SQL promotes it to FLOAT for better performance and data accuracy. Therefore, the assumed data type for a non-assignment operation is never REAL.

Dynamic Parameter Formats

In addition to default data types, dynamic parameters have **default data formats** as shown in the table below:

Table 4-8. ALLBASE/SQL Default Data Formats for Dynamic Parameters

Type of Dynamic Parameter	ALLBASE/SQL Default Data Type	ALLBASE/SQL Default Data Format
column value	LONG BINARY	CHAR(96) - contains the long column descriptor
column value	LONG VARBINARY	CHAR(96) - contains the long column descriptor
column value	BINARY(*n*)	BINARY(*n*)
column value	VARBINARY(*n*)	VARBINARY(*n*)
column value	DATE	CHAR(10)
column value	TIME	CHAR(8)
column value	DATETIME	CHAR(23)
column value	INTERVAL	CHAR(20)
second argument in an ADD_MONTHS function	INTEGER	INTEGER
first or second argument in a TO_DATE, TO_TIME, TO_DATETIME, or TO_INTERVAL function	CHAR or VARCHAR	CHAR(72)
second argument in a TO_CHAR or TO_INTEGER function	CHAR or VARCHAR	CHAR(72)
escape character in a LIKE predicate	CHAR	CHAR(2)
RAISE ERROR number	INTEGER	INTEGER
RAISE ERROR text	CHAR	CHAR(250)
second or third argument in a SUBSTRING function	INTEGER	INTEGER

Conversion of Actual Data Types to Default Data Types

Your application provides a dynamic parameter value at run time by using either a host variable or a set of ALLBASE/SQL data structures and a data buffer. (These coding techniques are further discussed in the section in this document, "Programming with Dynamic Parameters.") When this **actual data type** differs from the ALLBASE/SQL default data type, data conversion takes place from the actual data type to the default data type when either the OPEN or the EXECUTE statement executes. Conversion occurs as follows:

- For assignment operations, if the data types are compatible. (See the ALLBASE/SQL application programming guides and the *ALLBASE/SQL Reference Manual* "Data Types" chapter for further information on data type compatibility.)

  For instance, in an INSERT VALUES clause or an UPDATE SET clause, be sure to assign dynamic parameter data to a program variable having a data type that is compatible with the ALLBASE/SQL default data type for the dynamic parameter.

- For expressions involving a comparison predicate or an arithmetic operator, conversion takes place from a smaller to a larger or equal data type, as shown in table Table 4-9.

  For example, suppose your application specifies a host variable to hold data for a column defined in a table as VARCHAR with a maximum length of 32. If this host variable can hold 32 bytes or less of character data, data conversion will take place. By contrast, if you have defined the host variable to hold more than 32 bytes of character data, it is not smaller than the dynamic parameter default data type, and a run time error will result.

Table 4-9. Actual to Default Data Type Conversion for Dynamic Parameters

Actual Data Type, Based on Your Application	Default Data Type, Based on Dynamic Parameter Usage
SMALLINT, INTEGER, DECIMAL, REAL, or FLOAT	FLOAT
SMALLINT, INTEGER, or DECIMAL	DECIMAL
SMALLINT or INTEGER	INTEGER
SMALLINT	SMALLINT
CHAR(N) or VARCHAR(N)	CHAR($N+M$) or VARCHAR($N+M$) where $M >= 0$ and $N > 0$
CHAR or VARCHAR	*Case Insensitive* CHAR or VARCHAR

Note that for a non-assignment operation, when the default data type is determined to be REAL, ALLBASE/SQL promotes it to FLOAT for better performance and data accuracy. Therefore, the assumed data type for a non-assignment operation is never REAL.

Data Overflow and Truncation

For character data types, no error or warning is given if truncation occurs when an INSERT or UPDATE statement executes. For numeric data types, when zeroes are dropped from the left or when any digit is dropped from the fractional part of DECIMAL or FLOAT values, no error or warning occurs. Otherwise, any overflow or underflow of numeric values causes an error.

5

Using Procedures in Application Programs

This chapter describes the use of procedures in application programs. It highlights features that are available only within application programs and, in particular, methods of passing information between an application and a procedure. The final section compares the use of application code to procedure code to accomplish the same task. The material in this chapter assumes familiarity with the more global presentation of procedures found in the *ALLBASE/SQL Reference Manual* chapter, "Constraints, Procedures, and Rules."

In the present discussion, the term **procedure** refers to a database object that you define with a CREATE PROCEDURE statement. Like an application program, a procedure may have stored sections associated with it and may execute transaction statements. As in an application, when a **severe error** (one having an error code of −4008 or equal to or less than −14024) is encountered in a procedure, any active transaction is automatically rolled back, and the procedure continues.

A difference between a procedure and an application is that the SQL Communication Area (SQLCA) and SQLEXPLAIN cannot be used in a procedure. The built-in variable, ::sqlcode holds the error code for the first message in the message buffer (guaranteed to be the most severe error). When a procedure ends and control returns to the calling application, the SQLCODE field of the SQLCA can be tested. However, only certain types of procedure errors cause SQLCODE to be set. (Refer to the later sections, "Testing SQLCODE and SQLWARN0 on Return from a Procedure" and "Additional Error and Message Handling.")

Within a procedure, you can implement a general form of error checking by coding a WHENEVER SQLERROR STOP statement at the beginning of the procedure. Then whenever an error is encountered during procedure execution, any active transaction is rolled back, the procedure is terminated, and control returns to the application. At this point, SQLCODE contains the error number of the error that caused the WHENEVER SQLERROR STOP statement to be invoked. (Note that, unlike an application, a severe error encountered in a procedure having WHENEVER SQLERROR STOP in effect does not release a DBEnvironment.)

When your application requires more specific information about procedure errors or other information concerning procedure execution, when multiple row result set data is passed from the procedure to the application, and when dynamic parameters are needed to pass data between the application and the procedure, several features are available as described in the following sections:

- Using Cursors with Procedures.
- Using Host Variables to Pass Parameter Values.
- Using Dynamic Procedure Parameters.
- Returning a Return Status Code.
- Testing SQLCODE and SQLWARN0 on Return from a Procedure.
- Returning Output Values.
- Additional Error and Message Handling.
- Comparing a Procedure and an Embedded SQL Application

Note, it is recommended that you handle procedure statement errors and warnings within the procedure.

Using Cursors with Procedures

Two types of cursors are available in ALLBASE/SQL:

- A **select cursor** is one declared for a SELECT statement within either an application or a procedure. It is a pointer used to indicate the current row in a set of rows retrieved by a SELECT statement.

  A select cursor opened in an application program cannot be accessed within the procedure. However, a procedure can open and access its own select cursors.

- A **procedure cursor** is one declared for an EXECUTE PROCEDURE statement within an application. It is a pointer used to indicate the current result set and row in a set of rows retrieved by a set of SELECT statements in a procedure.

  A procedure cursor must be opened and accessed outside of the specified procedure, in an application program. A given application can open more than one procedure cursor.

When you declare a procedure cursor and use procedure cursor processing statements in your application, the read functionality of a select cursor declared in an application is provided for any query with no INTO clause located in the procedure. Results of queries within such a procedure are available within your application, not within the procedure.

A procedure cursor allows callers to process multiple row result sets from a procedure one row at a time either statically or dynamically. Access to multiple row result sets from a procedure is read-only.

Table 5-1 compares the functionality available for a procedure cursor and a select cursor in a procedure.

Table 5-1. Using Cursors with Procedures within an Application

Cursor Type	Available Functionality
Procedure Cursor	BULK processing
	Passing multiple (or single) row query results based on procedure queries to the invoking application
	SQLCA error checking
Select Cursor in a Procedure	UPDATE or DELETE WHERE CURRENT
	Single or multiple row query results
	Built-in variable error checking

Refer to Table 4-4 and Table 4-5 in the "Using Parameter Substitution in Dynamic Statements" chapter in this manual for coding information related to dynamic cursors.

Procedures with Multiple Row Result Sets of Different Formats

The following example shows a procedure definition followed by an excerpt from an application program that uses a procedure cursor:

```
EXEC SQL CREATE PROCEDURE InventoryReport (option INTEGER, qty INTEGER OUTPUT)
AS BEGIN
IF option = 1 THEN
  SELECT PartNumber FROM PurchDB.Inventory;
ELSEIF option = 2 THEN
  SELECT DISTINCT BinNumber FROM PurchDB.Inventory;
ELSE
  SELECT PartNumber, BinNumber, QtyOnHand FROM PurchDB.Inventory;
ENDIF;
SELECT SUM (QtyOnHand) INTO :qty FROM PurchDB.Inventory;
RETURN ::sqlcode;
END;
```

Static Processing

The following example shows the execution of procedure InventoryReport, retrieving multiple row result sets:

First, declare and open a cursor for the procedure returning multiple row result sets.

```
EXEC SQL DECLARE InvRepCursor CURSOR FOR EXECUTE PROCEDURE
    :ReturnStatus = InventoryReport (:opt, :qty OUTPUT);
```

You must initialize the input value for :opt before the OPEN cursor call.

```
EXEC SQL OPEN InvRepCursor;
```

```
while (sqlca.sqlcode >= 0) && (sqlca.sqlcode != 200)
  {
```

Advance to the next query result set from the procedure. Any remaining rows in the current query result set are discarded. Procedure execution continues with the next statement. Control returns to the caller when the next multiple row result set statement is executed or the procedure terminates. The number of columns and format information for the next query result set is returned in the specified SQLDA. This information may be used to process the query result set. When the last result set has been processed, ALLBASE/SQL sets SQLCA.SQLCODE to 200.

```
  EXEC SQL ADVANCE InvRepCursor USING sqldaresult;

  if (sqlca.sqlcode == 0 )
    {
```

```
    while (sqlca.sqlcode == 0)
      {
      Fetch one or more rows from the current query result set.  Use the
      same SQLDA as that specified in the ADVANCE statement.  When the
      last row in the last result set has been fetched, ALLBASE/SQL
      sets SQLCA.SQLCODE to 100. When the last row in the current result
      set has been fetched, ALLBASE/SQL automatically advances to the next
      result set.

      EXEC SQL FETCH InvRepCursor USING sqldaresult;

      if (sqlca.sqlcode = 0)
        {
        Use the number of columns and column format information to process
        the query result.  A detailed description is found in the "Using
        Dynamic Operations" chapters of the ALLBASE/SQL C Application
        Programming Guide.
        }
      }
    }
  }
```

The CLOSE statement will cancel processing of any remaining query result sets. Procedure execution continues with the next statement. No data is returned, nor does control return to the application for any subsequent multiple row result set queries executed by the procedure. The return status, :ReturnStatus, and output parameter, :qty, values are available to the caller after the CLOSE call.

```
EXEC SQL CLOSE InvRepCursor;
```

Dynamic Processing

You need special techniques to handle dynamic EXECUTE PROCEDURE statements. In a program that accepts both EXECUTE PROCEDURE statements, and other SQL commands, you should first PREPARE the command, then use the DESCRIBE command with the OUTPUT option.

The following C pseudocode outlines the above scenario with emphasis on ALLBASE/SQL programming for dynamically executing procedures with multiple row result sets.

Set up a dynamic command with dynamic parameters.

```
:DynamicCmd = "EXECUTE PROCEDURE ? = InventoryReport (?, ? OUTPUT);";
```

Assume you don't know the format for the statement. Prepare the statement.

```
EXEC SQL PREPARE cmd FROM :DynamicCmd;
```

For a dynamic EXECUTE PROCEDURE statement, the DESCRIBE command with the OUTPUT option sets the sqld field of the SQLDA to 0 and sets the sqlmproc field to a non-zero value for a procedure having multiple row result sets. The sqloparm field is set to the number of output parameters (including return status) in the EXECUTE PROCEDURE statement. The sqlfmtarr of the sqldaout is set to the data formats for the return status and output parameters of the procedure, if any. If you know there are no dynamic parameters in the prepared statement, use the EXECUTE statement to execute the dynamically preprocessed statement.

```
EXEC SQL DESCRIBE OUTPUT cmd INTO sqldaout;
```

For a dynamic EXECUTE PROCEDURE statement, the DESCRIBE command with the INPUT option sets the sqld field of the SQLDA to the number of input dynamic parameters in the prepared statement.

```
EXEC SQL DESCRIBE INPUT cmd USING sqldain;
```

If there are input dynamic parameters, the appropriate data buffer or host variables must be loaded with the values for the input dynamic parameters. Since the sqlmproc field is set to a non-zero value, the procedure has multiple row result sets. You define a procedure CURSOR to move through the query results sets row by row.

Declare a cursor for the procedure returning multiple row result sets.

```
EXEC SQL DECLARE InvRepCursor CURSOR FOR cmd;
```

Place the appropriate values into the SQLDA sqldain. Use the USING DESCRIPTOR clause of the OPEN statement to identify where the input dynamic parameter information is located.

```
EXEC SQL OPEN InvRepCursor USING sqldain;
```

```
while (sqlca.sqlcode >= 0) && (sqlca.sqlcode != 200)
  {
```
Use the USING DESCRIPTOR clause of the ADVANCE statement to identify where to place the query result format information. Advance to the next query result set from the procedure. Any remaining rows in the previous query result set are discarded. Procedure execution continues with the next statement. Control returns to the caller when the next multiple row result set statement is executed. The number of columns and format information for the current query result set are returned in the specified SQLDA. This information may be used to process the query result set. When the last result set has been processed, ALLBASE/SQL sets SQLCA.SQLCODE to 200.

```
EXEC SQL ADVANCE InvRepCursor USING sqldaresult;

if (sqlca.sqlcode != 0)
  {
  while (sqlca.sqlcode == 0)
    {
    Fetch as many rows from the current query result set as specified in
    SQLDA.sqlnrow. Specify the same SQLDA as specified in the ADVANCE
    statement. When the last row in the current result set has been
    fetched, ALLBASE/SQL sets SQLCA.SQLCODE to 100.

    EXEC SQL FETCH InvRepCursor USING sqldaresult;

    if (sqlca.sqlcode == 0)
      {
      Use number of columns and column format information to process
      the query result.  A detailed description is found in the "Using
      Dynamic Operations" chapters of the ALLBASE/SQL C Application
      Programming Guide.
      }
    }
  }

}
```

The CLOSE statement will cancel processing of any remaining query result sets. Procedure execution continues with the next statement. No data is returned, nor does control return to the application for any subsequent multiple row result set queries executed by the procedure. Use the SQLDA specified in the DESCRIBE OUTPUT statement to retrieve return status and output parameter values.

```
CLOSE InvRepCursor USING sqldaout;
```

Procedures with no Multiple Row Result Sets

Static Processing

If a procedure is known to contain no multiple row result sets, or the caller does not wish to retrieve such results, a simple EXECUTE PROCEDURE statement can be issued.

```
EXEC SQL EXECUTE PROCEDURE :ReturnStatus = InventoryReport (:opt, :qty OUTPUT);
```

The EXECUTE PROCEDURE statement will return a warning if the procedure contains any multiple row result sets.

Dynamic Processing

In this example, the prepared statement is an EXECUTE PROCEDURE statement with both INPUT and OUTPUT dynamic parameters. After using DESCRIBE INPUT and OUTPUT for cmd, it can be executed using input and output SQL descriptor areas, or input and output host variables.

```
EXEC SQL EXECUTE cmd USING DESCRIPTOR INPUT sqldain AND OUTPUT sqldaout;
```

OR

```
EXEC SQL EXECUTE cmd USING INPUT :opt, :qty AND OUTPUT :ReturnStatus, :qty;
```

The EXECUTE PROCEDURE statement will return a warning if the procedure contains any multiple row result sets.

Single Format Multiple Row Result Sets

Example Schema

The following example defines a procedure returning single format multiple row result sets.

```
CREATE PROCEDURE ReportActivity (Activity CHAR (18))
  WITH RESULT CHAR (20) NOT NULL, SMALLINT AS
  BEGIN
  DECLARE Clubtid TID;
  SELECT TID () INTO :Clubtid FROM RecDB.Clubs
    WHERE Activity = :Activity;
  SELECT ClubName, ClubPhone
    FROM RecDB.Clubs
      WHERE TID () = :ClubTID;
  SELECT MemberName, MemberPhone
    FROM RecDB.Members
      WHERE Club =
        (SELECT ClubName FROM RecDB.Clubs WHERE TID () = :ClubTID);
  END;
```

Static Processing

For static processing, use the DECLARE, OPEN, FETCH, and CLOSE statements to retrieve rows as usual. The ADVANCE statement is not required.

Dynamic Processing

Prepare a dynamic command with dynamic parameters.

```
:DynamicCmd = "EXECUTE PROCEDURE ? = ReportActivity (?);";
```

```
PREPARE cmd FROM :DynamicCmd;
```

For a dynamic EXECUTE PROCEDURE statement, the DESCRIBE command with the OUTPUT option sets the sqld field of the SQLDA to 0 and sets the sqlmproc field to a non-zero value for a procedure having multiple row result sets. The sqloparm field is set to the number of output parameters (including return status) in the EXECUTE PROCEDURE statement. The sqlfmtarr of the sqldaout is set to the data formats for the return status and output parameters of the procedure, if any.

```
EXEC SQL DESCRIBE OUTPUT cmd INTO sqldaout;
```

For a dynamic EXECUTE PROCEDURE statement, the DESCRIBE command with the INPUT option sets the sqld field of the SQLDA to the number of input dynamic parameters in the prepared statement.

```
EXEC SQL DESCRIBE INPUT cmd USING sqldain;
```

If there are input dynamic parameters, the appropriate data buffer or host variables must be loaded with the values for the input dynamic parameters.

If the dynamic command (cmd) is an EXECUTE PROCEDURE statement with single format multiple row result sets, the DESCRIBE RESULT command returns format information for the procedure result sets. In contrast, when the procedure does not have single format multiple row result sets, the sqld field of SQLDA is set to zero.

If the dynamic command is not an EXECUTE PROCEDURE statement, sqlca.sqlcode is set to a non-zero value.

```
DESCRIBE RESULT cmd USING sqldaresult;
```

Declare a cursor for a procedure returning single format multiple row result sets.

```
DECLARE RepActCursor CURSOR FOR cmd;
```

Place the appropriate values into the SQLDA sqldain. Use the USING DESCRIPTOR clause of the OPEN statement to identify where the input dynamic parameter information is located.

```
OPEN RepActCursor USING sqldain;
```

```
while (sqlca.sqlcode >= 0) && (sqlca.sqlcode != 100)
  {
```

Fetch as many rows from the current query result set as specified in SQLDA.sqlnrow. Specify the same SQLDA as specified in the ADVANCE statement. When the last row in the current result set has been

fetched, ALLBASE/SQL automatically advances to the next result set. When the last row in the last result set has been fetched, ALLBASE/SQL sets SQLCA.SQLCODE to 100.

```
FETCH RepActCursor USING sqldaresult;
```

Use number of columns and column format information to process the query result. A detailed description is found in the "Using Dynamic Operations" chapters of the ALLBASE/SQL C Application Programming Guide.

```
}
```

The CLOSE statement will cancel processing of any remaining query result sets. Procedure execution continues with the next statement. No data is returned, nor does control return to the application for any subsequent multiple row result set queries executed by the procedure. Use the SQLDA specified in the DESCRIBE OUTPUT statement to retrieve return status and output parameter values.

```
CLOSE RepActCursor USING sqldaout;
```

Using Host Variables to Pass Parameter Values

You can specify up to 1023 parameters for passing data between an application and a procedure. You can pass values into a procedure using any SQL expression except a subquery, an aggregate function, or a LONG column, string, TID or Date/Time function. Refer to the *ALLBASE/SQL Reference Manual* "Expressions" chapter for more information. (Built-in variables cannot be used as parameters, since they are only available within the procedure.) Input parameter values can be passed using host variables or literal values. For any OUTPUT parameters, you *must* use host variables. (In the EXECUTE PROCEDURE clause of a CREATE RULE statement, you can pass a column name as an input parameter, but column names are not permitted in procedures invoked through the EXECUTE PROCEDURE statement.) Note that this section discusses static parameters. Dynamic parameters are discussed in the following section, "Using Dynamic Procedure Parameters."

The following example shows a procedure definition, followed by an excerpt from an application program that executes the procedure programmatically:

```
CREATE PROCEDURE ManufDB.Process12
(Operator CHAR(20),
Shift CHAR(20),
FailureType CHAR(10) NOT NULL) AS
BEGIN
   INSERT INTO ManufDB.TestMonitor
       VALUES (:Operator, CURRENT_DATETIME,
       :Shift, :FailureType);
```

```
   IF ::sqlcode = 0 THEN
      COMMIT WORK;
      RETURN 0;
   ELSE
      RETURN 1;
   ENDIF;
END;
```

The following example shows the execution of procedure ManufDB.Process12 using host variables to pass in the values for Operator, Shift, and FailureType:

First, declare host variables. Note that the same names are used for host variables in the application program that were used in the parameter definitions of the CREATE PROCEDURE statement. While this is not required, it helps in seeing the relationship between the procedure and the application program.

C Declarations:

```
EXEC SQL BEGIN DECLARE SECTION;
   char Operator[21];
   sqlind OperatorInd;
   char Shift[21];
   sqlind ShiftInd;
   char FailureType[11];
EXEC SQL END DECLARE SECTION;
```

COBOL Declarations:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01   OPERATOR            PIC X(20).
01   OPERATORIND         SQLIND.
01   SHIFT               PIC X(20).
01   SHIFTIND            SQLIND.
01   FAILURETYPE         PIC X(10).
EXEC SQL END DECLARE SECTION END-EXEC.
```

A routine within the application program obtains values for Operator, Shift, and FailureType from a user who enters the data. If either Shift or Operator is null, the corresponding indicator variable is set to −1.

Next, call the procedure to enter the failure information into the database:

In C:

```
EXEC SQL EXECUTE PROCEDURE
ManufDB.Process12(:Operator
:OperatorInd, :Shift
:ShiftInd, :FailureType);
```

In COBOL:

```
EXEC SQL EXECUTE PROCEDURE
ManufDB.Process12 (:OPERATOR
:OPERATORIND, :SHIFT
:SHIFTIND, :FAILURETYPE) END-EXEC.
```

Each host variable name in the EXECUTE PROCEDURE statement maps in sequential order to a parameter definition in the CREATE PROCEDURE statement. Inside the procedure, values are referenced by parameter name, not by host variable name. Thus, parameter names and host variable names need not be the same.

As in the example, you use indicator variables in the parameter list of the EXECUTE PROCEDURE statement to indicate null values. But inside the procedure, the parameter itself contains the null value. In order to pass a null operator name to the procedure, you set the indicator variable OperatorInd to −1 and the content of the host variable *Operator* is undefined. Inside the procedure, the parameter Operator is set to NULL, and this fact can be determined through a test:

```
IF :Operator IS NULL THEN
  PRINT 'Parameter is null';
ELSE
  PRINT :Operator;
ENDIF;
```

Using Dynamic Procedure Parameters

You can specify dynamic parameters in a prepared EXECUTE PROCEDURE statement. Dynamic input parameters are passed from host variables or an SQLDA data buffer in the application to procedure parmeters. Dynamic output parameters are passed from procedure parameters to host variables or an SQLDA data buffer in the application. Table 5-2 shows when dynamic parameter values are passed.

Table 5-2.
When Dynamic Parameters are Passed Between an Application and a Procedure

Type of Processing	Input Parameter Values	Output Parameter Values
Cursor Processing	OPEN	CLOSE
Non-Cursor Processing	EXECUTE PROCEDURE	EXECUTE PROCEDURE

The following statement contains three dynamic procedure parameters:

```
EXECUTE PROCEDURE ? = InventoryReport (?, ? OUTPUT)
```

Note that a procedure return status is always an output (only) parameter. Any other dynamic procedure parameter is assumed to be for input only unless OUTPUT or OUTPUT ONLY is specified.

Refer to the *ALLBASE/SQL Reference Manual* for complete syntax and to the "Using Parameter Substitution in Dynamic Statements" chapter in this manual.

Returning a Return Status Code

A return status code is an integer value returned from a procedure not invoked by a rule. You define the meaning of the code within the procedure. You could use the return status code to indicate the success or failure of the procedure, the value of SQLCODE, or some other flag.

You must declare an integer host variable in your application to hold the code, and use the host variable as a part of the EXECUTE PROCEDURE statement. Any legal host variable name can be used as a return code name. The following example in C uses an integer host variable named Status:

```
EXEC SQL EXECUTE PROCEDURE :Status =
   Process12(:OpName :OpNameInd, :Shift :ShiftInd, :FailureType);
if(sqlca.sqlcode==0) {
   if(Status==0) printf("Failure type entered\n");
   else printf("Failure type not entered. INSERT failed\n");
   }
```

A similar example in COBOL uses an integer host variable named RETCODE:

```
EXEC SQL EXECUTE PROCEDURE :RETCODE = PROCESS12 (:OPERATOR :OPERATORIND,
     :SHIFT :SHIFTIND, :FAILURETYPE) END-EXEC.
IF SQLCODE IS ZERO THEN
   IF RETCODE IS ZERO THEN
      DISPLAY "FAILURE TYPE ENTERED."
   ELSE
      DISPLAY "FAILURE TYPE NOT ENTERED. INSERT FAILED."
   END-IF
END-IF.
```

On return from the execution of the procedure, you first test SQLCODE, and if it is zero, you then test the value of the return code. If SQLCODE is not zero, the return code is undefined, since the procedure failed to execute properly.

Testing SQLCODE and SQLWARN0 on Return from a Procedure

Upon return from an EXECUTE PROCEDURE call, the value of SQLCODE indicates the success of procedure execution itself, not of any individual statement in the procedure. After returning from an EXECUTE PROCEDURE statement, your application should test SQLCODE to determine if the proceedure was successful and SQLWARN0 for possible errors, warnings, and informational messages.

A non-zero SQLCODE is returned from procedure execution in the following situations:

- The procedure was not executed at all. Possible reasons are that the procedure was not found or that the user does not have the appropriate authority.
- An error occurred in the procedure when a WHENEVER SQLERROR STOP statement was active. Such an error causes the procedure to terminate, and the current transaction is rolled back.
- An error occurred in evaluating an IF or WHILE condition or an expression in an assignment statement. Such an error causes the procedure to terminate, and previously executed statements are not rolled back (unless a severe error occurred).
- An error occurred in copying output values to the user's host variables. For example, a null value might be returned, but a null indicator variable was not provided in the parameter list.

In all other cases, SQLCODE is 0 on return from a procedure, *including cases in which errors occurred in a procedure and did not cause the procedure to stop*. Messages for any errors from the last SQL statement executed by the procedure are available on return from the procedure by testing SQLWARN0 for a value of 'W' and using SQLEXPLAIN.

Checking for All Errors and Warnings

The following type of routine is recommended on return from an EXECUTE PROCEDURE statement in C:

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

The following is a similar routine in COBOL:

```
IF SQLCODE IS NOT ZERO OR SQLWARN0 = "W" THEN
   PERFORM M100-DISPLAY-MESSAGE
   UNTIL SQLCODE IS ZERO AND SQLWARN0 <> "W".

⋮

M100-DISPLAY-MESSAGE.
   EXEC SQL SQLEXPLAIN :SQLMESSAGE END-EXEC.
   DISPLAY SQLMESSAGE.
M100-EXIT.
   EXIT.
```

This routine tests both SQLCODE and SQLWARN0 for the presence of error conditions, warnings, and messages of all kinds, including those generated by PRINT and RAISE ERROR statements and resulting from PRINTRULES being set on.

Returning Output Values

You can return data values to an application from a procedure (although not from a procedure invoked by a rule) by using the OUTPUT option in the parameter list. To do this you must use the OUTPUT option in both the CREATE PROCEDURE and EXECUTE PROCEDURE statements. The following example shows the statement that creates such a procedure:

```
CREATE PROCEDURE GetName (PartNumber CHAR(16) NOT NULL,
PartName CHAR(30) NOT NULL OUTPUT) as
BEGIN
   SELECT PartName INTO :PartName
   FROM PurchDB.Parts
   WHERE PartNumber = :PartNumber);
   RETURN ::sqlcode;
END;
```

The following shows how the procedure is invoked from an application program:

Host variables are declared. In this example, the host variable names are different from the parameter names used in the CREATE PROCEDURE statement.

C Declarations:

```
EXEC SQL BEGIN DECLARE SECTION;
   char PartNo[16];
   char Part[31];
   int ReturnStatus;
EXEC SQL END DECLARE SECTION;
```

COBOL Declarations:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  PARTNO          PIC X(15).
01  PART            PIC X(30).
01  RETCODE         PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
```

The application prompts for a part number, then calls the procedure to obtain the part's name from the Parts table, as follows:

In C:

```
EXEC SQL EXECUTE PROCEDURE
:ReturnStatus=GetName
(:PartNo, :Part OUTPUT);
if(sqlca.sqlcode==0)
  if(ReturnStatus==0)
    printf("Name is %s\n", Part);
```

In COBOL:

```
EXEC SQL EXECUTE PROCEDURE
  :RETCODE = GETNAME (:PARTNO,
   :PART OUTPUT) END-EXEC.
IF SQLCODE IS ZERO THEN
  IF RETCODE IS ZERO THEN
     DISPLAY "NAME IS " PART
  END-IF.
END-IF.
```

In the CREATE PROCEDURE statement, two parameters are defined and given data types and sizes: PartNumber and PartName, which is declared for output. In the EXECUTE PROCEDURE statement, the parameters are passed to the procedure in host variables :PartNo and :Part, which is marked for OUTPUT. On a successful return from the procedure, the part name is printed out. (If your application requires an output only parameter, you can specify the ONLY option to avoid unnecessary initialization of procedure parameters.)

Note Be sure to test SQLCODE first and then (if desired) the return status code before examining the data returned in OUTPUT parameters.

Additional Error and Message Handling

The use of procedures in application programming can result in errors and messages at preprocessing time and at run time. At preprocessing time, syntax errors appear in the SQLMSG file. These are like other syntax errors detected by the preprocessor. Here is an example:

```
DBEnvironment = PartsDBE
Module Name   = RULEPROC


CREATE PROCEDURE PurchDB.RemovePart (PartNum CHAR (16) not null) as begin
delete from purchdb.inventory where PartNumber = :PartNum; delet from
                                                            |
*** Error in SQL statement ending in line 33.
*** ALLBASE/SQL statement parser error. (DBERR 10978)
*** Unexpected keyword.  (DBERR 1006)
```

At run time four kinds of messages are generated by procedures. Following an EXECUTE PROCEDURE statement, your application can check the message buffer for these types of messages:

- Messages from failure of the EXECUTE PROCEDURE statement.
- Any messages from the *last* SQL statement executed by the procedure. This includes RAISE ERROR messages.
- Messages from any PRINT statement executed by the procedure.
- Any PRINTRULES messages generated.

As in an application, the most recent or severe ALLBASE/SQL message catalog number is stored in the SQLCODE field of the SQLCA. (It is the number of the first error generated by a statement or the most severe error, if a more severe error occurs after the first.)

Messages from Failure of the EXECUTE PROCEDURE Statement

At run time, the EXECUTE PROCEDURE statement returns an error message if the procedure does not exist, or if a required parameter is not supplied. SQLCODE contains the error number, and SQLWARN0 is set to 'W' if there is also a warning.

The following type of error results from an incorrect procedure call:

```
No value was provided for non-nullable parameter VENDORNUMBER in
procedure PURCHDB.CHECKVENDOR.   (DBERR 2234)
```

Messages from the Last SQL Statement Executed by the Procedure

Inside a procedure, each SQL statement after the declaration of local variables is assigned a statement number, including all the control flow and status statements. Note that the following *do not* have statement numbers:

- BEGIN
- ENDIF
- ELSE
- ENDWHILE
- END

When an SQL statement in a procedure causes the procedure to fail, a message indicating the statement number is loaded into the message buffer. This message is available to your application (by using SQLEXPLAIN) in addition to any other messages in the message buffer. The following is an example of such a message:

```
Error occurred executing procedure PURCHDB.DELVENDOR statement 8.  (DBERR 2235)
```

If a WHENEVER SQLERROR STOP directive is active, an SQL runtime error within the procedure terminates the procedure, causes a rollback and a return to the calling application with SQLCODE set to the error number of the statement that failed.

If no WHENEVER SQLERROR STOP directive is active, the procedure continues to completion if non-severe errors occur. On return from the procedure, SQLCODE is 0, and SQLWARN0 is set to 'W' if the last SQL statement executed by the procedure generated any error or warning messages (or if any PRINT or PRINTRULES messages were generated).

Whether or not WHENEVER SQLERROR STOP is in effect, on exiting the procedure your application can display any messages generated by the procedure by using SQLEXPLAIN.

It is recommended that whenever possible you handle errors within a procedure by examining built-in variables and taking appropriate action. The values returned in built-in variables can be returned to the calling application through OUTPUT parameters or through the RETURN statement. Inside the procedure, only the most serious error encountered is available through the built-in variable ::sqlcode. This value can always be returned to the calling application by means of the return status code, as in the following examples:

In C:

```
EXEC SQL :ReturnCode =
   EXECUTE PROCEDURE
   PurchDB.DelVendor(:VNumber);
if(sqlca.sqlcode==0) {
   if (ReturnCode != 0)
      printf("SQL ERROR\n");
   }
```

In COBOL:

```
EXEC SQL :RETCODE =
   EXECUTE PROCEDURE
   PURCHDB.DELVENDOR (:VNUMBER) END-EXEC.
IF SQLCODE IS ZERO THEN
   IF RETCODE IS NOT ZERO THEN
       DISPLAY "SQL ERROR."
   END-IF
END-IF.
```

Inside procedure PurchDB.DelVendor:

```
DELETE FROM PurchDB.Orders
      WHERE VendorNumber = :VendorNumber;
     RETURN ::sqlcode;
```

Messages from Errors Caused by the RAISE ERROR Statement

The RAISE ERROR statement provides a way of specifying your own error message numbers and text. This is very useful inside procedures that are triggered by rules, and it is also useful if you wish to build a set of error messages of your own. RAISE ERROR lets you assign an error number and a message, as in the following example:

```
RAISE ERROR 7001 MESSAGE 'Vendor number exists in the "Orders" table.';
```

The number range 7000-7999 is reserved for use by this statement (that is, no ALLBASE/SQL errors appear in this range). In the previous example, when the RAISE ERROR statement executes, the number −7001 is placed in the local variable, ::sqlcode, and you can test for the error *within* the procedure. After exiting the procedure, SQLCODE is set to −7001 if a WHENEVER SQLERROR STOP statement is in effect.

The following example illustrates the use of a RAISE ERROR statement in an application that tests for errors following the execution of a procedure by a rule triggered by a DELETE statement. An error is raised in procedure PurchDB.DelVendor and displayed on return to the calling application. The calling application includes the following DELETE statement:

```
DELETE FROM PurchDB.Vendors WHERE VendorNumber = :VendorNumber
```

The attempted deletion fires rule PurchDB.CheckVendor, which invokes procedure PurchDB.DelVendor. The procedure allows the deletion to take place only if the vendor number is not found in other tables. If the vendor number does appear in some other table, an error results. SQLCODE is set to the number of the raised error, and messages like the following are returned to the message buffer, from which they can be displayed with SQLEXPLAIN by the application that encountered the error:

```
Vendor number exists in the "Orders" table.
Error occurred executing procedure PURCHDB.DELVENDOR statement 8.
 (DBERR 2235)
INSERT/UPDATE/DELETE statement had no effect due to execution errors.
 (DBERR 2292)
```

In the first message, the raised error reports that the vendor number exists in the Orders table. A second message in the buffer identifies the location in the procedure of the RAISE ERROR statement that contained the first message. The third message reports the failure of the DELETE statement that fired the PurchDB.CheckVendors rule, which in turn invoked the PurchDB.DelVendor procedure.

Note that for an error raised in a procedure called by an application, SQLCODE and SQLWARN0 are set as described in the previous section, "Messages from Errors Caused by the RAISE ERROR Statement."

RAISE ERROR is the same as other SQL statements in that within or outside of a procedure the message buffer is cleared of other errors before the raised error is stored. (Messages for PRINT and PRINTRULES remain until the procedure returns to the calling application.) Therefore, it is most useful for errors that cause the procedure to return in an error state.

For more information about RAISE ERROR, refer to the section "User Defined Messages" in the "Introduction" section of the *ALLBASE/SQL Message Manual*.

Messages from the PRINT Statement

Use the PRINT statement to store procedure messages in the SQL message buffer. PRINT is useful for presenting informational messages that do not generate an error code in the procedure and for debugging your procedure. When print messages have been generated, on return to the calling application, SQLWARN0 is set to 'W' and all such messages can be retrieved with SQLEXPLAIN.

Here is a C example that uses PRINT statements:

```
if ::sqlcode = 100 then
   print 'Row was not found';
else
   print 'Error in SELECT statement';
endif;
```

On returning from the procedure, use SQLEXPLAIN in a loop to extract all the messages generated by PRINT during the operation of the procedure:

```
while (sqlcode != 0 || sqlwarn[0]=='W') {
 EXEC SQL SQLEXPLAIN :SQLMessage;
 printf("%s\n",SQLMessage);
 }
```

In COBOL:

```
IF SQLCODE IS NOT ZERO OR SQLWARN0 = "W" THEN
   PERFORM M100-DISPLAY-MESSAGE
   UNTIL SQLCODE IS ZERO AND SQLWARN0 <> "W".

⋮

M100-DISPLAY-MESSAGE.
   EXEC SQL SQLEXPLAIN :SQLMESSAGE END-EXEC.
   DISPLAY SQLMESSAGE.
M100-EXIT.
   EXIT.
```

The above routine displays all warnings and errors, including all messages generated by PRINT and as a result of rules firing when PRINTRULES is set on. Note that any message generated by PRINT or resulting from PRINTRULES being set on is loaded into the message buffer each time such a statement executes. Unlike other SQL statement messages, these are not cleared from the message buffer until return to the calling application and just before the next SQL statement executes.

For more information about PRINT, refer to the section "User Defined Messages" in the "Introduction" section of the *ALLBASE/SQL Message Manual*.

Comparing a Procedure and an Embedded SQL Application

Imagine a data entry application which either updates prices or adds new parts to the Parts table in the sample DBEnvironment PartsDBE depending on whether the Part number is currently in the table. You could code this application using conventional embedded SQL programming. In that approach, you would declare host variables, then prompt for data, then access the database:

In C:

```
EXEC SQL BEGIN DECLARE SECTION;
char  PartNumber[17];
float SalesPrice;
sqlind SalesPriceInd;
float InputPrice;
EXEC SQL END DECLARE SECTION;
```

In COBOL:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  PARTNUMBER    PIC X(16).
01  SALESPRICE    PIC S9(8)V9(2) COMP-3.
01  SALESPRICEIND SQLIND.
01  INPUTPRICE    PIC S9(8)V9(2) COMP-3.
EXEC SQL END DECLARE SECTION END-EXEC.
```

Prompt for values for part number and input salesprice,
then test for the existence of the part number in the Parts table.

In C:

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

In COBOL:

```
EXEC SQL SELECT SALESPRICE INTO
:SALESPRICE :SALESPRICEIND
FROM PURCHDB.PARTS
WHERE PARTNUMBER = :PARTNUMBER END-EXEC.
```

If the part number is in the table, update the price.

In C:

```
if (sqlcode == 0) {
   EXEC SQL UPDATE PurchDB.Parts
   SET SalesPrice = :InputPrice
   WHERE PartNumber = :PartNumber;
   if(sqlcode != 0)
      printf("Error on UPDATE\n");
  }
```

In COBOL:

```
IF SQLCODE IS ZERO THEN
   EXEC SQL UPDATE PurchDB.Parts
   SET SalesPrice = :InputPrice
   WHERE PartNumber = :PARTNUMBER END-EXEC.
   IF SQLCODE IS NOT ZERO THEN
      DISPLAY "ERROR ON UPDATE."
   END-IF
END-IF.
```

If the part number is not in the table, add it.

In C:

```
elseif (sqlcode == 100) {
   EXEC SQL INSERT INTO
   PurchDB.Parts (PartNumber,
   SalesPrice) VALUES
   (:PartNumber, :InputPrice);
   if(sqlcode != 0)
     printf("Error on INSERT\n");
   }
```

In COBOL:

```
IF SQLCODE IS 100 THEN
   EXEC SQL INSERT INTO
     PURCHDB.PARTS (PARTNUMBER,SALESPRICE)
     VALUES (:PARTNUMBER,:INPUTPRICE)
   END-EXEC.
   IF SQLCODE IS NOT ZERO THEN
      DISPLAY "ERROR ON INSERT."
   END-IF
END-IF.
```

As an alternative, you could code all this in a procedure, then call the procedure from your application. Here is the CREATE PROCEDURE statement:

```
CREATE PROCEDURE NewPrice(PartNumber CHAR(16) NOT NULL,
                          InputPrice DECIMAL(10,2) NOT NULL) AS
BEGIN
   SELECT PartNumber INTO :PartNumber FROM
   PurchDB.Parts WHERE PartNumber = :PartNumber;

   if ::sqlcode = 0 then /* Row was found, so price is updated */
            UPDATE PurchDB.Parts SET SalesPrice = :InputPrice
               WHERE PartNumber = :PartNumber;
            if ::sqlcode <> 0 then
                print 'Error occurred during UPDATE';
            endif;
   elseif ::sqlcode= 100 then  /* Row not found, so insert it */
            INSERT INTO PurchDB.Parts (PartNumber, SalesPrice)
                 VALUES (:PartNumber, :InputPrice);
            if ::sqlcode <> 0 then
                print 'Error occurred during INSERT';
            endif;
   else
      print 'Error occurred during SELECT';

   endif;
   return ::sqlcode;
end;
```

The following is the code that would be required *in your application* to execute the procedure:

Declare host variables.

In C:

```
EXEC SQL BEGIN DECLARE SECTION
char  Number[17];
double Price;
integer ReturnCode;
EXEC SQL END DECLARE SECTION
```

In COBOL:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  NUMBER       PIC X(16).
01  PRICE        PIC S9(8)V9(2) COMP-3.
01  RETCODE      PIC S9(4) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
```

Prompt for values for part number and input salesprice.

Call the procedure to process the entry.

In C:

```
EXEC SQL EXECUTE PROCEDURE
   :ReturnCode =
     NewPrice(:Number, :Price);

if(sqlca.sqlcode==0) {
  if(ReturnCode!=0)
    printf("Error in NewPrice\n");
  }
```

In COBOL:

```
EXEC SQL EXECUTE PROCEDURE
   :RETCODE =
      NEWPRICE (:NUMBER, :PRICE)
END-EXEC.
IF SQLCODE IS ZERO THEN
    IF RETCODE IS NOT ZERO THEN
        DISPLAY "ERROR IN NEWPRICE."
    END-IF
END-IF.
```

The host variables may be passed as parameters, but host variables are not available within the procedure. You must define parameters to store data that is passed into the procedure. Parameters *PartNumber* and *SalesPrice* are used within the procedure to store the data passed in from the host variables *:Number* and *:Price*. In this example, the two variables are named differently so as to distinguish them. However, you could use the same names for the parameters as for the host variables, since their scopes do not overlap.

Why Use a Procedure?

The advantage of coding a procedure instead of a segment of application code is that you can separate the programmatic function (entering a new price) from the need to know the structure of the database itself. Thus, if the structure of the database should change, it is only necessary to modify the procedure, not the application code. If the procedure is used by many different applications, the savings in application maintenance can be considerable.

6

Using Data Integrity Features

There are several features involving data integrity which are not discussed in the language specific ALLBASE/SQL application programming guides. This chapter deals with them from a programming perspective under the following headings:

- Setting the Error Checking Level.
- Using Table Check Constraints.
- Defining and Dropping Table Constraints.
- Defining and Dropping View Constraints.
- Deferring Constraint Error Checking.
- Locating Constraint Errors

Most examples are based on the recreation database, RecDB, that is provided as part of the sample database environment, PartsDBE.

Refer to the *ALLBASE/SQL Reference Manual* and the ALLBASE/SQL application programming guides for complete syntax and further discussion of constraint features.

Setting the Error Checking Level

You can choose either statement level or row level error checking. The default is statement level; either the entire statement succeeds, or none of it succeeds. For example, if there is an error on the fifteenth row of a twenty-row BULK INSERT statement, the entire statement has no effect and no rows are inserted. When you choose row level error checking, an error on the fifteenth row results in fourteen rows being inserted into the database. An error message is returned in both cases.

Row level checking enhances performance but generally requires additional coding on your part. Statement level checking requires less coding possibly at the expense of additional overhead during statement execution. Partial changes by a statement are tracked by ALLBASE/SQL if statement level error checking is in effect. This allows the statement to be undone should errors occur. Statement level checking is the ISO/ANSI SQL standard.

The SET DML ATOMICITY statement can be used to set error checking to row or statement level at any point in an application. The statement sets the error checking level for all ALLBASE/SQL statements, including those involving constraints. However, it can be temporarily overridden for constraint errors only, as described in a later section.

Note that when a transaction terminates (with either a COMMIT WORK statement or by being rolled back), the error checking level always returns to the default of statement level.

Refer to the "SQL Commands" chapter of the *ALLBASE/SQL Reference Manual* for the full syntax of the SET DML ATOMICITY statement.

Using Table Check Constraints

Check constraints check data against a defined expression in an INSERT or UPDATE statement. You can define check constraints on any column of a table or on a view. For information about view constraints, see the section "Defining and Dropping View Constraints" later in this document. Use table check constraints when you want to test a column against a condition you define rather than against data in the database. This applies only to table checks since they cannot contain subqueries. (Views using WITH CHECK OPTION can contain subqueries and are discussed in a separate section below.)

Table check constraints can be defined at the table level or the column level. At the table level, the constraint can reference multiple columns. At the column level, the constraint can reference only the column on which it is defined. The following syntax applies to both table and column level check constraints:

`CHECK (`*SearchCondition*`)`

The search condition is an expression which must not evaluate to false. If digits and NULLs are combined in an expression, the result is the unknown boolean value. A true or unknown value satisfies a table check constraint.

In the following search condition, for example, when ColumnA equals 15, the search condition is satisfied because the result of the expression is true:

```
ColumnA > 10
```

When ColumnA equals NULL, the search condition is also satisfied because the result of this same expression is unknown. However, when ColumnA equals 5, the search condition is false and a check constraint error would occur (unless constraint checking is currently deferred).

Note that their are restrictions on the search condition of a table check constraint. The "SQL Commands" chapter of the *ALLBASE/SQL Reference Manual* lists these restrictions along with complete syntax for the CREATE TABLE and ALTER TABLE statements.

Defining and Dropping Table Constraints

By using either the CREATE TABLE or the ALTER TABLE statement, you can add any type of table constraint (check, unique, or referential) on an existing column or a new column and you can drop any type of constraint.

The following example uses a CREATE TABLE statement to define a table level check constraint based on the Date and Time columns for the Events table:

```
CREATE PUBLIC TABLE Events
(Event CHAR(30),
 Coordinator CHAR(20),
 SponsorClub CHAR(15),
 Date DATE DEFAULT CURRENT_DATE,
 CHECK (Date >= '1992-02-21' AND Time > '00:00:00') CONSTRAINT DateTimeCheck,
 Time TIME,
 FOREIGN KEY (Coordinator, SponsorClub)
    REFERENCES Members (MemberName, Club) CONSTRAINT Events_FK)
IN RecFS;
```

Note that you can define a table level constraint at any point in the column definition list of a CREATE TABLE statement.

To define a column level check constraint on the Date column of the Events table, the CREATE TABLE statement might look like this:

```
CREATE PUBLIC TABLE Events
(Event CHAR(30),
 Coordinator CHAR(20),
 SponsorClub CHAR(15),
 Date DATE DEFAULT CURRENT_DATE CHECK (Date >= '1992-02-21') CONSTRAINT DateCheck,
 Time TIME,
 FOREIGN KEY (Coordinator, SponsorClub)
    REFERENCES Members (MemberName, Club) CONSTRAINT Events_FK)
IN RecFS;
```

The following examples illustrate use of the ALTER TABLE statement to add a column, add a constraint, and drop a constraint in the Recreation database:

Adding a Column to the Recreation Database

You can use the ALTER TABLE statement to add one or more columns to a table. In the following example the ClubContact column is added to the Clubs table. The new column will contain the member name of the person designated as the contact for the club. The column's value cannot be null.

```
ALTER TABLE RecDB.Clubs
 ADD COLUMN ClubContact CHAR(20) NOT NULL
```

Adding a Constraint to the Recreation Database

With the ALTER TABLE statement, you can add any type of constraint to a table without having to drop the table and recreate it with the new constraint. In the following example, a referential constraint is added to the Clubs table. The new constraint ensures that any club contact name exists as a member name in the Members table.

Note that you must have REFERENCES authority on the Members table to add a constraint that references a column in the Members table.

```
   ALTER TABLE RecDB.Clubs
ADD CONSTRAINT
   FOREIGN KEY (ClubContact)
    REFERENCES RecDB.Members (MemberName) CONSTRAINT (Contact_PK)
```

When the ClubContact column of the RecDB.Clubs table is modified, a club contact name that does not appear in the Members table will cause a referential constraint error.

Note that you cannot add a constraint to a view without dropping the view and recreating it.

Dropping a Constraint from the Recreation Database

A constraint can be dropped from a table without having to drop the table and recreate it. In the following example, a referential constraint is dropped from the Clubs table. After the constraint is dropped, future modifications to the ClubContact column of the Clubs table will not involve a check for a corresponding MemberName in the Members table.

```
ALTER TABLE RecDB.Clubs DROP CONSTRAINT Contact_PK
```

Defining and Dropping View Constraints

In contrast to table constraints, unique and referential constraints cannot be defined on a view. A type of check constraint is available with the CREATE VIEW statement. The WITH CHECK OPTION ensures that modifications made through an updatable view satisfy all conditions of the view definition.

The following example defines a view check constraint named DateCheck based on the view definition of the updatable view named RecDB.EventView:

```
CREATE VIEW RecDB.EventView
            (Event,
             Date)
        AS
    SELECT RecDB.Event,
           RecDB.Date
      FROM RecDB.Events
     WHERE Date >= CURRENT_DATE
WITH CHECK OPTION CONSTRAINT DateCheck
```

When modifications are made through a view defined WITH CHECK OPTION, the new values must be visible in the view definition. That is, any attempt to change data through such a view must satisfy all conditions in the query specification. If this is not so, the view check is violated, an error is returned, and the statement has no effect.

Note that view check constraints are not deferrable, and a SET CHECK DEFERRED statement does not affect them.

Also note that to drop a view check constraint, you must drop the view and recreate it.

Deferring Constraint Error Checking

With the SET CONSTRAINTS statement you can defer constraint error checking for any constraint type for all or part of a transaction. You can defer constraint error checking at any time, within or outside of a transaction. Note, however, that both error checking and constraint error checking are set to their defaults when a transaction ends. Error checking is set to row level, and constraint error checking is set to immediate when a transaction is committed or rolled back.

Within a transaction, the SET CONSTRAINTS statement does not succeed if you attempt to set deferred constraint checking to immediate when constraint errors exist. An error message is generated indicating that there are one or more constraint errors of a particular constraint type. Constraint checking remains deferred. If there are unresolved constraint errors, you can check error codes and choose to correct the the errors and issue a COMMIT WORK statement, or you can issue a ROLLBACK WORK statement.

You might want to defer constraint checking to avoid errors on referential constraints that will be resolved by the end of a transaction. Refer to the "Coding with Deferred Constraint Error Checking" section later in this chapter for an example.

Another use might be to avoid a partially processed SQL statement (due to constraint errors) within a transaction in which row level error checking is in effect. You can defer constraint checking until just before the end of the transaction. Then set constraints to immediate to determine if any constraint errors occurred. If errors were detected, you can correct them before ending the transaction. Refer to the ISQL LOAD command for an example of loading data with row level error checking and constraint checking deferred.

Refer to the "SQL Commands" chapter of the *ALLBASE/SQL Reference Manual* for the full syntax of the SET CONSTRAINTS statement.

Locating Constraint Errors

When your transaction defers constraint checking, you can minimize the possibility of rollback due to constraint errors by setting constraint checking to IMMEDIATE just before the COMMIT WORK statement is executed. Then check sqlcode for constraint errors. If errors were encountered, either prompt the user to make corrections or use the trouble shooting templates below to locate the errors. Once all errors have been corrected, issue a COMMIT WORK statement. The "Coding with Deferred Constraint Error Checking" section provides an additional example.

Template for Single Column Unique Constraint Errors

This template returns the values in rows where a unique value in a single column unique constraint or unique index is duplicated:

```
        SELECT UniqueColumn
          FROM UniqueTable
      GROUP BY UniqueColumn
  HAVING COUNT (UniqueColumn) > 1
```

Template for Multiple Column Unique Constraint Errors

This template returns the values in rows where unique values are duplicated in a multiple column unique constraint or unique index having n columns:

```
        SELECT UniqueColumn1, UniqueColumn2,  ... , UniqueColumnn
          FROM UniqueTable
      GROUP BY UniqueColumn1, UniqueColumn2,  ... , UniqueColumnn
  HAVING COUNT (UniqueColumn1) > 1
     AND COUNT (UniqueColumn2) > 1
      ⋮
     AND COUNT (UniqueColumnn) > 1
```

Template for Single Column Referential Constraint Errors

This template returns the values in rows where the referencing value in a single referencing column matches no referenced value:

```
            SELECT ForeignKeyColumn
              FROM ForeignKeyTable
             WHERE ForeignKeyColumn IS NOT NULL
    AND NOT EXISTS (SELECT *
                      FROM PrimaryKeyTable
                     WHERE ForeignKeyColumn = PrimaryKeyColumn)
```

Template for Multiple Column Referential Constraint Errors

This template returns the values in rows where the referencing values in a multiple column referencial constraint with *n* columns match no referenced values:

```
          SELECT ForeignKeyColumn1, ForeignKeyColumn2,  ... , ForeignKeyColumnn
            FROM ForeignKeyTable
           WHERE ForeignKeyColumn1 IS NOT NULL
             AND ForeignKeyColumn2 IS NOT NULL
              ⋮
             AND ForeignKeyColumnn IS NOT NULL
  AND NOT EXISTS (SELECT *
                    FROM PrimaryKeyTable
                   WHERE ForeignKeyColumn1 = PrimaryKeyColumn1
                     AND ForeignKeyColumn2 = PrimaryKeyColumn2
                      ⋮
                     AND ForeignKeyColumnn = PrimaryKeyColumnn
```

Coding with Deferred Constraint Error Checking

Suppose the user wants to update information in the Clubs table and in the Members table of RecDB. The Club column in the Members table references the ClubName column in the Clubs table, and the ClubContact column in the Clubs table references the MemberName column in the Members table. It is not possible to update both of these tables in the same instant, and a referential constraint error could occur if one table is modified and the other table is still unchanged. In order to resolve these circular referential constraints within the same transaction, you can defer constraint error checking until the end of the transaction at which point all constraints are resolved, as in the following example: (Error checking is set to statement level, the default.)

Execute subroutines to display and prompt for information needed in the Clubs table and the Members table.

Place user entered data in appropriate host variables.

```
BEGIN WORK
```

At this point you want to update the Clubs table. However, ClubContact in the Clubs table references MemberName in the Members table, and the Members table does not yet have the appropriate primary key value inserted.

Defer referential error checking to the transaction level so that all constraints in the transaction can be resolved before constraint errors are checked.

```
SET REFERENTIAL CONSTRAINTS DEFERRED

  UPDATE RecDB.Clubs
     SET ClubName = :NewClubName :ClubNameInd,
         ClubPhone = :ClubPhone,
         Activity = :Activity,
         ClubContact = :ClubContact
   WHERE ClubName = :ClubName
```

These indented statements are shown to illustrate the warning issued when constraint checking is set to a state at which it already exists and to show what constraint errors would stop statement execution if constraint checking had not been deferred.

```
SET REFERENTIAL CONSTRAINTS DEFERRED
```

A warning is issued, since constraints are already deferred.

```
REFERENTIAL constraints already set to DEFERRED.  (DBWARN 2066)
```

A referential constraint error occurs at this point. If you set constraints to IMMEDIATE, an error is issued saying that there are one or more referential constraint errors, but constraints stay deferred because the SET CONSTRAINTS IMMEDIATE statement fails when outstanding constraint errors exist.

```
SET REFERENTIAL CONSTRAINTS IMMEDIATE
```

```
FOREIGN KEY constraint violated. (DBERR 2293)
```

The sqlcode field of the sqlca equals −2293 because no primary key exists for the foreign key ClubContact. Constraint checking remains deferred.

Resolve the unsatisfied constraints by inserting the necessary primary keys in the Members table.

```
INSERT INTO RecDB.Members
   VALUES MemberName = :MemberName,
          Club = :Club,
          MemberPhone = :MemberPhone :MemberPhoneInd
```

Set constraint error checking to IMMEDIATE. If the SET CONSTRAINTS IMMEDIATE statement succeeds, constraints are set to IMMEDIATE. If the SET CONSTRAINTS IMMEDIATE statement fails because of constraint errors, constraints remain deferred. No rollback occurs.

```
SET REFERENTIAL CONSTRAINTS IMMEDIATE
```

Check the sqlcode field of the sqlca. If constraint errors exist, you could code statements that locate them, (See the templates in the previous section.) or you could prompt the user for input to correct the errors.

When all constraint errors are resolved, commit the transaction.

```
COMMIT WORK
```

If sqlcode is negative, the transaction is rolled back. Inform the user. For example, 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 or a new ClubName primary key in the Clubs table or to exit the transaction. Execute the appropriate subroutine.

Else, if sqlcode = 0, tell the user the transaction was successfully completed, and prompt for additional information for the Clubs and Members tables or a return to the main menu display.

7

Transaction Management with Multiple DBEnvironment Connections

It is possible to establish a maximum of 32 simultaneous database connections. When your application must access more than one DBEnvironment, there is no need to release one before connecting to another. Performance is greatly improved by using this method rather than connecting to and releasing each DBEnvironment sequentially.

This **multi-connect functionality** is available in either of two modes. **Single transaction mode** allows one transaction at a time to be active in the currently connected set of DBEnvironments. **Multi-transaction mode** allows multiple, simultaneous transactions across the currently connected set of DBEnvironments with a maximum of one active transaction per connection. The *ALLBASE/SQL Reference Manual* contains an introductory explanation in the "Using Multiple Connections and Transactions with Timeouts" section of the "Using ALLBASE/SQL" chapter. Complete syntax is presented in the "SQL Statements" chapter under SET CONNECTION and SET MULTITRANSACTION. The present chapter concentrates on application programming issues.

There are numerous scenarios in which multi-connect functionality could be useful. For example, a DBEnvironment maintenance application to be run in single-user mode might use multiple, nested transactions to select data from one DBEnvironment, insert it into another, then delete the selected data from the original DBEnvironment. Another application of multi-transaction mode might involve a subroutine that contains a security audit log transaction. The subroutine is called from within a complex transaction whenever a user requests access to data in a particular table. Another scenario might be a windows based application that displays information from several DBEnvironments. This chapter presents some general considerations and provides a pseudocode example. The following topics are addressed:

- Preprocessing and Installing Applications.
- Understanding Timeouts.
- Using Timeouts to Prevent Undetectable Deadlocks.
- Using Timeouts to Prevent Infinite Waits.
- Using Timeouts to Tune Performance.
- Example Using Single-transaction Mode with Timeouts.

Note that although multiple DBEnvironment connections are possible, a given transaction must require resources from just one DBEnvironment.

Preprocessing and Installing Applications

The method of preprocessing and installing a static application in a DBEnvironment is described in detail in the ALLBASE/SQL application programming guides. This section relates specifically to multi-connect applications.

To avoid producing multiple modules for the same application, it is recommended that you preprocess an application *once*. Then install the resulting module in every DBEnvironment accessed by the application.

Be aware that if different users are preprocessing the same application, the owner name of the resulting module defaults to that of the user doing the preprocessing. Unless you specify the owner name in the preprocessor command line, a separate module results for each user who has preprocessed a given application.

Understanding Timeouts

When an application requests a database resource that is unavailable, it is placed on a wait queue. Database resources that cause applications to be placed on a wait queue include the following:

Locks	The application attempts to lock a database object that has already been locked in a conflicting mode.
Transaction Slots	The maximum number of concurrent transactions has been reached and the application attempts to begin a transaction. Note that ALLBASE/SQL creates an implicit, brief transaction when the CONNECT statement is issued.

If the amount of time the application waits is longer than the timeout value, an error occurs and the transaction is rolled back. The application must check the sqlcode field of the sqlca for timeout error 2825.

The strategy for handling timeout errors depends on the specific needs of your application and on your business procedures. When encountering a timeout error, you may want to inform the user that a timeout has occurred and then halt execution of the program. Or, you may want to prompt the user to try again, in case the database resource is now available.

A timeout value can be changed with the following statements:

- SET USER TIMEOUT
- START DBE
- START DBE NEW
- START DBE NEWLOG
- SQLUtil ALTDBE

The SQLUtil SHOWDBE command displays the timeout values that have been set in the DBECon file. (Note that NONE is the default when no timeout value is specified at DBEnvironment creation time.) Remember, however, that DBECon file values can be temporarily overridden with a START DBE or START DBE NEWLOG statement. In such a case, the DBEnvironment startup parameters currently in effect are not reflected by issuing a SHOWDBE command.

Locking and transaction management strategies should be considered when setting timeout values. Refer to the following section "Using Timeouts to Tune Performance" and to the "Programming for Performance" chapter in the ALLBASE/SQL application programming guides for more information.

The following example illustrates how you can check for the occurrence of a timeout error:

```
⋮

CONNECT TO '../sampledb/PartsDBE'

Check the sqlcode field of the sqlca.

If sqlcode equals -2825, the CONNECT has timed out because the maximum
number of transaction slots has been exceeded.  Although the application has
not explicitly begun a transaction, ALLBASE/SQL creates an implicit, short-lived
transaction when a CONNECT is issued.  Since the application has not yet
executed the SET USER TIMEOUT statement, the timeout value in the DBECon file
is still in effect.

To prevent the application from waiting at all for a database resource,
such as the lock needed for an update, set the timeout value to zero.  The
timeout values of other applications are unaffected.

Timeout = 0

SET USER TIMEOUT :Timeout

BEGIN WORK

Check the sqlcode field of the sqlca.

If sqlcode equals -2825, the maximum number of transaction slots has been exceeded.

UPDATE PurchDB.Parts
   SET SalesPrice = SalesPrice * 1.25
 WHERE SalesPrice > 500.00

Check the sqlcode field of the sqlca.

If sqlcode equals -2825, another transaction has placed an incompatible lock
on a database object which your transaction wishes to lock for update.

COMMIT WORK

⋮
```

Using Timeouts to Prevent Undetectable Deadlocks and Infinite Waits

When multi-transaction mode is in effect across multiple DBEnvironments with multiple applications accessing the same DBEnvironments at the same time, it is possible that a deadlock that cannot be detected by ALLBASE/SQL could occur. This is known as an **undetectable deadlock**. In addition, when multi-transaction mode is used with multiple connections to the same DBEnvironment, an **infinite wait** can occur. To avoid these situations, be sure your timeout values are set to a value other than NONE. (Note that NONE is the default when no timeout value is specified at DBEnvironment creation time.)

Undetectable Deadlock Prevention

Suppose your application is simultaneously connected to two DBEnvironments in multi-transaction mode. At the same time, another user's application is connected to the same DBEnvironments. At some point as these applications execute, each one has an active transaction waiting for access to data locked by another transaction in the other application. It is a classic deadlock, and since the data being waited for spans more than one DBEnvironment, ALLBASE/SQL cannot detect this deadlock. The two databases will wait "forever" unless you have set a TIMEOUT value other than NONE for one or both of the application connections.

Infinite Wait Prevention

An application running in multi-transaction mode with multiple active transactions accessing data in the same DBEnvironment can encounter undetectable wait conditions. For instance, the first of two such transactions could be holding locks that cannot be released until the transaction is commited. And the second transaction could require the same locks in order to complete. The second transaction is waiting for the first to commit work, and the first transaction is waiting for the second to return control so that it can commit. Since both transactions are part of the same process, ALLBASE/SQL cannot detect this situation. Unless timeouts are set for at least one of these transactions, they could wait infinitely.

It may not be possible to predict an undetectable wait. Even when the transactions involved are accessing different tables, the physical proximity of system catalog data could mean that locks held by transaction one are required by transaction two. This is particularly true in relation to DDL statements (including the UPDATE STATISTICS statement, even though it is allowed in DML only mode) and dynamic space expansion.

To prevent an infinite wait, be sure to set appropriate timeout values and test for the occurrence of a timeout.

Using Timeouts to Tune Performance

The first connection that attaches to a DBEnvironment defines a set of startup parameters for the DBEnvironment in shared memory. These parameters remain memory resident until the last connection to the DBEnvironment is terminated. ALLBASE/SQL associates each connection with a unique session ID.

When a DBEnvironment connection is initiated with a STARTDBE NEW, SQLUtil ALTDBE, START DBE, or START DBE NEWLOG statement, any specified or default timeout values are in effect for this and all subsequent connections until there are no active connections to the DBEnvironment. The exception is the SET USER TIMEOUT statement. When an application executes this statement, any previously set timeout values are overridden for that particular session. (Note that an application specific timeout value *cannot* exceed a previously set DBECon file maximum timeout value.)

During the execution of a given application, any application specific timeout values are valid for a specific DBEnvironment while the application executes, unless overridden by a subsequent SET USER TIMEOUT statement for the same DBEnvironment.

Example Using Single-transaction Mode with Timeouts

Suppose you want to access three DBEnvironments (PartsDBE, SalesDBE, and AccountingDBE) to simultaneously display related information from each. Depending on your coding environment, you could display data from each DBEnvironment in a separate window or in a specific location in the same window. You choose single-transaction mode because just one transaction at a time must be active.

Since this is a display only application, you decide that all transactions are to be established with the read committed isolation level. You also decide on appropriate timeout values for each transaction and how you want to respond to each possible timeout condition. Your goal is to prevent long waits due to locks held or due to the maximum transaction limit being reached.

The following pseudocode illustrates this scenario:

Define and initialize host variables for DBEnvironment and connection names.

⋮

Put single-transaction mode in effect. Note that although single-transaction mode is the default, it is good coding practice to specify the transaction mode.

```
SET MULTITRANSACTION OFF
```

```
   DECLARE SalesCursor
CURSOR FOR
    SELECT  PartNumber, InvoiceNumber, SalesDate, SalesAmount, CustomerNumber
      FROM  Owner.Sales
     WHERE  PartNumber = :PartNumber
       AND  SalesDate BETWEEN '1991-01-01' AND '1991-06-30'
```

Connect to three DBEnvironments specifying a connection name for each connection. Set a timeout value following each current connection.

```
CONNECT TO :PartsDBE AS :Parts
```

Note that the following statement sets the Parts connection timeout to the maximum specified in the DBECon file. If the maximum is set to NONE (infinity), no timeout can occur. Here we'll assume that it is set to 300 seconds.

```
SET USER TIMEOUT MAXIMUM

CONNECT TO :SalesDBE AS :Sales
SET USER TIMEOUT 30 SECONDS

CONNECT TO :AccountingDBE AS :Accounting
SET USER TIMEOUT 30 SECONDS
```

Set the current connection to Parts.

```
SET CONNECTION :Parts
```

Begin a transaction that accesses PartsDBE. This transaction displays parts data for a range of part numbers. Here, for clarity, the range is hard coded. You could, however, use host variables to prompt the user for the lower and upper limits. Another alternative would be to use dynamic processing, possibly with dynamic parameters.

```
BEGIN WORK RC

BULK SELECT  PartNumber, PartName, SalesPrice
       INTO  :PartsArray, :StartIndex, :NumberOfRows
       FROM  PurchDB.Parts
       WHERE PartNumber BETWEEN 20000 AND 21000
```

Test the sqlcode field of the sqlca. If it equals -2825, a timeout has occurred, and the transaction was rolled back. Display a message and gracefully exit the application.

Otherwise, end the transaction.

```
COMMIT WORK
```

Set the current connection to Sales.

```
SET CONNECTION :Sales
```

Prompt the user for a part number in the displayed range and accept the response into a host variable named PartNumber.

```
OPEN SalesCursor
```

Begin a second transaction that accesses SalesDBE. This transaction displays sales data for the first six months of 1991 based on the PartNumber entered by the user. Here, for clarity, the range is hard coded. You could, however, use host variables to prompt the user for the lower and upper limits of a date range. Another alternative would be to use dynamic processing, possibly with dynamic parameters.

```
BEGIN WORK RC

BULK FETCH  SalesCursor
      INTO  :SalesArray, :StartIndex2, :NumberOfRows2
```

Test the sqlcode field of the sqlca. If it equals -2825, a timeout has occurred, and the transaction was rolled back. Display a message and prompt the user to try again or exit the application.

If they choose to try again, re-execute the transaction.
If they choose to exit the application, do so gracefully.

If no timeout error (or other error) occurred, continue.

```
COMMIT WORK
```

Set the current connection to Accounting.

```
SET CONNECTION :Accounting
```

Prompt the user for an invoice number, and accept the response into a host variable named InvoiceNumber.

Begin a third transaction accessing AccountingDBE. This transaction displays accounting data for a part number and an invoice number based on the user entered part number and invoice number. Again you could use dynamic processing, possibly with dynamic parameters.

```
BEGIN WORK RC
```

```
BULK SELECT  InvoiceNumber, PartNumber, InvoiceDate, DueDate, DatePaid
       INTO  :AccountingArray, :StartIndex3, :NumberOfRows3
       FROM  Owner.Accounting
      WHERE  InvoiceNumber  = :InvoiceNumber
        AND  PartNumber = :PartNumber
```

Test the sqlcode field of the sqlca. If it equals -2825, a timeout has occurred, and the transaction was rolled back. Display a message and prompt the user to try again or exit the application.

If they choose to try again, re-execute the transaction.
If they choose to exit the application, do so gracefully.

If no timeout error (or other error) occurred, continue.

```
COMMIT WORK
```

At this point you could loop back to ask the user for either another invoice number for the same part number, another part number from the range already selected in the first transaction, or a new range of part numbers. Or you could issue the DISCONNECT ALL statement and exit the application.

8

COBOL Preprocessor Enhancements

The following COBOL preprocessor features are not discussed in the *ALLBASE/SQL COBOL Application Programming Guide*:

- Record Descriptions For Non-Bulk Queries.
- Host Variables Initialized With The VALUE Clause.

Record Descriptions For Non-Bulk Queries

Prior to this release, record descriptions were allowed only for host variables referenced in bulk queries. Host variables used in non-bulk queries can now be grouped together into logical records. This facilitates data movement, data initialization, and code readability.

Group level identifiers cannot be referenced in non-bulk queries. In the following example, PART-RECORD cannot be referenced in a non-bulk query, but PARTNUMBER can be referenced.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  PART-RECORD.
    05  PARTNUMBER               PIC X(16).
    05  PARTNAME                 PIC X(30).
    05  SALESPRICE               PIC S9(8)V99 COMP-3.
    05  SALESPRICEIND            SQLIND.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
```

Host Variables Initialized With The VALUE Clause

Host variables can now be initialized with the VALUE clause when they are declared. For example:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  CREDIT-LIMIT         PIC  S9(7)V99 COMP-3  VALUE 1800.00.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
```

9

Programming with Indicator Variables in Expressions

Prior to this release, host variable indicator variables could be specified with any output host variable and with some input host variables, including those used as parameters to a data/time input function. With this release, you can specify an indicator variable with *any* input host variable, and indicators can be present in an expression.

Input host variables and their related input indicator variables can be used to provide column data in INSERT, UPDATE, and UPDATE WHERE CURRENT statements. Use them in any WHERE or HAVING clause in which host variables can be used. You cannot use host variables or indicator variables in a any DDL statement. Refer to the "Host Variables" chapter in your ALLBASE/SQL application programming guide for further details regarding host variables.

The indicator variable associated with a host variable determines whether the value in its host variable is considered to be NULL or to be the value stored in the host variable. This is the case for both input and output indicator variables.

Suppose you are writing an application that updates the PurchDB.Inventory table in the DBEnvironment PartsDBE. After selecting a row, you test the QtyOnHand column to see if it contains a negative value. If it does, you want to set the QtyOnHand to NULL. The following example shows the use of an output host variable with an output indicator variable to obtain values by means of a select list and an input host variable with an input indicator variable to supply values in a SET clause:

```
BEGIN DECLARE SECTION

Declare the QtyOnHand host variable and the QtyOnHandInd indicator variable.
In this example, they are used for data output and data input.

END DECLARE SECTION

⋮

SELECT QtyOnHand
  INTO :QtyOnHand :QtyOnHandInd
  FROM PurchDB.Inventory
 WHERE PartNumber = :PartNumber

Test the QtyOnHandInd output indicator variable.  If it contains a negative
number, the QtyOnHand column is already NULL.  If it contains zero or a
positive number, test the QtyOnHand output host variable for a negative
number.  If it is negative, update the column value to be null as follows.
```

Set the QtyOnHandInd input indicator variable equal to -1. No matter what value is in the QtyOnHand input host variable, the QtyOnHand column is set to NULL.

```
UPDATE PurchDB.Inventory
   SET QtyOnHand = :QtyOnHand :QtyOnHandInd
 WHERE PartNumber = :PartNumber

⋮
```

You can include an indicator host variable in an expression. In the following example, if QtyOnHandInd is non-negative, the value of QtyOnHand increases by two (The value of QtyOnHandInd is unchanged.):

```
UPDATE PurchDB.Inventory
   SET QtyOnHand = :QtyOnHand :QtyOnHandInd + 2
 WHERE PartNumber = :PartNumber
```

10

Analyzing Queries with GENPLAN

The GENPLAN statement can be useful in determining the way to write a SELECT, UPDATE, or DELETE statement for maximum performance. By issuing the GENPLAN statement in ISQL, you can see the optimizer's access plan for a given statement.

Suppose you have written an application containing a query, as in the following example:

```
SELECT PartName, VendorNumber, UnitPrice
  INTO :PartName, :VendorNumber, :UnitPrice
  FROM PurchDB.Parts p, PurchDB.SupplyPrice sp
 WHERE p.PartNumber = sp.PartNumber
   AND p.PartNumber = :PartNumber
```

You run the application and want to improve its performance. One approach would be to issue the ISQL GENPLAN statement with parameters for the embedded query. This provides any scan types and join types for a given statement by query block. You can change the statement in your application, run the application to check any change in performance, and, if necessary, again use GENPLAN to determine the specific access path.

To convert a statement to GENPLAN format:

- Remove the SELECT statement INTO clause.
- Remove any null indicator variables from the select statement.
- In the GENPLAN statement WITH clause, define any input host variables found in the SELECT statement WHERE clause. You must ensure that the SQL data type specified for each variable in the WITH clause is compatible with the data type declared in the application for the host variable. Refer to *ALLBASE/SQL Reference Manual*, SQL Commands chapter, for data type compatibility charts for each supported language.

Your GENPLAN statement for the above query would be as follows (The input host variable is shaded.):

```
GENPLAN WITH ( PartNumber char(16) ) FOR
      SELECT PartName, VendorNumber, UnitPrice
        FROM PurchDB.Parts p, PurchDB.SupplyPrice sp
       WHERE p.PartNumber = sp.PartNumber
         AND p.PartNumber =  :PartNumber
```

To display the access plan generated by GENPLAN, issue the following statement within the same transaction as the GENPLAN statement:

```
SELECT * FROM System.Plan
```

- The GENPLAN statement can only be used in ISQL. It cannot be used in an application in a static SQL statement nor in dynamic preprocessing.

Refer to the *ALLBASE/SQL Reference Manual* for detailed syntax and information regarding scans and indexes.

11

Using the VALIDATE Statement

When you run an application, ALLBASE/SQL automatically and transparently checks each section for validity before executing it. When a section is found to be invalid, ALLBASE/SQL revalidates it (if possible), then executes it. You may notice a slight delay as the revalidation takes place. To avoid the delay of runtime revalidation, you can re-preprocess the program or use the VALIDATE statement to revalidate the affected modules prior to runtime. Complete syntax for the VALIDATE statement is presented in the *ALLBASE/SQL Reference Manual*.

You may find it appropriate to use the VALIDATE statement after executing an UPDATE STATISTICS statement, since that statement will invalidate stored sections. If you issue both statements during a period of low activity for the DBEnvironment, the optimizer will have current statistics on which to base its calculations, with minimal performance degradation.

The validate statement will not revalidate sections that have never been validated under the F.0 release, for example, sections that have been migrated from a previous release. These sections can be revalidated by running the application to execute all its sections. An alternative is to recreate the module by repreprocessing the application. Thereafter, you can use the VALIDATE statement.

12

Corrections to the BCDToString Example Program Routine

This chapter corrects the BCDToString routine in the "Using Dynamic Operations" chapter of the C and Pascal ALLBASE/SQL application programming guides. For each language, replacement pages are supplied. Affected lines of the program are highlighted for ease in noting the changes.

Correcting the C Language Program

The replacement pages for program cex10a in chapter 8 of the "ALLBASE/SQL C Application Programming Guide" appear on the next three pages. The first of the three contains only the unchanged comments for the routine.

```
/* DataBuffer is the buffer containing retrieved data as a result */
/* of a dynamic SELECT. */

char     DataBuffer[MaxDataBuff];
boolean  Abort;

struct SQLVarChar {
  int      Length;
  char     VarCharCol[MaxColSize];
  };

main()    /* Beginning of Program */
{
printf("\nC program illustrating dynamic command processing -- cex10a");
printf("\n");
printf("\nEvent List:");
printf("\n  CONNECT TO PartsDBE");
printf("\n  Prompt for any SQL command");
printf("\n  BEGIN WORK");
printf("\n  PREPARE");
printf("\n  DESCRIBE");
printf("\n  If command is a non-query command, EXECUTE it");
printf("\n  Otherwise execute the following:");
printf("\n  DECLARE CURSOR");
printf("\n  OPEN Cursor");
printf("\n  FETCH a row");
printf("\n  CLOSE Cursor");
printf("\n  COMMIT WORK");
printf("\n  Repeat the above ten steps");
printf("\n  RELEASE PartsDBE\n");

if (ConnectDBE()) {                                          (4)
    Describe();                                              (23)
    ReleaseDBE();
    printf("\n");
  }
  else
    printf("\nError: Cannot Connect to PartsDBE");
    printf("\n");
}   /* End of Main Program */


/* Function BCDToString converts a binary field in the "DataBuffer" */
/* buffer to its ASCII representation.  Input parameters are */
/* the Length, Precision and Scale.  The input decimal field is passed */
/* via "DataBuffer" and the output String is passed via "result".  */
```

Figure 8-9. Program cex10a: Dynamic Commands of Unknown Format (page 2 of 11)

```
int BCDToString(DataBuffer, Length, Precision, Scale, Result0)      (2)
char  DataBuffer[];
short Length, Precision, Scale;
char  Result0[];
{
#define  hexd        '0123456789ABCDEF'
#define  ASCIIZero   '0'
#define  PlusSign    12
#define  MinusSign   13
#define  UnSigned    14
#define  btod(d,i)   ((i&1)?((d[i/2])&0xf):((d[i/2]>>4)&0xf))

  int       i;
  int       DecimalPlace;
  int       PutPos=0;
  int       DataEnd;
  int       DataStart;
  boolean   done;
  char      space[MaxStr];
  char      *Result;

  Result = space;
  DataEnd = (Length*2) - 2;
  DataStart = (DataEnd - Precision) + 1;
  for (i = 0; i < MaxStr; i++) Result[i] = '\0';
  DecimalPlace = (Precision-Scale) - 1;

/* convert decimal to character String */
  if (DecimalPlace == 0) Result[PutPos++] = '.';

  /* convert each Nibble into a character */
  for (i = DataStart; i <= DataEnd; i++) {
    Result[PutPos] = ASCIIZero + btod(DataBuffer,i);
    if (PutPos == DecimalPlace) Result[++PutPos] = '.';
    PutPos++;
    }

  i    = 0;
  done = FALSE;
  while (i<strlen(Result) && Result[i]=='0') ++Result;

  if (Result[0] == '\0')
     Result[0] = '0';
  else {
     /* place a zero at the left of the decimal point */
     if (Result[0] == '.') StrInsert('0', Result);
```

Figure 8-9. Program cex10a: Dynamic Commands of Unknown Format (page 3 of 11)

```
      /* insert sign */
      switch (btod(DataBuffer,(DataEnd + 1))) {
          case PlusSign:  StrInsert(' ', Result);
                          break;
          case MinusSign: StrInsert('-', Result);
                          break;
          default:        break;
          } /* End switch */
       } /* End else */
  strcpy(Result0, Result);
} /* End BCDToString */

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

int SQLStatusCheck()  /* Function to Display Error Messages */      3
{

Abort = FALSE;
if (sqlca.sqlcode < DeadLock) Abort = TRUE;

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

if (Abort) {

  EXEC SQL COMMIT WORK RELEASE;
  DynamicCommand[0] = '/';
  DynamicCommand[1] = '\0';
  }

}  /* End SQLStatusCheck Function */


int ConnectDBE()  /* Function to Connect to PartsDBE */
{                                                                   4
boolean Connect;
printf("\nConnect to PartsDBE");
EXEC SQL CONNECT TO 'PartsDBE';                                    37
```

Figure 8-9. Program cex10a: Dynamic Commands of Unknown Format (page 4 of 11)

Correcting the Pascal Language Program

The one replacement page for program pasex10a in chapter 10 of the "ALLBASE/SQL Pascal Application Programming Guide" appears on the next page.

```
  Abort             : boolean;
$PAGE $
(* Procedure BCDToString converts a decimal field in the "DataBuffer"
 * buffer to its decimal presentation.  Other input parameters are
 * the Length, precision and Scale.  The input decimal field is passed
 * via "DataBuffer" and the output String is passed via "result".
 *)
procedure BCDToString (DataBuffer : BCDType; Length : SmallInt;(2)
                       Precision : SmallInt; Scale : SmallInt;
                       var Result : String);
const
  hexd         = '0123456789ABCDEF';    (* Hexadecimal digits #001*)
  ASCIIZero = ord('0');
  PlusSign  = 12;
  MinusSign = 13;
  UnSigned  = 14;
var
  i,
  DecimalPlace,
  PutPos,
  DataEnd,
  DataStart : Integer;
  done  : boolean;

begin
  DataEnd := (Length*2) - 1;
  DataStart := (DataEnd - Precision) + 1;
  Result := StrRpt (' ',StrMax(Result));
  DecimalPlace := (Precision-Scale) -1;

(* convert decimal to character String *)
  if DecimalPlace = 0 then
    begin

      Result[1] := '.';
      PutPos := 2;

    end
  else
    PutPos := 1;
    for i := DataStart to DataEnd do
    begin
```

Figure 10-9. Program pasex10a: Dynamic Commands of Unknown Format (page 2 of 12)

Index

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W