PrimeBase™
Reference Manual
3.0

Virchowstr. 17

22767 Hamburg

Development Manager: Paul McCullagh

Software Development: Mitchell Frazer, Dieko Jacobi, Ingo Karge, Barry Leslie, Dirk Strack

Special Thanks: To all our ex-collegues at P.INK.

Hamburg, 12. July 1996

CONTENTS

DATA TYPES

Integer types

TINYINT	an unsigned 8-bit integer.
SMINT, SMALLINT	a signed 16-bit integer.
INT, INTEGER	a signed 32-bit integer.

```
<integer_literal>::= [ '-' | '+' ] <digit> {<digit> }
```

```
<digit>::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' |
'9'
```

Decimal types

DECIMAL, NUMERIC	a signed decimal number that has a total number of decimal digits and a scale, which is the total number of digits to the right of the decimal point.
MONEY	a special type of decimal value that can be converted to and from character strings in the form of currency values, for example, $12.34, 1.234,000 DM.

```
<decimal_literal>::= [ '-' | '+' ] <digit> { <digit> } '.'
<digit> { <digit> }
```

```
<money_literal> = '$' <decimal_literal>
```

Floating-point types

SMFLOAT, SMALLFLOAT
: a 4-byte floating point value.

REAL
: a 4-byte floating point value.

FLOAT
: an 8-byte floating point value.

REAL10
: a 10-byte floating point value.

REAL12
: a 12-byte floating point value.

```
<float_literal>::= [ '-' | '+' ] <digit> { <digit> } [ '.'
<digit> { <digit> } ](E|'e')['-'|'+']{'0'..'9'}
```

Boolean types

BOOLEAN
: a truth value. PrimeBase uses 3-valued logic, therefore a boolean value can either be true false, or maybe.

```
<boolean_literal>::= '$TRUE' | '$FALSE' | '$MAYBE'
```

Time types

DATE
: a 4-byte value consisting of the year, the month and the day.

TIME
: a 4-byte value consisting of hours (0-23), minutes, seconds and hundredths of a second.

TIMESTAMP, DATETIME
: an 8-byte value consisting of a date and a time value.

Submitted as character literals, according to the current value of either $datefmt, $timefmt, or $tsfmt.

Character types

CHAR, CHARACTER	a fixed length character string.
VARCHAR	a variable length character string.

```
<character_literal>::= ( '"' | ''' ) { <character> } ( '"' | ''' ) | ':' <var_name>
```

Binary types

BIN, BINARY	a fixed length byte string.
VAR, VARBINARY	a variable length byte string.

Special types

GENERIC	an item used to declare a variable that can assume any of the other data types when data is assigned to it.
OBJNAME	a data item whose value identifies an identifier.

DDL REFERENCE MANUAL

This section provides a reference guide to the following statement groups: data control statements, database statements, information statements, object manipulation statements and users, groups and privileges. An explanation of the function of each command is given, followed by the syntax, and an explanation of each part of the syntax. Any idiosyncrasies of a particular command are noted under the section, “notes“.

All examples throughout this chapter are based a the “golfers” database which is provided in Appendix C.

The following list shows the statements of this manual grouped according to the kind of statement they are. The order in which they are documented in this manual, however, is alphabetical.

DBMS Statements

- OPEN DBMS
- CLOSE DBMS
- USE DBMS

Database Manipulation Statements

- OPEN DATABASE
- CLOSE DATABASE
- USE DATABASE
- CREATE DATABASE
- BACKUP DATABASE

- RESTORE DATABASE
- MOUNT DATABASE
- UNMOUNT DATABASE

Information Statements

- DESCRIBE DBMS
- DESCRIBE OPEN DBMS
- DESCRIBE DATABASES
- DESCRIBE OPEN DATABASES
- DESCRIBE TABLES
- DESCRIBE COLUMNS
- DESCRIBE LINKSETS
- COMMENT SPECIFICATION

Object Manipulation Statements

- CREATE DOMAIN
- CREATE TABLE
- ALTER TABLE
- OPEN TABLE
- CLOSE TABLE
- BACKUP TABLE
- REORG TABLE
- CREATE KEY
- CREATE DEFAULT
- CREATE INDEX
- CREATE RULE
- CREATE VIEW

- CREATE VARIABLE
- DROP <object>
- RENAME <object>
- SET VARIABLE

Database Privileges Statements

- ADD USER
- REMOVE USER
- ALTER USER
- CREATE GROUP
- DROP GROUP
- GRANT
- REVOKE

Server Control Statements

- SERVER RESTART
- TRANS RESTART
- SERVER SHUTDOWN
- TRANS SHUTDOWN
- SERVER ERROR
- TRANS ERROR
- SERVER RESTORE
- SERVER CHECKPOINT
- SERVER COMMENT

ADD USER

function — This statement is used to either add one or more users to a database, or, if the TO clause is included, to add the user to a specific group within the database.

syntax

```
ADD USER <user_name> { <user_detail> } {',' <user_name>
<user_detail> {<user_detail>} } [ TO <group_name> ] ';'

<user_name>::= <character_literal>

<user_detail>::= CREATOR <creator_name> | PASSWORD <password> |
ABORT TIME <expression>

<password>::= <character_literal>

<group_name>::= <character_literal>
```

parameters

ADD USER	identifying keywords.
<user_name>	the name of the user who you are adding to the database. It must be a character literal - any printable character, enclosed by quotation marks.
<user_detail>	information on the user being added - concerning password, creator name and the user´s transaction abort time, which is the amount of time that a transaction started by this user may be idle, before it is aborted by the transaction manager.
CREATOR	keyword, indicating that the creator name for the new user follows. This name must adhere to the rules for identifiers, page 103. This clause is only necessary if you are adding a new user to the database. If this clause is omitted, the default creator name is “Common”.
<creator_name>	the creator name of the user. The users creator name is used when the user creates objects. All objects have names consisting of two parts. The first part is the creator name of the user creating the object, and the second part is the name given to the object at creation time

PASSWORD	keyword, indicating that the password to be used by the new user follows. As with the CREATOR clause, this clause is only necessary if you are adding a new user to the database. The default password is an empty string:"".
<password>	the actual password the user will use.
ABORT TIME	This clause allows the user´s transaction abort time to be set. The value given is in seconds, and represents the amount of time a transaction is allowed to remain idle before it is aborted. An idle transaction is a transaction that does no disk I/O. It is also the time that the system takes to detect a complex deadlock between transactions. A complex deadlock is a deadlock that involves more than two transactions. The default abort time is 30 minutes.
<expression>	A value given in seconds, representing the amount of time a transaction is allowed to remain idle before it is aborted.
TO	this clause enables you to assign the user to a specific group within the database. This clause is optional.
<group_name>	the actual name of the group to which you are assigning this user.

notes

A user may belong to any number of groups within the database; not just one group.

The <user_name> is unique within the whole database, and is used for log-in purposes. It identifies a user, and as a result, identifies also that user´s privileges.

The creator_name is an identifier, and therefore must conform to the rules for identifiers. It is not necessarily database-wide unique. This name also becomes a component of the qualified name of any objects created by that user. In addition, if a user specifies a database object without completely qualifying it, then that user´s <creator_name> is automatically used. If no object is found, the system will try using the creator names “Common” and “System”.

New users to a database are automatically assigned to the system group called “Public”.

To add a user to a normal database, the database must be in use, and the user must be the DBA.

To add a user to the Master database, the user must be an SA, and the Master database must be in use. Users added to this database do not receive SA status; they are, however, allowed to create their own databases. A user of the Master database can be promoted to SA, by granting SA privileges.

Users of the Master database that have SA (System Administrator´s) privileges can open any database, and SAs are automatically made DBA (Database Administrator) of any database the open. However, normal users of the Master database that do not have SA privileges are not automatically made DBA of any database they open.

Users of the Master database, that do not have SA privileges, are not automatically made DBA of any database they open, as is the case with the SA.

examples

```
/***In this example, the user, called Caspar Fyson is added to
the database. He is given a creator name, “golf”, the password,
“Birdy”, and is made a member of the group called “Golfer-
sPros”.***/

ADD USER “Caspar Fyson” CREATOR golf PASSWORD “Birdy” TO “Golf-
ersPros”;
```

see also create user, alter user, remove user, grant, revoke, create group, drop group

ALTER TABLE

function This statement is used to change the structure of an existing relation.

syntax

```
ALTER TABLE <table_reference> <column_command> ';'

<column_command>::= <append_column> | <rename_column>

<append_column>::= (APPEND | ADD) [COLUMN] <column_def>

<rename_column>::= RENAME [COLUMN] <column_name> TO
<column_name>
```

parameters	ALTER TABLE	statement identifying keywords.
	<table_reference>	the qualified name of the relation you want to alter.
	<column_command>	defines how you want to alter the relation; either APPEND or RENAME a column.
	<append_column>	adds another column to the relation.
	<rename_column>	changes the name of an existing column.
	APPEND	keyword.
	<column_def>	the definition of the column that is to be altered; either simple column definition or composite column definition. (See CREATE TABLE, page 44.)
	RENAME	keyword.
	<column_name>	the name of the column you want to rename.

notes In the APPEND clause, the name of the column must be distinct from those already existing in the table. If <column_def> is a composite column definition then the component columns must be simple columns already existing in the table. New columns must allow NULLs, as the value stored in new simple columns is NULL.

examples

```
ALTER TABLE Golfers

APPEND COLUMN
medalswon INT;
ALTER TABLE Golfers
RENAME COLUMN medalswon TO medals;
```

see also create table, drop table, rename table, reorg table, backup table, check table, open table, close table

ALTER USER

function This statement is used to alter details associated with a user.

syntax

```
ALTER [ USER <user_name> ] <user_detail> {<user_detail>} {','
<user_name> <user_detail> { <user_detail> } } ';'

<user_detail>::= CREATOR <creator_name> | PASSWORD <password> |
ABORT TIME <expression>

<password>::= <character_literal>

<group_name>::= <character_literal>
```

parameter

ALTER	statement identifying keyword.
USER	if this clause is omitted then the current user is assumed.
<user_name>	the name of the user to be affected by this statement.
CREATOR	to set the new creator name for the user.
<creator_name>	the new creator name. It must conform to the rules for identifiers, page 103.
PASSWORD	to set the new password for the user.
<password>	the new password.
ABORT TIME	the maximum time a transaction (that belongs to the user) may be idle. A transaction is idle when not reading or writing. For example, when a transaction is waiting for a lock, it is idle.
<expression>	the time in seconds that the transaction may be idle.

notes

Users may set their own passwords, but only the DBA may set the creator name of a user and the password of another user.

If the USER clause is omitted, then the current user is assumed.

examples

```
/***In this example, the password of the user, Caspar Fyson is
altered, and is changed to “Eagle”. The ABORT TIME is set at 5
seconds.***/

ALTER USER “Caspar Fyson” PASSWORD “Eagle” ABORT TIME 5;
```

see also

create user, add user, remove user, grant, revoke, create group, drop group

BACKUP DATABASE

function — This statement is used to backup a database.

syntax

```
BACKUP DATABASE <database_name> {<file_location_spec>}
{<backup_options>} ';'

<file_location_spec>::= [DATA|INDEX] [IN] LOCATION
<character_literal>

<backup_options> ::= <include_index> | <preserve_previous>

<include_index> ::= (WITH | WITHOUT) INDEX

<preserve_previous> ::= (REPLACE | PRESERVE) [PREVIOUS]
```

parameters

BACKUP DATABASE	keywords.
<database_name>	the name of database.
<file_location_spec>	this is an optional clause used to specify backup locations for the database.
DATA	indicates a path specified for data files.
INDEX	indicates a path specified for index files.
IN	an optional keyword.
LOCATION	indicates file system location follows.
<character_literal>	location in the file system.
(WITH | WITHOUT) INDEX	these are optional keywords, to specify whether or not the database should be backed up including the indices, (WITH INDEX).
(REPLACE | PRESERVE) [PREVIOUS]	these are optional keywords, to indicate whether a previous backup in the backup location should be overwritten or not.

notes

The BACKUP/RESTORE facility in PrimeBase is designed to guarantee complete recovery of a database including changes applied to the database after the backup was completed.

Backup of a database can be done while the database is online (i.e. while it is in normal use), using the BACKUP DATABASE statement. A backup of a database can be restored using the RESTORE DATABASE statement. The backup image of a database looks identical to the normal database image. However, if the backup was made while the database was in use, then the image may not be consistent, due to the fact that the tables were copied at different times. If the database was not in use during backup, the backup image can be mounted as any other database, provided the backup is the first of the database in that location. If this is done, the mounted database will reflect the state of the database at the time of the backup, and will not include any subsequent changes. It may be necessary to mount a backup image if any of the log file at the time of backup has been lost or corrupted.

By default, offline logs are deleted by the server. Offline logs are logs no longer needed by the server to do a normal restart (recovery). In order to bring a database completely up-to-date from a backup, the offline logs must be archived. To do this, the system administrator must set the system variable **OfflineFunction** to "Archive", as follows:

```
OPEN DATABASE master;
SET VARIABLE offlinefunction = "Archive";
CLOSE DATABASE;
```

To set the offline log function back to deletion, set OfflineFunction to "Delete". When the offline location function is set to Archive, log files are not deleted, but copied to a log archive location. An archived log is given a different name (the first letter of the log name is changed). The restore function will only look for an archived log in the archive location to which it was copied. In order for restore to succeed, all required archive logs must be available. This means that all volumes containing archive logs must be online.

example

```
/***In this example, the database Golfers is backed up.***/

BACKUP DATABASE Golfers;
```

BACKUP TABLE

function — This statement does a backup of a table.

syntax — `BACKUP TABLE <table_reference> ';'`

parameters

BACKUP TABLE	statement identifying keywords.
<table_reference>	name of the table you want to backup.

notes — The backup of the table is added to the last backup done of the database. The backup options and locations used are those specified in the original database backup command. It is necessary to backup a table after it has been reorganized (REORG TABLE).

CLOSE DATABASE

function — This statement closes a currently open database.

syntax

```
CLOSE DATABASE [<database_alias>]';'
<database_alias>::= <identifier>
```

parameters

CLOSE DATABASE	statement identifying keywords.
<database_alias>	an identifier. If no alias was specifically given in the ALIAS clause of OPEN DATABASE, then the default database is closed. An alias must conform to the rules for identifiers. See “Identifiers” , page ***.

notes — When a database is opened it becomes the current default database. If the default database is closed it is not possible to determine which open database will become the new default database (unless there is only one open database left). The USE DATABASE statement below should be used to reset the default database. You can first use the statement DESCRIBE OPEN DATABASES, which lists the default database.

example

```
/***In this example the database “Golfers” is closed. In OPEN
DATABASE, “Golfers” was assigned the alias “G”, which is then
used in this CLOSE DATABASE statement.***/

CLOSE DATABASE G;

/***In this example, the same as above is achieved, but in two
steps rather than one. Notice that the alias is not included in
the syntax of the close statement.***/

USE DATABASE G;
CLOSE DATABASE;
```

see also backup database, restore database, open/use database, create/drop database, mount/unmount database

CLOSE DBMS

function This statement closes an open DBMS. If <dbms_brand> is not given, then the current DBMS is closed. All the open databases of the DBMS are also closed by this statement.

syntax

```
CLOSE [<dbms_brand>] DBMS ';'
```

parameters

CLOSE DBMS	keywords.
<dbms_brand>	name of dbms brand to be closed.

examples

```
CLOSE DBMS;

CLOSE MyServer DBMS;
CLOSE 'PrimeBase Server' DBMS;
```

notes In PrimeBase the DAL concept of DBMS is equated with that of server or gateway. An “open dbms” is, therefore, an open connection to a server. The command CLOSE DBMS closes the connection.

Note that the name of the DBMS must be placed in quotes if it contains spaces or special characters.

see also open dbms, use dbms, describe dbms, describe open dbms

CLOSE TABLE

function This statement closes a table. PrimeBase, however, ignores this statement, as a table is automatically closed at the end of the transaction in which the table was opened.

syntax
```
CLOSE TABLE <table_reference>';'
```

parameters

CLOSE TABLE	keyword.
<table_reference>	the name of the table being closed.

example
```
/***In this example, the table, “Golfers” is closed. Please re-
member, however, that this has no meaning in PrimeBase, as the
table is automatically closed at the end of the transaction in
which it was opened.***/

CLOSE TABLE Golfers;
```

see also create/drop/rename table, alter table, backup table, reorg table, check table, open table

COMMENT ON

function This statement allows you to place a comment on any type of object and on columns. The type of object may be specified, but it is optional.

syntax
```
COMMENT ON (<object_comment> | <column_comment>) IS <expression>
';'

<object_comment>::= [<object_type>] <object_reference>

<object_type>::= DOMAIN | TABLE | KEY | DEFAULT | INDEX | RULE |
VIEW | VARIABLE
```

```
<column_comment>::= COLUMN <column_of_table_reference>
```

parameters

COMMENT ON	statement identifying keywords.
<object_comment>	specifies the object to which the comment is bound.
<object_type>	an optional specification of the type of the object.
<object_reference>	the name of the object.
<column_comment>	specifies the column to which the comment is bound.
COLUMN	this keyword is required when placing a comment on a column.
<column_of_table_reference>	the qualified name of a simple or composite column. The syntax is explained at the end of this Reference section, page 103.
<expression>	A string (value of type CHAR or VARCHAR) that is the comment text to be placed on the object/column.

notes A previous comment may be removed by specifying the comment as an empty string, (“”).

examples

```
/***In this example, a comment is added to the Courses table.***/

COMMENT ON TABLE Courses IS “Each club has a number of courses. The details of each hole of each course are stored in the Courses table.”
```

see also describe dbms, describe open dbms, describe databases, describe open databases, describe tables, describe linksets

CREATE DATABASE

function This statement creates the necessary system folders and files for a new database.

syntax

```
CREATE DATABASE <database_name> {<file_location_spec>} ';'

<file_location_spec>::= [DATA|INDEX] [IN] LOCATION
<character_literal>
```

parameters

CREATE DATABASE	statement identifying keywords.
<database_name>	a unique name for the database.
<file_location_spec>	an optional location specifications for data and/or index files.
DATA	keyword specifies that path name for data files follows.
INDEX	keyword indicates that path name for index files follows.
IN	optional keyword.
LOCATION	indicates that path name follows.
<character_literal>	path name.

notes

Only system administrators (SA) and master database users with DBA status may create a database.

The new database is created but not opened.

The creator of the database is entered as the second user of the database with DBA privileges. The first user of a database is the user “System”. ”System” is the creator and owner of the system tables and other system objects. When a system administrator opens a database in which he is not a user, he is then considered to be the user, “System”.

In creating a database two file system locations may be specified. The location for the data (DATA keyword in the IN LOCATION clause), and the location for the indices (INDEX keyword). These locations may be the same (i.e. both DATA and INDEX keywords may be omitted. If no location is specified, then location for both data and index is the DataServer root path by default. The dataserver root path is given when installing the dataserver, and contains the Master and Model databases. The dataserver will append a directory to the specified location, and then place the data/index files within that directory. The name of the directory is identical to the name of the database.

A database name must be an identifier, whether specified as a character string or not. Since the name of the directory containing the database files is identical to that of the database, the names of databases are limited as for directory names of the underlying operating system. For example, a dataserver running under DOS would only support database names of maximum 8 characters in length. However, the system ensures that the names of databases are case insensitive like all other identifiers.

New databases are created by duplicating the model database. The SA is able to configure created databases by modifying the model database.

Warning: Do not remove, rename, or delete any files or directories created by the server. If you wish to delete a database, use the DROP DATABASE statement. IF you want to change the location of a database, you can use the MOUNT and UNMOUNT command.

examples

```
/***In this example, the database “Golfers” is created.***/

CREATE DATABASE Golfers ;
```

see also restore database, backup database, open/close/use database, drop database, mount/unmount database

CREATE DEFAULT

function This statement is used to specify a value that will be automatically inserted into a column, if no value is explicitly supplied at insert time.

syntax

```
CREATE DEFAULT <default_reference> ON ( [ COLUMN ]
<column_of_table_reference> | DOMAIN <domain_reference> )
<default_def> ';'

<default_def>::= AS ( <expression> | USER | SERIAL
<variable_reference> | NOW )
```

parameters CREATE DEFAULT statement identifying keyword.

<default_reference>	qualified name of default.
ON	identifying keyword; indicates that you are specifying the column, or simple domain to which the default value will be bound.
COLUMN	optional identifying keyword; indicates that the default is to be bound to a column.
<column_of_table_reference>	the qualified name of a simple column.
DOMAIN	keyword; indicates that you want to bind the default to a simple domain.
<domain_reference>	the qualified name of a simple domain.
<default_def>	definition of the default.
AS	keyword.
<expression>	an expression that is evaluated when the default is created to produce a literal value.
USER	The USER function returns the name of the current user in the case of character columns, or the database user identifier in the case of numeric columns.
SERIAL	The SERIAL function returns the next in sequence of a particular data type.
<variable_reference>	the name of a variable of type, counter, that has already been defined, see CREATE VARIABLE, page 46.)
NOW	this keyword returns the current time or date.

notes

Defaults may be specified on a simple column or a simple domain. If a default is placed on a domain, a further default may still be specified on a column defined on that domain. This default takes priority over the domain default. If no default value is stated, the value is recorded as missing (NULL). If the column does not allow missing values an insert in which the column value is not specified is rejected. On insert the column default takes priority.

The default value must be compatible with the data type of the column or domain to which it is bound.

examples

```
/***In this example, a default is created on the domain, GolferID, called GolferDef. It is a SERIAL default, based on the counter variable, called GolferCnt.***/

CREATE COUNTER INTEGER GolferCnt = 1;
CREATE DOMAIN GolferID INTEGER NOT NULL;
CREATE DEFAULT GolferDef ON DOMAIN GolferID AS SERIAL GolferCnt;

/***In this example, a default, ParDef, is created on the column, Par, of the table Courses. A default value of 4 is always inserted into this column.***/

CREATE DEFAULT ParDef ON Courses.Par AS 4;
```

see also drop/rename default

CREATE DOMAIN

function The domain manipulation statement, create domain, allows the declaration of a user-defined, extended data type, which is distinct from any other domain within the database (simple or composite).

syntax

```
CREATE [ PRIMARY ] DOMAIN <domain_reference> (
<simple_domain_def> | <composite_domain_def> ) ';'

<simple_domain_def>::= <data_type> { [ ',' ]
<domain_specification> }

<domain_specification>::= <missing_specification> |
<arithmetic_specification> | <order_specification>

<missing_specification>::= [ NOT ] NULL

<arithmetic_specification>::= ARITHMETIC [ NOT ] APPLICABLE
```

```
<order_specification>::= ORDER [ [ NOT ] APPLICABLE ] [ AS <se-
quence> ]

<sequence>::= COLLATING SEQUENCE <variable_reference> |
<system_sequence> { ',' <system_sequence> }

<system_sequence>::= COMMON | CASE INSENSITIVE | IGNORE DIACRIT-
ICAL MARKS

<composite_domain_def>::= '(' <simple_domain> ','
<simple_domain> { ',' <simple_domain> } ')'

<simple_domain>::= <domain_reference> | <data_type>
```

parameters

CREATE [PRIMARY] DOMAIN
: statement identifying keyword;.

<domain_reference>
: the identifying name of the domain you are creating.

<simple_domain_def>
: the actual definition of a simple domain; a domain that is based on a single basic data type.

<data_type>
: a basic data type.

<domain_specification>
: either a missing specification, an arithmetic specification, or an order specification.

<missing_specification>
: an indication of whether values in a column based on a domain may be missing. The default is: missing values permitted.

[NOT] NULL
: NULL means that values may be missing; NOT NULL indicates that values may not be missing.

<arithmetic_specification>
: the arithmetic specification indicates whether arithmetic operations ('*', '/', '+', '-' , etc.) are allowed on the domain. The default is: arithmetic not permitted.

<order_specification>
: See notes for a full explanation of this clause.

ORDER APPLICABLE	declares that the comparison operators; '<', '>', '<=' and '>=' can be meaningfully applied to the extended data type being defined. The default is: order not applicable.
NOT	negates the above statement, (i.e.the comparison operators cannot be meaningfully applied).
<variable_reference>	a system variable of type collating sequence. System variables of this type may be created by the user using the CREATE VARIABLE statement. See notes for an explanation of the different types of system defined sequences.
<system_sequence>	There are three system defined sequences. See notes for details.
<composite_domain_def>	a domain defined on a combination of simple domains
<simple_domain>	either the name of an existing simple domain, or a basic data type.
<domain_reference>	the name of a previously declared simple domain.

notes

A domain in its definition stores information as to its basic data type, whether values of the domain are allowed to be missing, and information as to whether the comparative and arithmetic operators can be meaningfully applied. (The operators = and ≠ can always be meaningfully applied.)

The range of values permitted on a domain may be specified by placing a rule on the domain, (see CREATE RULE).

By default, the missing_specification should be set as NULL in the case of a non-primary domain, and NOT NULL in the case of a primary domain.

By specifying that a domain is primary (CREATE RPIMARY DOMAIN), the user indicates that values in primary keys defined on that domain must be domain-wide unique. For example, it is possible to create a number of primary keys that draw values from a common domain. The system ensures that the sets of values in various primary keys on a primary domain are disjoint. If the domain is not primary, uniqueness of primary key values on the domain are only ensured within

the table on which the primary key is defined. Note that if there is only ONE primary key on a domain then it makes no difference whether the domain is primary or not.

Please remember, however, that when we talk about a primary domain, we do not mean a domain that has a primary key defined on it, but we mean a domain that has been explicitly declared as primary..

Domains cannot be declared recursively, in terms of one another.

Composite domains may include basic data types as well as simple domains as components. Domains cannot be declared recursively, in terms of one another.

The order of the components of a composite domain is significant in that, when sorting values in a domain, the left-most component is considered the most important. This means that if a domain is ordered, its components are sorted from right to left, and compared left to right.

Note that <domain_reference> in <simple_domain> is the name of a previously declared simple domain.

The order specification statement declares the following information:

1. Whether or not the operators '<', '>', '<=', or '>=' can be meaningfully applied to values of the domain (ORDER APPLICABLE). If order is not applicable, only equality tests, (equals ('='), not-equals ('!=')), may be done when comparing two values of the domain. Sorting, however, is still possible.

2. That a collating sequence should be used when comparing values of the domain. The name of the collating sequence may be explicitly specified (<variable_reference>), or a system defined collating sequence can be selected using the COMMON, CASE INSENSITIVE... keywords.

Collating Sequences

COMMON: The common ordering is an improved ordering of the ASCII character set, which places alphabetically similar characters together, and upper case before lower case:

AÄaääâBbCÇcç. . .

CASE INSENSITIVE: In a case-insensitive sequence, the case of the characters is ignored. In such a sequence, A=a, and Ä=ä, etc..

IGNORE DIACRITICAL MARKS: When ignoring diacritical marks, A=Ä, and a=ä, for example.

example

```
/***In this example a series of domains, defaults and variables
are created, to provide a structure for the table, “Golf-
ers”.***/

CREATE COUNTER INTEGER GolferCnt = 1;
CREATE DOMAIN GolferID INTEGER NOT NULL;
CREATE DEFAULT GolferDef ON DOMAIN GolferID AS SERIAL GolferCnt;

CREATE DOMAIN NameType VARCHAR[55] ORDER APPLICABLE AS CASE IN-
SENSITIVE;

CREATE DOMAIN StatusType CHAR[8];
CREATE RULE StatusRule ON StatusType AS StatusType IN ( 'Ama-
teur', 'Pro', 'Pro/Am' );

CREATE DOMAIN HandicapType SMINT;

CREATE RULE HandicapRule ON HandicapType AS HandicapType BETWEEN
36 AND -5;

CREATE COUNTER INTEGER ClubCnt = 1;
CREATE DOMAIN ClubID INTEGER;
CREATE DEFAULT ClubDef ON DOMAIN ClubID AS SERIAL ClubCnt;

CREATE TABLE Golfers
(ID GolferID NOT NULL,
SurName NameType NOT NULL,
FirstNames NameType NOT NULL,
Name (SurName, FirstNames),
Title CHAR[10],
Sex CHAR[1] NOT NULL,
Nationality NameType,
DateOfBirth DATE,
Status StatusType,
```

```
Handicap HandicapType,
MemberOfClub ClubID,
Earnings MONEY[12,2]
);
```

see also drop/rename domain

CREATE GROUP

function This statement creates a group, or number of groups, within a database.

syntax `CREATE GROUP <group_name> { ',' <group_name> } ';'`

parameter

CREATE GROUP	statement identifying keywords.
<group_name>	name of the group you are creating. This must be a character literal, and must be enclosed in quotation marks.

notes The group will be created in the database that you are currently working in.

example

```
/***In this example, a group is created called GolfersPros, containing the names of professional golfers, who use this database.***/

CREATE GROUP “GolfersPros”;
```

see also create user, alter user, add user, remove user, grant, revoke, drop group

CREATE INDEX

function This statement is used to create an index on a column, group of columns or a domain.

syntax

```
CREATE <index_spec> <index_reference> ON [TABLE]
<table_reference> <column_group> ) ';'

<index_spec>::= INDEX { SUPPRESS ZERO | SUPPRESS NULL }
```

parameters	CREATE	statement identifying keyword.
	<index_spec>	this specifies the definition of the index.
	<index_reference>	the qualified name of the index.
	ON	identifying keyword; introduces the clause indicating the object on which the index is created.
	<table_reference>	the qualified name of the table on which the index is to be created.
	<column_group>	an ordered list of columns (simple and/or composite), on which the index is to be created.
	INDEX	identifying keyword; indicates that an index is being created.
	SUPPRESS ZERO	identifying keyword indicating that these values are to be excluded from the index.
	SUPPRESS NULL	identifying keyword; indicates that these values are to be excluded from the index.

notes

Indices are a performance related feature, for example, they can speed up data retrieval, but can slow down data insert and update.

If duplicate key values occur very often, the speed of data retrieval will not increase, and update/delete will slow down.

The fewer rows there are that match the search conditions, the more effective the index will be.

Remember that is zero depression has been defined on an index, a search on that index will **not** retrieve any zeros.

examples

```
/***in this example, an index is created, called “GolfersIndex”,
and is defined on the table called “Golfers” - on the column, ID.
The primary key for the table is also defined on this column. It
is recommended that you define your indices on the same columns
as your primary keys.***/

CREATE INDEX GolfersIndex ON Golfers (ID);
```

```
/***In the next example, an index is defined on a composite col-
umn. In this case, each separate component of the composite
column must be listed. You may not simple give the name of the
composite column itself.***/

CREATE INDEX CoursesIndex ON Courses (Club, Course, Hole);
```

see also drop/rename index

CREATE KEY

function The key manipulation statement, CREATE KEY, defines a primary, candidate or foreign key on a base relation.

syntax

```
CREATE <key_spec> <key_reference> ON [ COLUMN ]
<column_of_table_reference> [<reference_spec>];

<key_spec> ::= UNIQUE | ( ( PRIMARY | CANDIDATE | FOREIGN ) KEY )

<reference_spec>::= REFERENCES <table_reference> { ','
<table_reference> } { <triggered_action> }

<triggered_action>::= ON UPDATE <referential_action> | ON DELETE
<referential_action>

<referential_action>::= RESTRICT | CASCADE | SET NULL | SET DE-
FAULT
```

parameters

CREATE	keyword.
<key_spec>	key specification, either primary, candidate or foreign.
<key_reference>	the qualified name of the simple or composite key you wish to define.
UNIQUE	alternate keyword to specify the definition of a candidate key.
PRIMARY, CANDIDATE, FOREIGN	a full explanation of these keywords is the user manual in chapter KEYS.

COLUMN	identifying keyword; indicates that the key you are creating is to be bound to a column. This keyword is optional.
<column_of_table_reference>	the qualified name of a simple or composite column. Syntax specifications - page 103.
<reference_spec>	an optional specification of the target table(s) of the foreign keys. Each target table must have a primary key defined on the same domain as the foreign key. Only one of the tables is required to contain the corresponding primary key value.
REFERENCES	keyword.
<table_reference>	the qualified name of a target table.
<triggered_action>	indicates that after certain commands performed on any of the target tables, a particular function is to be carried out.
ON UPDATE	keywords, indicating that when an update is carried out on the target table, referential action must be taken!
ON DELETE	keywords, indicating that when a delete is carried out on the target table, referential action must be taken.
<referential_action>	from a list of functions, you can specify what happens to foreign key values that no longer have corresponding primary key values.
RESTRICTED	The update or delete operation is restricted to the case where there are no related values (it is otherwise rejected).
CASCADES	The update, or delete operation “cascades” to update the foreign key in all related values.
SET NULL	On update, or deletion, the foreign key is set to null in all related values and the target record is then updated, or deleted (of course, this case could not apply if the foreign key cannot accept nulls in the first place).

SET DEFAULT	On foreign key columns that have a default bound to them, on an update, or delete, operation to the primary key column, the foreign key is then updated to the default value - set by the CREATE DEFAULT statement.

notes

Foreign keys may only be defined on a column that is based on a previously defined domain (simple or composite).

All primary keys must fulfil the entity integrity rule, which states: no component of the primary key in a base relation is allowed to contain a NULL. When a primary key is defined on a column which allows NULLs, the column will no longer accept missing values.

All primary and candidate keys must satisfy the uniqueness property, which states: No two tuples of a key may have the same value, therefore although it is allowed for candidate keys to be defined on a column that allows NULLs, this columns may only include one NULL, as two NULLs are considered as duplicate values.

All composite candidate and primary keys must satisfy the minimality property, which states: If a candidate key is composite, then no component of that key can be removed from that combination without the uniqueness of that key being lost. However, adherence to this requirement of the relational model cannot be verified by the DBMS.

All foreign keys must fulfil the referential integrity rule, which states: The database may not contain any unmatched foreign key values. These values are all drawn from the primary key which is being referenced, via the primary domain on which the foreign key is based.

A base relation must have one and only one primary key defined on it.

A primary key must be defined on a base relation before the relation can be used.

example

```
/***in this example, a primary key is created, GolfersPk, and
defined on the column, ID, in the table, Golfers.***/

CREATE PRIMARY KEY GolfersPk
ON Golfers.ID;
```

```
/***A candidate key is defined on the same table, on the column,
Names. This column is a composite column, made up of the compo-
nent columns, SurName and FirstName.***/

CREATE CANDIDATE KEY GolferNameCk ON Golfers.Name;

/***A foreign key is created on the Clubs table, and is defined
on the column Professionals. There is no need to explicitly
define the reference to the primary key, as the foreign key is
defined on the same domain as the primary key - therefore a re-
lationship between the two keys is automatically created. You
can however set up triggered action, independent of the REFER-
ENCES clause.***/

CREATE FOREIGN KEY ClubsProFk ON Clubs.Professional ON UPDATE
CASCADE ON DELETE CASCADE.
```

see also drop/rename key

CREATE RULE

function Rules can be applied to tables or simple domains. They restrict the values and combinations of values of a row of a table or a simple domain.

syntax

```
CREATE RULE <rule_reference> ON ( [TABLE] <table_reference> |
[DOMAIN] <domain_reference> ) <rule_def> ';'

<rule_def>::= ( CHECK | AS ) <search_condition>
```

parameters

CREATE RULE	statement identifying keyword.
ON	specify to which table or domain the rule is to be bound.
<rule_reference>	the qualified name of the rule.
<table_reference>	the qualified name of the table to which you want to bind the rule.

<domain_reference> the qualified name of the domain to which the rule is bound. Rules can only be specified on simple domains.

<rule_def> specifies the conditions of the rule.

<search_condition> can be any expression that would be valid in a WHERE clause. Subqueries, however, are not allowed.

notes

If a base table is dropped, all rules defined on that table are also dropped.

In order for a new rule to be defined on a given table, the old rule must first be dropped.

When a rule is defined on a table, existing rules are not checked to conform to the rule. Only subsequent inserts and updates are checked.

Columns or components of columns referenced in <search_condition> are limited to the columns of <table_reference> (a single row). Note that only the comparison operators are defined on composite columns

Rules can only be specified on a simple domain (i.e. a domain defined on a single basic data type). In the case of rules defined on domains, the name of the domain (optionally qualified by the creator name) may be used in place of columns in <search_condition>.

examples

```
/***In this example, a rule, StatusRule is defined on the do-
main, StatusType.***/

CREATE DOMAIN StatusType CHAR[8];
CREATE RULE StatusRule ON StatusType AS StatusType IN ( 'Ama-
teur', 'Pro', 'Pro/Am' );

/***A rule, ParRule is defined on the column, Par, in the table,
Courses.***/

CREATE RULE ParRule ON Courses AS Par IN ( 3, 4, 5 );
```

see also

drop/rename rule

CREATE TABLE

function — This statement creates a relation within a currently open database.

syntax

```
CREATE TABLE <table_reference> <table_def> ';'

<table_def>::= '(' <column_def> {',' <column_def> } ')'

<column_def>::= <column_name> ( <simple_column_def> |
<composite_column_def> )

<simple_column_def>::= <simple_domain>
[<missing_specification>]

<simple_domain>::= <domain_reference> | <data_type>

<composite_column_def>::= <column_group> [<domain_reference>]

<column_group>::= '(' <column_name> { ',' <column_name> } ')'
```

parameters

CREATE TABLE	statement identifying keywords.
<table_reference>	the qualified name of the relation you are creating.
<table_def>	the definition of the relation being created, which is a list of column definitions.
<column_def>	the definition of a column of the relation, which consists of a column name followed by a simple or composite column specification.
<column_name>	the identifying name for a particular column..
<simple_column_def>	the definition of a simple column, specifying the type or domain of the column, and whether or not this column may contain NULLS (missing_specification>).
<simple_domain>	the domain from which a simple column draws its values. This must be an already existing simple domain (see section on domains), or a basic data type.
<data_type>	any one of the basic data types. See section on data types, in the user manual in chapter DATA TYPES..

<composite_column_def>
: the definition of a composite column, consisting of two, or more, simple columns, optionally based on a composite domain.

<column_group>
: an ordered list of simple columns that comprise the composite column.

<domain_name>
: the composite domain on which the composite column is based.

notes

A column can either be simple or composite. A simple column is defined using a simple domain (<simple_domain> in <simple_column_def>). Please note however, that <simple_domain> also allows the direct specification of a basic data type. This means that the user is not required to declare a domain for every column in the database, and also renders PrimeBase compatible with other database management systems.

A composite column is a combination of simple columns.

If a composite domain is specified in the declaration of a composite column, it is not required that the simple columns mentioned in the <column_group> have been previously declared, as the simple column definition can be deduced from the composite domain that follows, (<domain_name> in <composite_column_def>).

The <missing_secification> for simple columns declared in this way is as per default. If the <missing_specification> in the definition of a simple column is omitted, the column will assume the <missing_specification> of the underlying domain, or missing value allowed (NULL) in the case of simple columns defined on a basic data type. If the underlying simple domain is defined as NOT NULL, then the column cannot be defined as NULL.

Please note, that a simple column may be a member of more than one composite column.

A table may not be used until a primary key has been defined on it. Equally, when a primary key is dropped from a base relation, that relation is temporarily disabled until a new primary key has been defined.

A CHAR or VARCHAR defined column with NULLs allowed takes up much more space than a VARCHAR or CHAR column where NULLs are not allowed.

examples

```
/***A table is created, called Results. Two of the columns are
defined on the domains, CompetitionID, and GolferID. A composite
column has been created, called Key. The primary key is defined
on this column, as is the index.
```

```
CREATE TABLE Results
(
Year SMINT NOT NULL,
Competition CompetitionID NOT NULL,
Place SMINT NOT NULL,
Key ( Year, Competition, Place ),
Golfer GolferID,
TotalScore SMINT,
Points SMINT,
Winnings MONEY[10,2]
);
```

```
CREATE PRIMARY KEY ResultsPk ON Results.Key;
CREATE FOREIGN KEY ResultGolferFk ON Results.Golfer;
CREATE INDEX ResultsIndex ON Results (Year, Competition, Place);
```

see also drop/rename table, alter table, reorg table, backup table, check table, open/close table

CREATE VARIABLE

function This statement is used to create a database variable.

syntax

```
CREATE (<collating_sequence> | <user_counter> | <user_variable>
) ';'
```

```
<collating_sequence>::= COLLATING SEQUENCE [VARIABLE]
<variable_reference> '=' <comparison_order>
```

```
<comparison_order>::= '('<equivalent_sequence> { ','
<equivalent_sequence> } ')'

<equivalent_sequence>::= '(' <expression> { ',' <expression> }
')' | <expression>

<user_counter>::= COUNTER [VARIABLE] <data_type>
<variable_reference> [ '=' <expression> ]

<user_variable>::= VARIABLE <data_type> <variable_reference> [
'=' <expression> ]
```

parameters

CREATE — keyword.

<collating_sequence> — specifies how character string values are to be compared and sorted.

COLLATING SEQUENCE — identifying keywords: indicate that a collating sequence variable is to be created.

[VARIABLE] — an optional keyword.

<variable_reference> — a name for the variable you are creating. It must conform to the rules for identifiers.

<comparison_order> — an ascending list of <equivalent_sequence>s.

<equivalent_sequence> — specifies that all characters in the sequence are considered to be equal for comparison purposes.

<expression> — any valid expression, which is interpreted as a single or string of characters.

<user_counter> — a database variable that can be used to generate unique identifiers. By using a counter as a serial default (see CREATE DEFAULT, page 30) the current value of the counter may be automatically inserted into a column, and then the counter variable incremented.

COUNTER [VARIABLE]
identifying keywords: indicate that a variable of the counter type is being created.

<data_type> a numeric data type, such as INTEGER, FLOAT or DECIMAL.

<variable_reference> a name for the variable. It must conform to the rules for references, page 103.

<expression> any valid expression. This clause is optional. If you do not set an expression, the counter will start at zero, (0). An expression can also later be set with the SET VARIABLE statement.

<user_variable> a user defined variable.

VARIABLE statement identifying keyword: indicates what kind of a database structure is being created.

<data_type> any of the basic data types.

<variable_reference> a name for the variable. It must conform to the rules for references, page 103.

<expression> any valid expression. This clause is optional. If no expression is set here, this can be done later with the SET VARIABLE statement.

examples

```
/***In the following example, a collating sequence variable is
created, called “normal”. It defines the order for sorting and
comparison purposes.***/

CREATE COLLATING SEQUENCE VARIABLE normal =
(
'AÀÄÃÅaáàâäãå',
'Ææ',
'Bb',
'CÇcç',
'Dd',
'EÉeéèêë',
```

```
'Ff',
'Gg',
'Hh',
'IiíìîÏ',
'Jj',
'Kk',
'Ll',
'Mm',
'NÑnñ',
'OÖÕoóòôöõ',
'Œœ',
'Øø',
'Pp',
'Qq',
'Rr',
'Ssß',
'Tt',
'UÜuúùûü',
'Vv',
'Ww',
'Xx',
'Yyÿ',
'Zz',
'0',
'1',
'2',
'3',
'4',
'5',
'6',
'7',
'8',
'9'
);
```

```
/***Here are some examples of how the equivalent sequence
'AÀÄÃÅaáàâäãå' could have been written:***/

('A', 'À', 'Ä', 'Ã', 'Å', 'a', 'á', 'à', 'â', 'ä', 'ã', 'å' ),
or
('AÀÄÃÅ', 'aáàâäãå' ),

/***Note that the characters that are omitted (there are 256
characters, although many are not printable) from the collating
sequence are added automatically, each in there own equivalent
sequence in order of ASCII numbers.***/

/***In the next example, three counter variables are created.
These counter variables are referred to in later default state-
ments.***/

CREATE COUNTER INTEGER GolferCnt = 1;
CREATE COUNTER INTEGER ClubCnt = 1;
CREATE COUNTER INTEGER CompetitionCnt = 1;

/***In the last example, a user defined variable is created,
called “year_end_no”. It is of the data type, DATE, and the ex-
pression is the date of the year - “1992”.***/

CREATE VARIABLE DATE year_end_no = “1992”;
```

notes

A database user variable can be used to store certain values used by an application permanently in the database.

The order of characters within an <equivalent_sequence> is important when sorting

Each expression in an equivalent sequence represents a character or sequence of characters. Strings (CHAR, VARCHAR, BIN and VARBIN) represent a sequence of characters. Other numbers (INTEGER, FLOAT,...) represent single characters. The number is considered to be the ASCII value of a character (e.g. 65 = 'A', 66 = 'B', etc.).

All the characters in an <equivalent_sequence> are considered to be equal for comparison ('=', '<', '>', BETWEEN and LIKE) purposes. The order of characters within an <equivalent_sequence> is important when sorting (e.g. by the

ORDER BY clause). In this case, characters are sorted ascending from left to right. For example, the <equivalent_sequence> “Aa” indicates that “A” and “a” are equal for comparison purposes, but when sorting, “A” will appear before “a”.

While an <equivalent_sequence> consists of characters that are considered equal when compared, the <comparison_order> indicates the result of '<', '<=', '>' and '>=' operations performed on characters from different <equivalent_sequences>.

The only way of finding the current value of a database variable is by selecting the value from the SysVariables table.

see also set/drop/rename variable

CREATE VIEW

function The statement create view enables you to create a virtual, derived table. Any table that can be retrieved via a SELECT statement (any derivable table) can be defined as a view.

syntax
```
CREATE VIEW <view_reference> [ <column_group> ]
AS <query_spec>
[ WITH CHECK OPTION ]';'
```

parameters

CREATE VIEW	statement identifying keyword.
AS	specifies the mapping of that object to the conceptual level.
WITH CHECK OPTION	indicates that update and insert operations are to be checked, to ensure that they satisfy the view-defining condition. It is optional.
<view_reference>	the qualified name of the view being created.

	<column_group>	optional; list of unique column names, should two or more columns of the view otherwise have the same name, or if view is derived from a function, operational expression, or a literal, and thus has no name that can be inherited.
	<query_spec>	query specification; the SELECT statement that defines the view.

example

```
/***In this example, a view, GolfersAmateurs is created. It consists of columns from the table Golfers, and includes all golfers with the status Amateur.***/

CREATE VIEW GolfersAmateurs
AS SELECT ID, Name, Title, Handicap, Status
FROM Golfers
WHERE Status ="Amateur"
WITH CHECK OPTION;
```

notes

Views are dynamic, meaning that changes to the underlying table will automatically and immediately be reflected in the view. Equally, changes made to the view itself are also applied to the underlying relation(s).

It is strongly advisable to always include the primary and/or candidate key of the underlying relation(s) in your view definition. This will ensure that it is possible to update (INSERT, UPDATE, DELETE) the view.

see also

drop/rename view

DESCRIBE COLUMNS

function

This statement describes all columns of a particular table. The table must be accessible to the user and must be in an already open database. The resulting table is described below:

syntax

```
DESCRIBE COLUMNS [OF] <table_reference> [INTO <cursor>] ';'
```

return values

col#	Data Type	Name	Description
1	SMINT	colnr	Column number
2	SMINT	level	Column level number
3	VARCHAR[31]	name	Column name
4	SMINT	type	Column data type
5	SMINT	len	Column length in bytes
6	SMINT	places	Column scale
7	BOOLEAN	nullsok	Nulls allowed
8	BOOLEAN	groupcol	Group column
9	SMINT	parentnr	Parent column number
10	SMINT	occurs	Number of occurrences
11	SMINT	occdep	Occurs depending on column
12	BOOLEAN	updtok	Is column updatable
13	VARCHAR[31]	title	Column title
14	VARCHAR[255]	remarks	Column remarks

parameters DESCRIBE COLUMNS keywords.

OF an optional keyword

<table_reference> a reference to a table.

	INTO	an optional clause: rowset can be put into a specific cursor. If no cursor is mentioned, the rowset is placed into the system defined cursor, $cursor.
	<cursor>	A cursor variable to receive the rowset. It must conform to the rules for identifiers.

notes It is not necessary for the table specified by <table_reference> to have been previously opened with an OPEN TABLE statement.

examples
```
/***A describe columns of the table Golfers is carried out in
this example.***/

DESCRIBE COLUMNS OF Golfers;
PRINTALL;

1 0 ID 3 4 0 $FALSE $FALSE 0 0 0 $TRUE
2 0 SurName 12 55 0 $FALSE $FALSE 0 0 0 $TRUE
3 0 FirstNames 12 55 0 $FALSE $FALSE 0 0 0 $TRUE
4 0 Title 9 10 0 $TRUE $FALSE 0 0 0 $TRUE
5 0 Gender 9 1 0 $FALSE $FALSE 0 0 0 $TRUE
6 0 Nationality 12 55 0 $TRUE $FALSE 0 0 0 $TRUE
7 0 DateOfBirth 6 4 0 $TRUE $FALSE 0 0 0 $TRUE
8 0 Status 9 8 0 $TRUE $FALSE 0 0 0 $TRUE
9 0 Handicap 2 2 0 $TRUE $FALSE 0 0 0 $TRUE
10 0 MemberOfClub 3 4 0 $TRUE $FALSE 0 0 0 $TRUE
11 0 Earnings 11 12 2 $TRUE $FALSE 0 0 0 $TRUE
```

see also describe dbms, describe open dbms, describe databases, describe open databases, describe tables, describe linksets

DESCRIBE DATABASES

function This statement returns a list of all databases on a specific DBMS (server or gateway).

syntax
```
DESCRIBE [<dbms_brand>] DATABASES
```

```
[[IN] LOCATION <character_literal>]
[INTO <cursor>]';'
```

return values The rowset returned into <cursor> is as follows:

col#	Data Type	Name
1	VARCHAR[31]	name

parameters DESCRIBE DATABASESkeywords

<dbms_brand> specify which brand of DBMS. This parameter is optional. If given, it must be the name of a previously opened DBMS (see OPEN DBMS command). The default is the current DBMS as selected by the USE DBMS command.

IN an optional keyword. The IN LOCATION clause is ignored by the PrimeBase server.

LOCATION an optional clause, to specify the location of the databases.

<character_literal> In this case, the path name of the databases in the form of a string. This parameter is ignored by the PrimeBase server, due to the fact that all databases are listed in the SysDatabases table in the Master database. When accessing a non-PrimeBase server through a gateway, however, the use of this parameter depends on the type of DBMS (brand).

INTO an optional clause: rowset can be put into a specified cursor.

<cursor> A cursor variable to receive the rowset. Must conform to the rules for identifiers.

notes — A PrimeBase server can deliver information other than just the names of the databases. This information includes the ID of the databases and the privilege level of the user. The PINKCTL statement is used to control what information is provided by DESCRIBE DATABASES. By default, only the names of the databases are listed in order to maintain DAL compatibility.

example

```
/***In this example the databases of the default DBMS are
listed.***/

DESCRIBE DATABASES;
PRINTALL;

Master
Model
test
SysTestMaster
SysTestTemp
Golfers
```

see also — describe dbms, describe open dbms, describe open databases, describe tables, describe linksets, describe columns, open database

DESCRIBE DBMS

function — This function returns the names of all DBMSs that can be accessed by the client.The result is a rowset, as described below.

syntax

```
DESCRIBE DBMS [INTO <cursor>]';'

<cursor>::= <identifier>
```

return values — The resulting rowset contains one row for each DBMS. Each row contains the 17 columns of information shown in the table below.

col#	Data Type	Name	Information
1	VARCHAR[31]	brand	DBMS name or server alias
2	VARCHAR[31]	rev	Version number
3	VARCHAR[31]	brparms	Brand open parameters
4	VARCHAR[31]	dbparms	Database open parameters
5	VARCHAR[31]	tbparms	Table open parameters
6	VARCHAR[31]	struct	Database structure info
7	VARCHAR[31]	txns	Transaction support
8	VARCHAR[31]	types	Supported data types
9	VARCHAR[31]	stmts	Supported statements
10	VARCHAR[31]	queries	Query processing options
11	VARCHAR[31]	aggfcns	Aggregate function support
12	VARCHAR[31]	brtype	DBMS brand (brand type)
13	VARCHAR[31]	prot	Protocol that may be used to connect to the DBMS

col#	Data Type	Name	Information
14	VARCHAR[255]	options	The options that should be used to connect
15	VARCHAR[31]	zone	The zone (if applicable) of the DBMS
16	VARCHAR[31]	server	The actual server name
17	VARCHAR[31]	unused6	Reserved

parameters

DESCRIBE DBMS	statement identifying keywords.
INTO	an optional clause: rowset can be put into a specified cursor.
<cursor>	A cursor variable to receive the rowset. Must conform to the rules for identifiers.

example

```
/***In this example, the DBMS, PrimeBase, is described. The
printall statement is used to print the results.***/

DESCRIBE DBMS; PRINTALL;
SQLonPPC 0 NNN YNNNNNYN NYN YYYYYNNNN YYN YYYYYYYYYYYYYYNN
YYYYYYYYNNYY YYYYYYYYYY YYYYYYYY PrimeBase adsp \zDev-
Zone\bPrimeBase DevZone
SUNServer2400 0 NNN YNNNNNYN NYN YYYYYNNNN YYN YYYYYYYYYYYYYYNN
YYYYYYYYNNYY YYYYYYYYYY YYYYYYYY PrimeBase adsp \zDev-
Zone\bPrimeBase DevZone
SQLonPPC 0 NNN YNNNNNYN NYN YYYYYNNNN YYN YYYYYYYYYYYYYYNN
YYYYYYYYNNYY YYYYYYYYYY YYYYYYYY PrimeBase ppc \zDevZone\wPaul's
Quadra\bPrimeBase DevZone
IBMTestDBMS 0 NNN YNNNNNYN NYN YYYYYNNNN YYN YYYYYYYYYYYYYYNN
YYYYYYYYNNYY YYYYYYYYYY YYYYYYYY PrimeBase adsp \zDev-
Zone\bPrimeBase DevZone
```

```
PressSQL24 0 NNN YNNNNNYN NYN YYYYYNNNN YYN YYYYYYYYYYYYYYNN
YYYYYYYYNNYY YYYYYYYYYY YYYYYYYY PrimeBase adsp \zMyZone\bPrime-
Base MyZone
```

notes

Unlike standard DAL, the PrimeBase DESCRIBE DBMS command lists all DBMSs that are published on the network, not just those on a particular host. As a result, DBMS brand (the first column of the rowset described above) is, in fact, the DBMS name. The actual DBMS brand is given in column 12: brtype. After creating a session, many 3rd party DAL applications allow the user to select a DBMS. In doing this, they present a list of values in the 'brand' column. In the case of P:INK DAL, this is a list of server and gateways on the network accessible from the client.

Other details provided by the DESCRIBE DBMS command include the version number, 15 profile strings and connection information.

The connection information is a PrimeBase extension to standard DAL. All the information required to make a connection to a particular DBMS, using OPEN DBMS, is provided in the columns, brtype, prot, options and zone. In the following two sections we describe the profile strings and the connection information.

Profile Strings

The 15 DBMS profile strings provide a description of a particular DBMS brand. They tell which features are supported or not supported by the brand, what parameters are required or optional, and so forth. The strings are positional, with character positions numbered from 0 (the first one) to N-1, where N is the length of the string. In each position of each string, the character will either be a Y (meaning the feature is supported) or an N (meaning the feature is not supported).

BRPARMS

The profile string, brparms, specifies which parameters of the OPEN DBMS statement are relevant for the DBMS brand. It has three characters in the following positions:

Position	PrimeBase Value	Meaning
0	Y	Is the user name used?
1	Y	Is the password used?
2	Y	Is the option string used?

DBPARMS

The profile string dbparms specifies which parameters of the OPEN DATABASE statement are relevant for the DBMS brand. It has eight characters in the following positions:

Position	PrimeBase Value	Meaning
0	Y	Is the database name used?
1	N	Is the location used?
2	N	Is the user name used?
3	N	Is the password used?
4	N	Is the option string used?
5	N	Is SHARED mode supported?
6	Y	Is PROTECTED mode supported?
7	N	Is EXCLUSIVE mode supported?

TBPARMS

The profile string tbparms specifies which parameters of the OPEN TABLE statement are relevant for the DBMS brand. It has three characters in the following positions:

Position	PrimeBase Value	Meaning
0	N	Is SHARED mode supported?
1	Y	Is PROTECTED mode supported?
2	N	Is EXCLUSIVE mode supported?

STRUCT

The profile string struct specifies general structural information about how the DBMS brand organizes its databases and whether various database features are present or absent. It has nine characters in the following positions:

Position	PrimeBase Value	Meaning
0	Y	Does the DBMS support creation of multiple databases on a single host system? (If N, there is one system-wide database.)
1	Y	Are individual databases named? (If N, databases are unnamed.)
2	Y	Does the DBMS use host locations (directories, catalogues) to structure its databases?

Position	PrimeBase Value	Meaning
3	Y	Does the DBMS support concurrent access to multiple databases? (If Y, this DBMS brand supports multiple OPEN DATABASE statements; if N, only one database of this brand can be open at a time.)
4	Y	Does the DBMS support queries across different databases? (If Y, the FROM clause of a SELECT statement can include tables from multiple databases; if N, all tables must be from the same database.)
5	N	Are linksets present in databases of this brand? (If N, the DESCRIBE LINKSETS statement will always produce a rowset with no row.)
6	N	Are hierarchical columns present in databases of this brand? (If Y, column names can have the form a.b.c.)
7	N	Are repeating columns present in databases of this brand? (If Y, columns names can have the form colname[6].)
8	N	Are variable repeating columns present in databases of this brand?

TXNS

The profile string txns specifies the transaction-processing support provided by the DBMS brand. It has three characters in the following positions:

Position	PrimeBase Value	Meaning
0	Y	Does the DBMS support transactions (that is, is the DAL COMMIT/ROLLBACK mechanism supported?)
1	Y	Are transactions performed in repeatable-read (RR) mode? That is, can the client application be sure that data it has read during the current transaction will be identical if re-read before a COMMIT or ROLLBACK?
2	N	Are transactions performed in cursor-stability (CS) mode? That is, can the client application be sure only that data read through a single cursor is consistent? (Either CS or RR mode will be TRUE for a given DBMS, but not both.)

TYPES

The profile string types specifies which DAL data types can result from a database of the DBMS brand. Each position of the string corresponds to a single DAL data type. It has 16 characters in the following positions:

Position	PrimeBase Value	Meaning
0	Y	Does the DBMS generate NULL data?
1	Y	Does the DBMS generate BOOLEAN data?

Position	PrimeBase Value	Meaning
2	Y	Does the DBMS generate SMINT data?
3	Y	Does the DBMS generate INTEGER data?
4	Y	Does the DBMS generate SMFLOAT data?
5	Y	Does the DBMS generate FLOAT data?
6	Y	Does the DBMS generate DATE data?
7	Y	Does the DBMS generate TIME data?
8	Y	Does the DBMS generate TIMESTAMP data?
9	Y	Does the DBMS generate CHAR data?
10	Y	Does the DBMS generate DECIMAL data?
11	Y	Does the DBMS generate MONEY data?
12	Y	Does the DBMS generate VARCHAR data?
13	Y	Does the DBMS generate VARBIN data?
14	N	Does the DBMS generate LONGCHAR data?
15	N	Does the DBMS generate LONBIN data?

STMTS

The profile string stmts specifies which DAL statements are supported for databases of the DBMS brand. Each position of the string corresponds to a single DAL data-manipulation statement. It has 12 characters in the following positions:

Position	PrimeBase Value	Meaning
0	Y	Is SELECT statement supported?
1	Y	Is FETCH statement supported?
2	Y	Is DESELECT statement supported?
3	Y	Is searched UPDATE statement supported?
4	Y	Is positioned UPDATE statement supported?
5	Y	Is searched DELETE statement supported?
6	Y	Is positioned DELETE statement supported?
7	Y	Is INSERT statement supported?
8	N	Is LINK statement supported?
9	N	Is UNLINK statement supported?
10	Y	Is COMMIT statement supported?
11	Y	Is ROLLBACK statement supported?

QUERIES

The profile string queries specifies the features supported in DAL queries against databases of the DBMS brand. It has ten characters in the following positions:

Position	PrimeBase Value	Meaning
0	Y	Are select-list expressions supported? (If N, only column references and aggregate functions of column references can appear in a select list.)
1	Y	Are joins supported? (If N, the FROM clause of a SELECT statement can include only a single table.)
2	Y	Are row-selection criteria supported? (If N, there can be no WHERE clause in a SELECT statement.)
3	Y	Is grouping supported? (If N, there can be no GROUP BY clause in a SELECT statement.)
4	Y	Is group selection supported? (If N, there can be no HAVING clause in a SELECT statement.)
5	Y	Is sorting supported? (If N, there can be no ORDER BY clause in a SELECT statement.)
6	Y	Are subqueries supported? (If N, the IN (subquery) predicate, the EXISTS predicate, the quantified predicates, and comparison predicates with their associated subqueries are not supported.)

Position	PrimeBase Value	Meaning
7	Y	Are aggregate functions supported in a select list?
8	Y	Are aggregate functions supported in a HAVING clause?
9	Y	Are aggregate functions with outer references supported in subqueries?

AGGFCNS

The profile string aggfcns specifies which DAL aggregate functions are supported in queries against databases of the DBMS brand. It has eight characters in the following positions:

Position	PrimeBase Value	Meaning
0	Y	Is the COUNT(*) supported?
1	Y	Is the COUNT(DISTINCTx) function supported?
2	Y	Is the MIN(x) function supported?
3	Y	Is the MAX(x) function supported?
4	Y	Is the SUM(x) function supported?
5	Y	Is the SUM(DISTINCT x) function supported?
6	Y	Is the AVG(x) function supported?
7	Y	Is the AVG(DISTINCT x) function supported?

Since the PrimeBase Server only supports the PrimeBase DBMS, the table will have only one entry - describing the PrimeBase DBMS.

Connection Information

The connection information is a PrimeBase extension to DAL.

BRTYPE

This is the actual brand of the DBMS.The brand is given here, because the column called 'brand' (the first column of the rowset) contains the name/alias of the server or gateway. A particular brand may be selected by setting the global variable $dbmsbrand. Setting this variable to NULL lists all known DBMS brands that may be accessed by a PrimeBase client.Setting the variable to 'PrimeBase' will list only PrimeBase servers.

PROT

The protocol that may be used to connect to the server or gateway.If more than one protocol is possible, the server will be listed once for each. Possible values for this field are:

adsp	Apple Data Stream Protocol. The standard protocol for Appletalk networks. This protocol can only be used when connecting to a remote server. Use the ppc protocol to communicate with a local Mac server.
ppc	Apple's Program-to-Program Communication. Also known as program-linking. This protocol is mainly used to communicate with a server on the same Macintosh as a client, however it may also be used over the network. To enable remote clients to connect to a server using ppc, start program-linking and enable guest logins for linking programs.
tcp	Transport Control Protocol/Internet Protocol (TCP/IP).

	ipc	Inter-process Communications. A protocol that uses shared memory to communicate between processes on the same machine. Currently this protocol is only implemented for UNIX systems.

OPTIONS

Options are the connection specific parameters required by the OPEN DBMS command to make a connection to this particular DBMS.

ZONE

On AppleTalk networks, this is the zone of the server or gateway. If a lookup zone has been selected using the global variable $dbmszone, then this field contains no additional information.

SERVER

This is the actual server name as published on the network. This value may be different to the DBMS brand name or server alias appearing in column 1.

see also describe open dbms, describe databases, describe open databases, describe tables, describe linksets, describe columns

DESCRIBE LINKSETS

function This statement is included for DAL compatibility only.

syntax `DESCRIBE LINKSETS [ [OF] <database_alias>] [INTO <cursor>] ';'`

parameters DESCRIBE LINKSETS keywords.

	OF	an optional keyword.

	<database_alias>	the alias of the database being queried.
	INTO	an optional clause: rowset can be put into a specific cursor. If no cursor is mentioned, the rowset is placed into the system defined cursor, $cursor.
	<cursor>	A cursor variable to receive the rowset. It must conform to the rules for identifiers.

notes — It returns no rows, as there are no linksets in PrimeBase. They are not included, as they are not a feature of the relational model.

see also — describe dbms, describe open dbms, describe databases, describe open databases, describe tables, describe columns

DESCRIBE OPEN DATABASES

function — This statement returns a cursor containing information about the currently open databases. The structure of the table is given below.

syntax — `DESCRIBE OPEN DATABASES [INTO <cursor>]';'`

return values — The resulting rowset has one entry for each database that is currently open. The first database described in the rowset is the current default database. Each row contains the five columns of information shown below:

col#	Data Type	Name	Description
1	SMINT	order	Sequence number
2	VARCHAR[31]	alias	Database alias
3	VARCHAR[31]	brand	DBMS brand
4	SMINT	shrmode	Sharing mode
5	SMINT	updmode	Update mode

col#	Data Type	Name	Description
6	VARCHAR[31]	owner	current owner

notes

The rowset created by this statement has an EXTRACT mode, so the number of databases described is available through the $rowcnt system variable.

The sharing mode (shrmode) is reported as 1=SHARED, 2=PROTECTED, 3=EXCLUSIVE.

The update mode (updmode) is reported as 1=READONLY, 2=UPDATE, 3=SCROLLING, 4=EXTRACT mode.

Current Owner: This column is always an empty string, as there are no owners in PrimeBase.

example

```
DESCRIBE OPEN DATABASES;
```

```
0 Golfers PrimeBase 2 2
```

see also

describe dbms, describe open dbms, describe databases, describe tables, describe linksets, describe columns

DESCRIBE OPEN DBMS

function

This statement describes all currently open database management systems (DBMSs).

syntax

```
DESCRIBE OPEN DBMS [INTO <cursor>]';'
```

return values

a rowset which consists of a DBMS number followed by a row with the identical structure of the rowset returned by the DESCRIBE DBMS statement.

parameter

DESCRIBE OPEN DBMS
: statement identifying keywords.

INTO
: an optional clause: rowset can be put into a specified cursor.

<cursor> A cursor variable to receive the rowset. Must conform to the rules for identifiers.

notes This statement lists all open connections to DBMSs. The first row returned is the current DBMS (as specified in USE DBMS.

see also describe dbms, describe databases, describe open databases, describe tables, describe linksets, describe columns

DESCRIBE TABLES

function This statement returns a cursor describing the tables of a particular database.

syntax `DESCRIBE TABLES [ [ OF ] <database_alias> ] [ INTO <cursor> ] ';'`

return values The structure of the table returned into <cursor> is given below.

col#	Data Type	Name	Description
1	VARCHAR[255]	name	Table name
2	VARCHAR[1]	type	Table(T) or view (V)
3	BOOLEAN	ordered	Is table ordered?
4	SMINT	colcnt	Column count
5	INTEGER	rowcnt	Row count
6	SMINT	parentcnt	Parent count
7	SMINT	childcnt	Child count
8	VARCHAR[31]	title	Table title
9	VARCHAR[255]	remarks	Remarks
10	VARCHAR[255]	owner	Table owner

parameters	DESCRIBE TABLES	statement identifying keywords
	OF	an optional keyword. If this clause is omitted, the default database is used.
	<database_alias>	The alias of the database whose tables are to be described. If dbalias is omitted, the default database is used.
	INTO	an optional clause: the resulting rowset can be put into a user-specified cursor.
	<cursor>	A cursor variable to receive the rowset. Must conform to the rules for identifiers.

notes

The OPEN DATABASE statement must be used to open the database specified by <database_alias> (or the default database if the database alias was not specified) before the DESCRIBE TABLES statement can be used.

If the <database_alias> is omitted, the default database is used (refer to USE/OPEN DATABASE statements).

Note also that the DESCRIBE TABLES statement describes not just the user defined tables, but also the system tables of the currently open database.

example

```
/***In this example, the tables in the database Golfers are described. Golfers is the default database.***/

OPEN DATABASE Golfers;
DESCRIBE TABLES;
printall;

Clubs T $FALSE 9 0 0 9 $NULL $NULL Common

Competitions T $FALSE 9 0 0 9 $NULL $NULL Common

Courses T $FALSE 7 0 0 7 $NULL $NULL Common

Golfers T $FALSE 11 0 0 11 $NULL $NULL Common

GolfersAmateurs V $FALSE 6 0 0 6 $NULL $NULL Common

Results T $FALSE 7 0 0 7 $NULL $NULL Common

Scores T $FALSE 8 0 0 8 $NULL $NULL Common
```

```
SysColumnComps T $FALSE 7 0 0 7 $NULL $NULL System
SysColumnPrivs T $FALSE 11 0 0 11 $NULL $NULL System
SysColumns T $FALSE 15 0 0 15 $NULL $NULL System
SysDataTypes T $FALSE 6 0 0 6 $NULL $NULL System
SysDefaults T $FALSE 10 0 0 10 $NULL $NULL System
SysDomainComps T $FALSE 8 0 0 8 $NULL $NULL System
SysDomains T $FALSE 13 0 0 13 $NULL $NULL System
SysIndexComps T $FALSE 6 0 0 6 $NULL $NULL System
SysIndices T $FALSE 18 0 0 18 $NULL $NULL System
SysKeys T $FALSE 9 0 0 9 $NULL $NULL System
SysMembers T $FALSE 2 0 0 2 $NULL $NULL System
SysObjects T $FALSE 7 0 0 7 $NULL $NULL System
SysPrivileges T $FALSE 16 0 0 16 $NULL $NULL System
SysReferences T $FALSE 5 0 0 5 $NULL $NULL System
SysRules T $FALSE 6 0 0 6 $NULL $NULL System
SysTables T $FALSE 15 0 0 15 $NULL $NULL System
SysUsers T $FALSE 12 0 0 12 $NULL $NULL System
SysVariables T $FALSE 8 0 0 8 $NULL $NULL System
SysViews T $FALSE 14 0 0 14 $NULL $NULL System
```

see also describe dbms, describe open dbms, describe databases, describe open databases, describe tables, describe linksets, describe columns

DROP GROUP

function	This statement drops a specific group or groups from the database.	
syntax	`DROP GROUP <group_name> { ','<group_name> } ';'`	
parameter	DROP GROUP	statement identifying keywords.
	group_name	the name of the group you want to drop.
notes	Naturally, once a group has been dropped, the users who were part of it are no longer members of it.	
example	`/***In this example, the group called GolfersPros is dropped.***/`	
	`DROP GROUP GolfersPros;`	
see also	see create user, alter user, add user, remove user, grant, revoke	

DROP <OBJECT>

function	This statement allows the deletion of a database object from the database.	
syntax	`DROP [<object>] <object_reference> (',' <object_reference>) ';'`	
parameters	DROP	keyword
	<object>	the keyword of the object you are dropping. For example, if you are dropping a table, you may write: DROP TABLE. It is however optional whether you write in the keyword of the object or not.
	<object_reference>	the name of the object you wish to drop.
notes	Please note, that it is not possible to undo this statement.	
example	`/***In this example, a database is dropped. Please note, that first the database had to be closed, in order to be dropped.***/`	

```
CLOSE DATABASE Golfers;
DROP DATABASE Golfers;
```

GRANT

function — This statement grants object or command privileges to the specified users and groups.

syntax

```
GRANT ( <command_privileges> | <object_privileges> ) TO ( PUBLIC
| <user_name> { ',' <user_name> } ) [ WITH GRANT OPTION ] ';'

<command_privileges>::= DBA | SA | RESOURCE |

<object_privileges>::= ALL [ PRIVILEGES ] | (
<object_privs_spec> { ','<action> } ) ON <object_reference> [
<column_group> ] ]

<object_privs_spec>::= INSERT | DELETE | REFERENCE | SELECT |
UPDATE | EXECUTE
```

parameters

GRANT — statement identifying keywords.

<command_privileges> — determine which commands (or statements) a user is permitted to issue.

DBA — Database Administrator: this status means that a user in a particular database can perform any action on the database, without requiring specific privileges to do it. This user also has the ability to introduce new users to the database, and to grant DBA status to them. The DBA may also drop the database.

SA — System Administrator: SAs have DBA privileges to all databases controlled by the DataServer. SAs can create, alter, and delete any database. SAs do not need to be a user of a database in order to open a database.

RESOURCE	This means that a user may issue all CREATE commands. Note that this does not include the DROP command. This is because drop privileges are fixed as follows: Users can drop any object created by themselves and DBA can drop any object in the database. The ability to create databases is only given to users of the master database. Creating groups and adding user to a database may only be performed by the database DBA (or SA in the case of the master database).
<object_privileges>	determines which database objects (and columns) a user is permitted to access.
ALL	the user is permitted access to all those statements that are applicable to the object in question. The following privileges can be granted:
INSERT	permission to add a new row to a relation.
DELETE	permission to remove a row from a relation
REFERENCE	can only be granted on a domain, and allows the user to create a foreign key on that domain.
SELECT	permission to retrieve rows and columns from a relation or relations.
UPDATE	permission to update a row or column of a relation.
ON	indicates on which database object the user will be able to use these privileges.
<object_reference>	the qualified name of the object.
<column_group>	the specific references to those columns that are to be affected. If the column_group is not specified, then all columns are included in the privilege.
TO	which group or user, these privileges are to affect.

PUBLIC — All users are a member of the system group, PUBLIC. As a result, granting a privilege to PUBLIC grants the privilege to all users (or future users) of the database.

WITH GRANT OPTION — this clause enables the user, or group in question to also grant the privileges that have been granted. This clause is optional.

notes

Two types of privileges can be granted with this statement, command or object privileges. A command privilege determines whether a user is permitted to issue certain commands (or statements). Object privileges, on the other hand, associate privileges with specific objects (and columns) in the database.

DROP privileges are a special case. Users can drop any object that they themselves have created, and DBAs can drop any object in the database, including the database itself, but he/she may not drop any system objects.

When a user is added to the database, he/she is automatically given the lowest privileges possible. To add a user to a normal database, the database must be in use and the user must be a DBA. To add a user to the Master database, the master database must be in use and the user must be an SA. Users added to the Master database are not given SA status, but are allowed to create their own databases (they are called DBAs in the master database). A user of the Master database may be promoted to SA by granting SA privileges (SA in <command_privilege> above). Users of the master database who do not have SA privileges still need to be a user of a database to open the database. A user of the Master database that has SA privileges, however, is able to open any database and is automatically given DBA privileges in that database.

If <column_group> is not specified all columns are included in the privilege grant. A <column_group> may be specified in the case of INSERT, SELECT and the UPDATE privilege granting. DELETE and REFERENCE are always granted on an object level. The REFERENCE privilege can only be granted on a domain and means the user has the power to create a foreign key on that domain.

examples

```
/***In this example, the privilege, RESOURCE is granted to the
group called GolfersPros. This means that all users who are mem-
bers of this group are automatically assigned these privileges:
namely that they can issue all CREATE object commands.***/

GRANT RESOURCE TO GolfersPros;

/***In the following example, DBA privileges are granted to the
user Julian Baldock.***/

GRANT DBA TO “Julian Baldock”;

/***In the next example, the privilege to issue the SELECT
statement on the table Courses, specifically to select from the
columns, Key, and Description, has been granted to Heather
Fyson. She may also grant this privilege to other users - hence
the WITH GRANT OPTION clause. This clause does not apply to the
<command_privileges>.***/

GRANT SELECT ON Golfers (Key, Description) TO “Heather Fyson”
WITH GRANT OPTION;
```

see also see create user, alter user, add user, remove user, revoke, drop group

MOUNT DATABASE

function This statement allows a dataserver to register the existence and location(s) of a database that was created by another PrimeBase Server.

syntax

```
MOUNT DATABASE <database_name> {<file_location_spec>}';'

<file_location_spec>::= [ DATA | INDEX ] [ IN ] LOCATION
<character_literal>
```

parameters

MOUNT DATABASE	identifying keywords: register database structure
<database_name>	identifying name of database
<file_location_spec>	path name specifications

	DATA	keyword indicating that you want to specify the location for data.
	INDEX	keyword indicating you want to specify the location for indices.
	IN	optional keyword indicating that the location is about to follow.
	LOCATION	keyword indicating that the path name to the database follows.
	<character_literal>	the path name for the database that is to be mounted.

notes Do NOT mount a database that is already mounted by another server. Use the UNMOUNT statement before you mount the database to a new server, or make a copy of the entire database (when the server is not running).

Restore is not possible for a newly mounted database until a backup has been done.

If no location for data and indices is specified, then their location is by default the dataserver root path. The dataserver root path is given when installing the dataserver. The dataserver will append the name of the database to the specified location(s). It will expect to find the index and data files of the database at these locations.

example
```
/***In this example, the database Golfers is mounted. No loca-
tion is specified, as this is not strictly required.***/

MOUNT DATABASE Golfers;
```

see also create database, restore database, backup database, open/close/use database, drop database, unmount database

OPEN DATABASE

function This statement opens a database on the host.

syntax

```
OPEN [<dbms_brand>] DATABASE [<database_name>]

[ALIAS <database_alias>]
[[IN] LOCATION <character_literal>]
[[AS] USER <character_literal> [[WITH] PASSWORD
<character_literal>]]
[FOR [<shared_mode>] [<access_mode>]]';'

<database_name>::= <character_literal>

<database_alias>::= <identifier>

<shared_mode>::= SHARED | PROTECTED | EXCLUSIVE

<access_mode>::= READONLY | UPDATE
```

parameters

OPEN	keyword.
<dbms_brand>	an identifier specifying a DBMS alias of a previously opened connection.
DATABASE	keyword.
<database_name>	a character literal specifying the database name.You will find the rules for identifiers page 103. DAL syntax allows the database name to be optional (as indicated in the syntax), but since PrimeBase simply requires the name, leaving the database name out causes an error. Other DBMSs do not require the name because they only support one database, or the database is identified by the location. The name of the database is used if the ALIAS clause is omitted - but only if the database name follows the rules for identifiers.
ALIAS	an optional clause. If no alias is specified, the database name is assumed as default.
<database_alias>	an identifier that is to be used as an alias for the open database. See page 103 for the rules on identifiers.

The syntax after and including the IN LOCATION clause is accepted but ignored.

notes

If specified, the dbms_brand must be PrimeBase.

The LOCATION, USER, and PASSWORD clauses are ignored with warning. The <user_name> and <password> used to access the database are those given when the connection was made to the host.

The LOCATION clause is ignored, because in PrimeBase, the location of a database has already been specified in the CREATE DATABASE statement. This information is remembered and stored with that database entry in the system table, SysDatabases. There is, therefore, no need to repeat this information. When you remount a database, and the location of data and index files is thus changed, the change of location is noted in the SysDatabases table.

PrimeBase does not support the FOR clause in the OPEN DATABASE statement, however, if you have this syntax hardcoded into your program, this will not cause an error - PrimeBase simply ignores it.Databases are always open for protected update no matter what mode is specified.

When a database is opened, it becomes the current default database.

example

```
/*** In this example, the Golfers database is opened, and given
an alias “G”.***/

OPEN PrimeBasePrimeBasePrimeBase DATABASE Golfers ALIAS G;
```

see also

create database, restore database, backup database, close/use database, drop database, mount/unmount database

OPEN DBMS

function

This statement opens a DBMS.

syntax

```
OPEN <dbms_brand> DBMS

[[AS] USER <character_literal> [[WITH] PASSWORD <password>]]
[OPTION <character_literal>]';'
```

parameters

OPEN DBMS keywords

<dbms_brand> the name (or alias) of a server or gateway, as listed by the DESCRIBE DBMS command.

USER Keyword indicating the name of a user follows. This clause is optional. If not given, the value stored in this system variable $user is used.

<character_literal>
the name a user of the DBMS.

PASSWORD keyword indicating a password follows. If the clause is omitted, it is assumed the password is blank.

<password> the password of the user of the DBMS.

OPTION an optional clause used to specify connection specific options.

<character_literal>
an option string. Options may include information such as: the protocol to be used for the connection, the zone of the DBMS, the type of server or gateway, etc. All options required to make a particular connection are listed by the DESCRIBE DBMS command.

notes This statement opens a connection to a DBMS. That is, a PrimeBase server or gateway to some other DBMS brand. Standard DAL allows only DBMSs on the host computer to be opened. PrimeBase DAL extends this idea to include any server or gateway published on the network. The "host" in PrimeBase DAL is, in fact, the entire network. As a result, <brand_name> is not the actual brand of the DBMS, but rather the name of the server or server alias. Furthermore, when a PrimeBase DAL session is created, it is not actually necessary to supply a host name, user name and password, due to the fact that a session is automatically connected to the "host".

If a host name is given when creating a PrimeBase DAL session, it is considered to be the name of a DBMS, and an OPEN DBMS statement is done automatically. This has the effect that a session has an open DBMS immediately after creation.

From this it is also clear that PrimeBase DAL session can have connections to multiple servers (each server is an open DBMS). And, in addition, these servers can be located locally, or anywhere on the network.

see also describe dbms, close dbms, use dbms

OPEN TABLE

function This statement opens a particular table for use. The purpose of opening the table is to lock it for read-only or exclusive use.

syntax
```
OPEN TABLE <table_reference> [FOR [<shared_mode>]
[<access_mode>]] ';'

<shared_mode>::= SHARED | PROTECTED | EXCLUSIVE

<access_mode>::= READONLY | UPDATE
```

parameters

OPEN TABLE	keywords: open database structure.
<table_reference>	the name of the table you want to open.
FOR	an optional clause to specify how the table can be used when it is opened. You can choose from a combination of <shared_mode> and <access_mode>.
<shared_mode>	indicates whether or not data can be accessed concurrently by other users.
SHARED	data can be accessed by other users
PROTECTED	data can be read and updated concurrently by other users, but updates are not allowed to conflict.
EXCLUSIVE	other users are not tolerated when reading data, or updating data in the table.
<access_mode>	indicates whether the user intends to update or only read table data.

READONLY	data may only be read, and not updated.
UPDATE	whether or not updates may be carried out on a table.

notes

Tables need not be opened before accessing data in the table. In PrimeBase DAL there is also no performance gain in opening a table before use. The main reason for using OPEN TABLE is to gain table level readonly or exclusive locks during a transaction.

Not all combinations of sharing and access modes are supported by PrimeBase. The default mode is PROTECTED UPDATE, which means that the user can read and update data concurrently with other users, but updates are not allowed to conflict.

A further two modes are supported: PROTECTED READONLY and EXCLUSIVE UPDATE. PrimeBase treats EXCLUSIVE READONLY as PROTECTED READONLY. If the user really wants exclusive access to a table, then the table must be opened for EXCLUSIVE UPDATE. In this mode the user may read or update the table, and no other user is allowed to either read or update the table.

The keyword SHARED is not supported by PrimeBase - but we have a good reason for not supporting it. The definition of SHARED is: table access may be shared by other users who concurrently update the database and whose operations may cause interference with this user. It is not good that a user´s operations can be interfered with. Therefore PrimeBase automatically upgrades SHARED mode to PROTECTED.

The table is automatically closed at the end of a transaction. This means that to continue with exclusive access to a table, an OPEN TABLE statement must be issued after every BEGIN transaction.

examples

```
/***in the following example, the table “Golfers” is opened for
PROTECTED UPDATE. There is no need to explicitly enter the key-
words, as PROTECTED UPDATE is the default setting. It means that
the user can read and update data concurrently with other users.
Updates are not allowed to conflict.***/

OPEN TABLE Golfers;
```

```
/***in the next example, the table “Golfers” is opened for EX-
CLUSIVE UPDATE. This means that the user may read and update the
table, but no other user may read or update the table.***/

OPEN TABLE Golfers FOR EXCLUSIVE UPDATE;
```

see also create/drop/rename table, alter table, reorg table, backup table, check table, close table

REMOVE USER

function This statement removes users either from the database or from a specific group.

syntax
```
REMOVE USER <user_name> { ',' <user_name> } [ FROM <group_name>
] ';'
```

parameter

REMOVE USER	statement identifying keyword.
<user_name>	the name of a user you are removing.
FROM	introduces an optional clause that is used when removing users from a group.
<group_name>	the name of a group in the default database.

return values

OK	the user has been removed.

notes To use this statement, the users must be users of the default database, (i.e., the database last used or opened); the group must also be an existent group.

When a user is removed from the database, all privileges that he or she granted are also removed.

examples
```
/***In this example, the user, called “Heather Fyson” is removed
from the group called “GolfersPros”. She is still in the data-
base - just not in the group, “GolfersPros”.***/

REMOVE USER “Heather Fyson” FROM “GolfersPros”;
```

see also see create user, alter user, add user, grant, revoke, create group, drop group

RENAME <OBJECT>

function — This statement alters the name of an already existing object - such as a domain, a table, an index, etc., etc.

syntax

```
RENAME [<object>] <object_reference> TO <object_name> ';'
```

parameters

RENAME	keyword
<object>	an optional keyword specifying the type of object you are renaming, for example: DOMAIN, TABLE, KEY, etc..
<object_reference>	the name of the object to be renamed.
TO	keyword
<object_name>	an identifier that is the new name for the object.

example

```
/***In this example, the table Clubs is renamed to Golf-
Clubs.***/

RENAME TABLE Clubs TO GolfClubs;
```

REORG TABLE

function — This command performs a low-level reorganization of the table. The user requires exclusive update access to the table before the command may run, and if the user has not already opened the table, the exclusive access mode will be acquired automatically for him by the system. The function statement packs the data (eliminating spaces left in the data file due to previous deletions) and rebuilds the indices of the table. Only the DBA or the creator of a table may reorganize a table.

syntax

```
REORG TABLE <table_reference>';'
```

parameters

REORG TABLE	statement identifying keywords.
<table_reference>	name of table to be reorganized.

RESTORE DATABASE

function — This command restores a database from backup.

syntax

```
RESTORE DATABASE <database_name> { <restore_options> } ';'
<restore_options>::= FROM <expression> | <partial_restore>
<partial_restore>::= PARTIAL | UNTIL ( ERROR | LOG <log_spec> )
<log_spec> ::= '{' <expression> ',' <expression> '}'
```

parameters — RESTORE DATABASE identifying keywords

<database_name>	the name of the database to be restored.
<restore_options>	specification of various restore options
FROM	an optional clause, used to select the backup to be stored. By default, the most recent backup is restored.
<expression>	the identifier of the backup to be restored.
<partial_restore>	specifies a partial restore of the database.
PARTIAL	restore the database, ignoring errors that occur during the process.
UNTIL	indicates restore should stop at some point before the database is completely restored.
ERROR	restore the database until the first error occurs.
LOG	restore the database until a certain log file.
<log_spec>	specifies a log file, by it's restart number and identifier.

notes — In its simplest form, the restore command will restore any database using the previous backup and **all** the log files starting at the time of backup.

In order to bring a database up to date during restore, the RESTORE command requires access to the copies made of the database tables during backup (the backup image), and all log files written since the backup began. If a log file is missing, the restore statement cannot bring the database up to date beyond this point in time.

A database may be partially restore if a missing log, or an error prevents complete restore. A database must be recovered to be restored, as the restore process is atomic. This means, that if it fails it has no effect on the current state of the database.

A database may not be in use while it is being restored.

example

```
/*** In this example, the database, Golfers is restored, where
“n” is the backup identifier noted in the system table called,
SysBackups***/

RESTORE DATABASE Golfers FROM n;
```

REVOKE

function — This statement removes specific privileges from a user or group of users.

syntax

```
REVOKE ( <command_privileges> | <object_privileges> ) FROM (
PUBLIC | <user_name> { ',' <user_name> } ) ';'
```

parameter

REVOKE	identifying keyword; revoke privilege.
<command_privileges>	see GRANT statement for an explanation of command privileges.
<object_privileges>	see GRANT statement for an explanation of object privileges.
FROM	defines from whom or what the privileges are being taken away.
PUBLIC	revokes all privileges from the group “Public”.

<user_name> the name of the user in question.

example

```
/***In this example, the privileges that were assigned to
Heather Fyson are revoked.***/

REVOKE SELECT ON Golfers (Key, Description) FROM “Heather Fy-
son”;
```

see also create user, alter user, add user, grant, remove, create group, drop group

SERVER CHECKPOINT

function This command starts a full checkpoint of the server.

syntax `SERVER CHECKPOINT ';'`

parameter `SERVER CHECKPOINT` identifying keywords.

notes The checkpoint process flushes all cache pages, and then writes a checkpoint record to the log file.

SERVER COMMENT

function Display a message on the console of all connected workstations.

syntax `SERVER COMMENT <character_literal> ';'`

parameter

`SERVER COMMENT` identifying keywords.

`<character_literal>` the text of the message that will appear immediately on all user's consoles.

notes The server comment command is used to display a message on the screen of all online users. For example, the command may be used to inform the user that a database backup or reorganization is about to take place.

SERVER ERROR

function — This statement loads error information from the error manager into a cursor.

syntax — `SERVER ERROR [INTO <cursor>]';'`

parameters

SERVER ERROR	keywords.
INTO	an optional clause to place the returned rowset in a specific cursor.
<cursor>	A cursor variable to receive the rowset. It must conform to the rules for identifiers.

return values — This command returns a rowset containing information on file I/O errors that may occur during a query, or certain other commands. The rowset has the following structure.

col#	Data Type	Name	Information
1	SMINT	PrimaryError	The primary error code
2	SMINT]	SecondaryError	Additional error information
3	SMINT	SystemError	System specific error code
4	TIMESTAMP	Time	Time of error
5	VARCHAR[31]	FunctionName	Operation attempted
6	INTEGER	DevID	The device on which the error occurred
7	INTEGER	SeekPosition	Seek position of error

col#	Data Type	Name	Information
8	INTEGER	TransferSize	Byte transfer required
9	INTEGER	DatabaseID	The database in which the error occurred
10	VARCHAR[255]	FileName	The system file name in which the error occurred

PrimaryError is the primary error code of the error that occurred. If this is zero, no error has occurred, and in this case, all other columns will have the value NULL.

The system error is the error code provided by the system. The meaning of this error is dependent on the operating system on which the server is running.

The columns SeekPosition and TransferSize have significance depending on the value of the column FunctionName. In the table below, 'Yes' indicates that the value is significant to the operation, and '-' indicates that the value is not applicable.

FunctionName	SeekPosition	TransferSize
Read	Yes	Yes
Write	Yes	Yes
Grow	-	Yes
Seek	Yes	-
Flush	-	-
Create	-	-
Open	-	-

FunctionName	SeekPosition	TransferSize
Delete	-	-
Rename	-	-
Make Directory	-	-
Remove Directory	-	-

The file name on which the operation occurred is given in the last column of the rowset.

notes

The statement loads the details of the most recently occurred errors from the error manager on the server. This error information is global for the entire server, and not related to an individual session. Error can be traced to the files in which they occurred using this statement.

If no errors have occurred since startup, the statement will return no rows.

SERVER RESTART

function

Start the normal server startup sequence, which includes recovery of all databases.

syntax

```
SERVER RESTART [COLD] [<partial_restart>]
[<restart_location_spec>] ';'

<restart_location_spec>::= [IN] LOCATION [<character_literal>]
{',' [<character_literal>] }

<partial_restart>::= [PARTIAL] [WITHOUT <expression> { ','
<expression>} ]
```

parameters

SERVER RESTART	keywords: indicate that the server is to be restarted.
COLD	an optional keyword that indicates whether or not a cold start is permitted.

<partial_restart>	a partial restart will recover those databases for which no error occurs during the restart. Another form of partial restart allows the SA to select databases for which to omit the recovery procedure. The database identifiers are given in a list in the WITHOUT clause.
<restart_location_spec>	a clause to specify locations that are important to server restart.
[IN] LOCATION	optionally specify a certain paths (location in the host file system) that are required for recovery.
<character_literal>	a string specify a path. Up to three paths may be specified. The first two are paths of the restart files, and the last is the server root.
PARTIAL	keyword: initiates a partial restart.
WITHOUT	keyword introducing a list of database identifiers.

notes

This statement initiates the normal server restart sequence. The normal restart sequence begins automatically when the server is started. However if this fails, the server requires the intervention of the system administrator. If a normal restart fails, the system administrator should attempt to correct the problem and manually initiate the restart sequence using this command.

Details of the error that occurred during restart can be obtained from the error manager using the TRANS ERROR command. The system error number, file name, database identifier and other details are contained in the rowset returned by this command.

The restart process includes: (a) restarting the transaction manager, and recovering the master database, (b) recovery of all user databases that are set recover pending in the master database, (c) bringing the previous active log online, and (d) setting all system parameters as recorded in the master database.

For this purpose, the server searches for the following: firstly, a restart file (RESTART.SQL), which indicates in which log files, and where recovery should begin. Secondly, the restart process must locate the master database (this location is also known as the server root).

Once the master database has been recovered, it is opened, and the information concerning the location of the user databases is used to recover the user databases.

In general, the keyword COLD, indicates that the actual recovery process should be skipped. Using a cold start, databases can be brought online without recovery. This should only be done if there is no way the problem that prevents recovery can be fixed, because without recovery the server cannot guarantee the integrity of the database or prevent data loss. After a cold start, a database will probably need to be reorganized. If the integrity of the system tables has been compromised, it may not be possible to open the database. In this case the only possibility is to restore the database from backup.

A partial restart (the PARTIAL keyword) is in any case better that a cold start as it tells the server to ignore errors during recovery and to do what it can. The convenience of the partial restart carries with it the problem that you will not be quite sure what was done and what not. For example, if restart was failing because of insufficient disk space, partial restart could result in significant data loss.

Cold starting the server means that all databases are cold started. This is an extremely harsh measure if the error is only occurring in one database. Using the WITHOUT clause, you may indicate to the server to leave certain databases out of the recovery process. Once the server and most of the databases are on line, you may then attempt to correct the problem concerning the database. A database can be recovered separately using the RECOVER DATABASE command.

To prevent an inadvertent cold start, the server requires the administrator to remove any restart files before it will proceed with a cold start. As long as the server finds a file called RESTART.SQL, it will ignore the COLD keyword.

If the restart files or log files are lost of corrupted there is no other option but to cold start. As a result, and due to the fact that the restart file is vital for restoring the master database, the restart file can be duplicated by the server. The location of the restart files and the dataserver root are stored in the server environment

file ('PrimeBase Environment' on the Macintosh). If this information is lost or has changed, the paths may be specified in the LOCATION clause in the SERVER RESTART statement.

For security, the restart file, RESTART.LOG, may be duplicated. Normally, the location of the restart files is found where it is stored in the preferences file (in the Macintosh system, in the file "PrimeBase Preferences", in the Preferences folder). If no restart file is found, the normal server restart will fail.

SERVER RESTORE

function — This command restores the Master database from backup.

syntax

```
SERVER RESTORE [<restart_location_spec>] {<restore_options>}
';'

<restart_location_spec>::= [IN] LOCATION [<character_literal>]
{',' [<character_literal>] }
```

parameters

SERVER RESTORE — keywords: indicate that the server is to be restored.

<restart_location_spec> — an optional clause used to specify the location of the restart files and the server root.

<restore_options> — specifies the restore options. set RESTORE DATABASE for details.

notes — This statement restores the Master database from the most recent backup. The backup device(s) of the Master database must be online for the SERVER RESTORE statement to function correctly. When the restore is complete, only the Master database will be considered recovered. At this point a normal server restart should be attempted.

SERVER SHUTDOWN

function Shutdown the server application.

syntax `SERVER SHUTDOWN [<expression>]`

`[COMMENT <character_literal>] ';'`

parameters SERVER SHUTDOWN keywords: indicate that the server is to be shutdown.

<expression> an optional clause. In this expression state, in seconds, the time the server will wait before shutting down. If no expression is given, the server shuts down immediately.

COMMENT this is an optional clause, which allows the administrator to send a message to logged on users, to inform them that the server will shut down.

<character_literal> the text of the message that will appear immediately on all user's consoles.

notes On shutdown, any transactions still active are automatically rolled back. The alert that appears on the users screen, can be disabled on the client machine. If the expression is NULL the current shutdown sequence is cancelled.

SET VARIABLE

function This statement sets the value of an existing database variable.

syntax `SET VARIABLE <variable_reference> '=' <expression> ';'`

parameters SET VARIABLE identifying keywords

<variable_reference> the identifier of an existing variable

<expression> any valid expression

example

```
/***In this example, the variable, “YearEndNo” is set to a new
value. It was originally set at 1992 - in the CREATE VARIABLE
statement. In this statement, it is set at “1993”.***/

SET VARIABLE YearEndNo = “1993”;
```

see also create/drop/rename variable

TRANS ERROR

function This statement returns details of the error that caused the transaction manager to go down.

syntax `TRANS ERROR [INTO <cursor>]';'`

parameters

TRANS ERROR	keywords.
INTO	an optional clause to place the returned rowset in a specific cursor.
<cursor>	A cursor variable to receive the rowset. It must conform to the rules for identifiers.

notes The details of the error the transaction manager to go down are returned from the error manager on the server. If the transaction manager is not down, this command will return no rows. The transaction manager goes down when it can no longer guarantee that transactions are atomic, or that committed data will be written to the database. This command can be used to determine more precisely the reason for the transaction manager going down. When the problem has been corrected, use the SERVER RESTART command to start the transaction manager.

TRANS RESTART

function Start the transaction manager and recover the master database.

syntax

```
TRANS RESTART [COLD | PARTIAL] [<restart_location_spec>] ';'

<restart_location_spec>::= [IN] LOCATION [<character_literal>]
{',' [<character_literal>] }
```

parameters

TRANS RESTART	keywords: start the transaction manager.
COLD	an optional keyword that indicates whether or not a cold start of the master database is permitted.
PARTIAL	initiates a partial recovery of the master database, during which errors are ignored.
[IN] LOCATION	optionally specify a certain paths (location in the host file system) that are required for recovery.
<character_literal>	a string specify a path. Up to three paths may be specified. The first two are paths of the restart files, and the last is the server root.

notes

This command is similar to SERVER RESTART, but performs only two steps of a normal server restart, namely: (a) the restarting the transaction manager, and recovering the master database, (b) bringing the previous active log online.

This command can be used to correct problems that occur when a user database is recovered, or when a system parameter is set. After TRANS RESTART, it is possible to open the master database and adjust parameters and databases locations.

After the transaction manager has be started, the server is effectively still in single-user mode, due to the fact that the communications will only accept remote connection after server restart. As a result, transaction restart should be followed at some stage by either a full or partial server restart.

see also

trans shutdown, server shutdown, server restart

TRANS SHUTDOWN

function

This command is used to shut down the transaction manager.

syntax **TRANS SHUTDOWN** `[<expression>] ';'`

parameters

TRANS SHUTDOWN	keywords: indicate that the transaction managers is to be shutdown.
<expression>	an optional clause. In this expression state, in seconds, the time the transaction manager will take to shut down. If no expression is given, the transaction manager shuts down immediately.

notes While the transaction manager is shutting down, transactions can only be committed. Attempts to begin a transaction will return an error.

see also trans restart, server shutdown, server restart

UNMOUNT DATABASE

function This statement removes the link that a server has to a database. The database can then be mounted by another server, or the locations of the database can be changed and the database remounted by the server.

syntax `UNMOUNT DATABASE <database_name> ';'`

return values

OK	the database has been successfully unmounted.

parameters

UNMOUNT DATABASE	keywords indicating that a database is to be unmounted by the current server.
<database_name>	the identifying name of the database to be unmounted.

notes Here is an example situation: Changing the index location of a database.

1. Make sure that no users have the database open.

2. UNMOUNT the database.

3. Create a directory in the target location, and give it the same name as the database.

4. Move the index files of the database to this directory. Index files have a .ind extension.

5. You may now delete the directory in which you found the index files, if it is empty.

6. MOUNT the database giving the new index location using the INDEX clause. (The data location must also be given if it not the DataServer root path.)

example

```
/***In this example the database, Golfers, is unmounted.***/

UNMOUNT DATABASE Golfers;
```

see also create database, restore database, backup database, open/close/use database, drop database, mount database

USE DATABASE

function The USE DATABASE statement establishes a particular database as the default database.

syntax

```
USE DATABASE <database_alias>';'

<database_alias>::= <identifier>
```

parameters

USE DATABASE	keywords.
<database_alias>	the database alias given to the database when it was opened.

example

```
/***In this example the database Golfers is to be set as the default database. In OPEN DATABASE it was given the alias “G”.***/

USE DATABASE G;
```

see also create database, restore database, backup database, open/close database, drop database, mount/unmount database

USE DBMS

function	This statement establishes a previously opened DBMS as the default DBMS.	
syntax	`USE <dbms_brand> DBMS ';'`	
parameters	USE DBMS	keywords
	<dbms_brand>	the name of the DBMS.
example	`USE MyServer DBMS;`	
see also	open dbms, close dbms	

IDENTIFICATION

This section provides you with a quick reference to the syntax that is required for identifiers, aliases and references. You will encounter these procedures in almost all data definition language statements.

IDENTIFIERS

An identifier is a sequence of characters, that defines a database object. Like keywords, case is not significant in identifiers. Maximum length is 31 characters. An identifier is unique within the database.

Identifiers are so called as they are used to identify the many different types of objects that may be created using DAL and the PrimeBase data definition language. This includes DAL variables, procedures and cursors, and PrimeBase procedures, databases, database objects (tables, domains, keys, etc.,) and columns.

The following are all identifiers.

```
<var_name>::= <identifier>
<cursor>::= <identifier>
<database_name>::= <identifier>

<dbms_brand>::= <identifier> | ':' <var_name> | <expression>
<object_name>::= <identifier> | ':' <var_name>
<column_name>::= <identifier> | [':'] <var_name>

<creator_name>::= <identifier> | ':' <var_name>
```

The following are object names:

```
<domain_name>::= <object_name>
```

An identifier is a sequence of characters, defined as follows:

syntax

```
<identifier> ::= ( <alpha> | <diac> | '_' | '$' ) { <alpha> |
<diac> | <digit> | '_' | '#' }

<alpha>::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'

<diac>::= Upper and lower case alphabetic characters with dia-
critical marks (e.g. ä, å, à, ...)
```

notes

The hash ('#') symbol is one of the characters of an identifier (however, not the first character). So a column 'supplier number' can be given the name: 'P#'.

The dollar ('$') symbol is only allowed as the first character of an identifier. Note that identifiers of the form $... are used by DAL to indicate system variables and functions.

<var_name> must be of data type OBJNAME, see the user manual in chapter DATA TYPES.

ALIASES

Aliases are names that may be different to the actual name of the object that they identify. Specifying an alias name is always optional. When not specified, the actual name of the object is taken as the alias. This is due to the fact that in the context in which an alias is used, the use of an alias is not optional. For example, databases are always referred to using an alias, whether an alias is specified in the OPEN DATABASE statement or not.

Aliases can be created for three types of objects: for databases, tables, and columns.

Database Alias

A database alias is created in the OPEN DATABASE statement. The database alias is either the name of the database or the identifier specified in the ALIAS clause. Each open database must be specified by a unique alias. The database alias is valid until the database is closed and may be used to refer to the database and objects and columns within the database.

syntax `<database_alias>::= <identifier>`

Table Alias

A table alias is created in a query specification, and is only valid within the query or any subqueries.

A unique table alias is required for every table in the query specification, and is the table name by default.

syntax `<table_alias>::= <identifier>`

Column Alias

A column alias is created in the select list of a query specification. The column alias may then be used to make cursor based references. The column alias is valid until the cursor is deselected, or until another rowset is selected into the cursor.

syntax `<column_alias>::= <identifier>`

REFERENCES

References fully identify database objects such as domains, tables, keys, etc..

Object Reference

An object reference is a specification that identifies a particular database object. In its complete form, an object reference consists of the database alias, the object creator name, and the object name. If the database alias is omitted from this list, then the default database is assumed. If <creator_name> is omitted, the creator name of the user of the database specified in database alias (or the default database) is assumed. If no such object exists, then the creator names "common" and "System" will be tried.

syntax

```
<object_reference>::= [ <database_alias> '!' ] [ <creator_name>
'.' ] <object_name> | [ ':'] <var_name>
```

The following are also object references:

```
<domain_reference>::= <object_reference>
<table_reference>::= <object_reference>
<key_reference>::= <object_reference>
<default_reference>::= <object_reference>
<index_reference>::= <object_reference>
<rule_reference>::= <object_reference>
<view_reference>::= <object_reference>
```

Column Reference

A column reference is a specification that identifies a particular column ion a database. In its complete form, it consists of the table alias and the name of the column. If <table_alias> is omitted the column reference is resolved by searching all possible tables in the query. Note that ambiguous references are not reported. The user may assume that the tables in a query are searched in the order mentioned in the FROM clause, beginning with the innermost nested subquery and moving outwards.

syntax

```
<column_reference>::= [<table_alias> '.' ] <column_name>
```

Column of Table Reference

A column of table reference specifies a column within a table, where the table in which the column is situated, is different to the table mentioned in the FROM clause. In its complete form, the <column_of_table_reference> consists of the database alias, the creator name of the table, the object name, and the column name itself.

syntax

```
<column_of_table_reference>::= [ <database_alias> '!' ] [
<creator_name> '.' ] <object_name> '.' <column_name> | ':'
<var_name>
```

DAL LANGUAGE REFERENCEPRIMEBASE

Data manipulation statements retrieve, modify and delete information stored in the tables of the open databases.

BEGIN

function — This statement simply initiates transaction processing. Transaction processing is only "on" after execution of this statement.

syntax — `BEGIN [ WORK | TRANSACTION | TRANS ] ';'`

parameters

BEGIN — keyword: indicates that transaction processing can start.

WORK | TRANSACTION | TRANS — These keywords are optional: only include them for readability.

BREAK

function — This statement causes a premature break in the flow of a loop structure - like a WHILE, DO, FOR, FOR EACH, or SWITCH statement. Execution then continues with the first statement that follows the body of statements in which the BREAK appears. The loop is not repeated after a BREAK statement.

syntax — `BREAK ';'`

parameters	BREAK	keyword: indicates that execution should continue with the first statement following the loop structure in which it appears.

notes When a BREAK appears within nested loops, or SWITCH statements, it will cause the processing of the innermost loop (or rather, SWITCH) to break. It is not possible to cause a multilevel break in a single statement.

CALL

function This statement calls a procedure. This statement can call either a procedure or a function. In this statement, the parameters specify which procedure is to be called, the values must be passed to the procedure as its arguments, and the variables that receive the return values from the procedure.

syntax

```
/***calling a procedure***/

[CALL] <procedure_name> ['(' <expression> { ',' <expression> }
')' ]
[RETURNING <var_name> {',' <var_name> } ] ';'

/***calling a function***/

<var_name> '=' <procedure_name> '(' <expression> { ','
<expression> } ')' ';'
```

parameters	[CALL]	an optional clause, containing the keyword, CALL meaning that a procedure is to be called.
	<procedure_name>	the identifying name of the procedure to be called.
	<expression>	any valid expression, specifying which values are to be passed as arguments to the procedure.
	RETURNING	keyword: indicates that the variables that follow are those that will receive the return values from the procedure. This clause is optional.

<var_name> the name of a variable that will receive the return values from the procedure. The number and type of variables must agree with the procedure declaration.

notes Procedure calls may be nested deeply. The depth of nesting is subject only to resource constraints on the run-time environment.

Recursive procedure calls are not supported.

COMMIT

function This statement signals that the current transaction has been successfully completed. Any modifications to the database made during the transaction are committed, and new transaction can begin.

syntax `COMMIT [WORK | TRANSACTION | TRANS] ';'`

parameters COMMIT keyword: indicates the successful completion of the current transaction.

WORK | TRANSACTION | TRANS
These keywords are optional: only include them for readability.

CONTINUE

function This statement interrupts the flow of the body of a WHILE, DO, FOR, FOR EACH statement, thus causing the remainder of the current iteration of the WHILE, DO, FOR, FOR EACH statement to be skipped.

Execution continues with the loop control expression (for WHILE or DO) or with the loop-reinitialiser (for FOR), or with fetching of the next row (for FOR EACH).

syntax `CONTINUE ';'`

parameters	CONTINUE	keyword: indicates that the flow of program is to be interrupted.

notes When CONTINUE appears within nested loops or SWITCH statements, the processing of the innermost loop (or SWITCH) is broken. It is not possible to cause a multi-level break in a single statement.

DECLARE

function This statement declares one or more variables to be a specific data type, before they are used in a program.When this statement is in a procedure definition, it declares a local variable for that procedure whose scope lies within the procedure definition. If the DECLARE statement is within the outer block, it declares an outer block variable whose scope is limited to the outer block.

syntax
```
[DECLARE] <data_type> <declaration> {',' <declaration>} ';'

<declaration>::= <var_name> ['=' <expression>]
```

parameters	DECLARE	keyword: indicates that a variable is being declared. This keyword is optional.
	<data_type>	any valid data type.
	<declaration>	the name and initial value of the variable.
	<var_name>	an identifier that is the name for the variable.
	<expression>	an expression defining an initial values for the variable. This part is optional.

notes The same name space is shared for both outer-block variable names and procedure names. A DECLARE statement for an outer block automatically redeclares any procedure or outer block variable with the same name.

If an initial value is present, an outer block variable is initialised at declaration; a local variable is reinitialised on each procedure entry.

On return from a procedure, local variables lose their value; note particularly that rowsets associated with local CURSOR variables are then deselected.

If an initial value is not specified, the variable remains uninitialised and produces an error if it is used in an expression before a value is assigned.

A DECLARE statement takes priority over any previous declaration for a variable of the same name within the same block.

DECLARE CURSOR

function — This statement declares a cursor. Tables retrieved from databases are stored in cursors. A cursor has a current row pointer which refers to a row in the table stored in the cursor. The various values of a column in the current row of a cursor may be accessed by a cursor based reference.

syntax — `[DECLARE] CURSOR <var_name> { ',' <var_name>} ';'`

parameters

[DECLARE] CURSOR	keywords indicating that a cursor is to be declared. The keyword DECLARE is optional.
<var_name>	an identifier for the cursor being declared.

DECLARE PROCEDURE

function — This statement defines a PrimeBase procedure. This procedure can then be invoked by the CALL statement.A procedure can be invoked by other procedures also using the CALL statement. A PrimeBase procedure takes zero or more arguments, which are passed on by value. It returns zero or more return values. Local variables can be declared for use within the procedure.

syntax

```
[DECLARE] PROCEDURE <procedure_name> '(' [<var_name>] {','
<var_name> } ')'

[RETURNS <data_type> { ',' <data_type> ';' ]
```

```
{ <decl> }
'{' <statement_list> '}'
END PROCEDURE <procedure_name> ';'

<decl>::= ARGUMENT <data_type> <var_name> [ '=' <expression> ]
{',' <var_name> [ '=' <expression> ] } ';'
```

parameters

[DECLARE] PROCEDURE — keywords: indicate that a procedure is to be declared. The keyword DECLARE is optional.

<procedure_name> — the identifying name for the procedure: it must conform to the rules for identifiers.

<var_name> — the name of a variable to be used within the procedure.

RETURNS — an optional clause, defining the data type for the values that are to be returned by the RETURN statement (see next page)

<decl> — a list of argument declarations.

ARGUMENT — keyword: indicates that an argument declaration follows.

<data_type> — the data type of the argument declaration.

<var_name> — the name of the variable, if used in the DECLARE PROCEDURE clause.

<expression> — any valid expression.

<statement_list> — any valid statements. If return values are expected, then this statement_list must contain a RETURN statement. See notes for more details.

END PROCEDURE — keywords: indicate that the procedure is terminated here.

<procedure_name> — the name of the procedure to be ended.

notes

Execution of a procedure ends usually when a RETURN statement is executed. If no RETURN statement is encountered in <statement_list> then an implied RETURN statement with no return value is executed. This will, of course, result in an error if you had specified a return data type in the head of the procedure

statement. Procedure names and outer block variables share the same name space. A new procedure statement automatically redeclares any previously declared procedure or outer block variable that had the same name.

DELETE (POSITIONED)

function — This statement deletes the current row of a cursor from the underlying table. The search condition indicates which rows are to be deleted.

syntax — `DELETE [FROM] <table_reference> [WHERE CURRENT OF <cursor>] [';']`

parameters

- DELETE [FROM] — keywords, indicating that data is to be removed from a database structure. The FROM keyword is optional.
- <table_reference> — the qualified name of the table.
- WHERE CURRENT OF — keyword: indicating the current row of the specified cursor. This clause is optional. If it is omitted, then all rows in the active cursor are deleted.
- <cursor> — the name of an active cursor with a valid current row that is then deleted by this statement.

notes — The cursor must specify an updatable table or view and must specify a rowset created by a SELECT statement including the <update_mode>.

The database may include referential-integrity constraints that prevent certain rows from being dropped. Attempts to drop these rows will result in an error.

Rows from a view may only be dropped, if the view is derived from a single table.

$rowsaffected is set after a successfull INSERT, UPDATE or DELETE. The value is the number of rows effected by the query. If an error occurs $rowsaffected is set to zero.

DELETE (SEARCHED)

function — This statement deletes rows from a table. The search condition specifies which rows are to be deleted. Warning, rows cannot be retrieved once they have been deleted.

syntax — `DELETE [FROM] <table_reference> [WHERE <search_condition>] [';']`

parameters

DELETE [FROM]	keywords, indicating that data is to be removed from a database structure. The FROM keyword is optional.
<table_reference>	the qualified name of the table.
WHERE	keyword: indicating which row or rows are to be deleted. This clause is optional. If it is omitted, then all rows in the table_reference are deleted.
<search_condition>	the search condition that identifies the rows to be deleted. If this is not included, then all rows are deleted.

notes — The database may contain referential integrity constraints that prevent certain rows from being dropped. Attempts to drop these rows will result in an error.

Rows from a view may only be dropped if the view has been derived from a single table.

$rowsaffected is set after a successfull INSERT, UPDATE or DELETE. The value is the number of rows effected by the query. If an error occurs $rowsaffected is set to zero.

DESELECT

function — This statement ends access to a rowset. The specified rowset is discarded, and all associated resources are reclaimed. The cursor associated with the rowset is then invalid.

syntax — `DESELECT [<cursor>] ';'`

```
<cursor>::= <identifier>
```

parameters	DESELECT	keyword: indicates that access to rowset is to be ended.
	[<cursor>]	an optional clause. The name of the cursor variable containing the cursor created by the SELECT statement that created the rowset. If this clause is omitted, the default cursor ($cursor) is assumed.

notes Because PrimeBase supports cursors the allow all movement throughout a rowset, the rowset cannot be automatically discarded when FETCH statement moves beyond its last row. The rowset must be explicitly deselected, otherwise it will take up host resources (memory and disk space) for the length of the current session.

DO

function This statement executes repeatedly and terminates through a post-test, when this evaluates to FALSE. With each repetition, the DO statement executes a specified statement, and then evaluates the specified expression. If the expression evaluates as TRUE, then the cycle of statement execution followed by expression evaluation continues. When the expression eventually evaluates to FALSE, the flow then passes on to the next statement after the DO statement.

syntax

```
DO <statement> WHILE '(' <expression> ')' ';'
```

parameters	DO	keyword: indicates that a cycle of statement followed by expression to be evaluated follows.
	<statement>	any valid statement. Often this statement is compound.
	WHILE	keyword: indicates that an expression follows that is evaluated as a test. If it produces a TRUE result, then the cycle of statement/expression continues. If it produces a FALSE result, flow passes on to the next statement after the DO.
	<expression>	any valid expression.

notes The statement is always executed once, even if the expression produces a FALSE result the first time round.

An expression that evaluates to NULL is FALSE.

ERRORCTL

function This statement controls the handling of errors during the execution of a session. If the <expr_value_expr> evaluates to zero, the DAL program terminates immediately as soon as any error is encountered. If <expr_value_expr> is non-zero, then when a data management error occurs, the system variable, $sqlcode, is set to the error number, and program execution continues. In this case the DAL program itself is expected to handle the error. Please note: It is assumed that a data-management error means all errors that involve data-definition (CREATE, DROP...) and data-manipulation (SELECT, UPDATE...).

syntax `ERRORCTL <expr_value_expr>';'`

parameters

ERRORCTL	keyword: indicates that an expression follows that when evaluated defines how error handling is to be directed.
<expr_value_expr>	an expression defining the direction of error handling.

notes The ERRORCTL statement governs error processing until another ERRORCTL statement supersedes it.

EXECUTE

function This statement passes a command to the Server for execution. The command to be executed is passed on as a text string in the EXECUTE command. There is also an EXECUTE FILE command - see over leaf for more details.

syntax `EXECUTE [IN <dbbrand>] <expression> ';'`

parameters	EXECUTE	keyword: indicates that the following expression is to be evaluated, compiled and executed.
	[IN <dbbrand>]	specify which DBMS brand you want. This clause is only included for compatibility with DAL. You can only choose one DBMS brand - namely PrimeBase. Therefore this clause is optional.
	<expression>	the expression must be of type CHAR or VARCHAR. It is evaluated, compiled and executed. Unlike DAL, any valid sub-program may be compiled and executed in this manner.
notes	Unlike DAL PrimeBase can handle output generated from the command string. Therefore it is possible to send SELECT statements and the like from via the EXECUTE statement.	

EXECUTE FILE

function	This statement loads and executes statements from a file. Procedure definitions are loaded and statements of other kinds are executed directly.	
syntax	`EXECUTE FILE <expression> [[ [IN] LOCATION <expression> ] ';'`	
parameters	EXECUTE FILE	keywords: indicate that statements in a specific file are to be executed.
	<expression>	this first expression is the name of a file and the second expression is the path to the file. Both expressions must be of type CHAR or VARCHAR.
	IN LOCATION	this optional clause specifies the path to the file.
notes	The sub-program in the file is loaded and executed. Sufficient memory is required to read the entire program into memory. Unlike DAL, the file executed resides on the workstation, NOT on the Server machine.	
	There is a limit to the length of each line in the file; This limit is 512 characters.	

If the file is not in the current folder, use the IN LOCATION clause to specify the folder. A path from the current folder may be specified by preceeding the path with a colon (':')

FETCH

function — This statement requests a single row from a rowset. PrimeBase selects the requested row as the current row of the given rowset. Cursor-based column references to columns in the rowset can later be used in expressions as identifiers that compare values of the specified column in the current row. As default, the FETCH statement simply retrieves the next row of the current rowset: the further optional parameters support movement of the cursor in other directions within the rowset, and permit concurrent processing of multiple rowsets.

syntax

```
FETCH [<motion> ] [ OF <cursor> ] ';'

<motion>::= FIRST | LAST | ABSOLUTE <expr_value_expr> | NEXT |
PREVIOUS | RELATIVE <expr_value_expr>
```

parameters

FETCH	keyword: indicates that a row from a given rowset is to be retrieved.
<motion>	specifies which record to retrieve. If <motion> is not specified, then FETCH NEXT is assumed by default.
FIRST	the first row of the rowset is retrieved.
LAST	the last row of the rowset is retrieved.
ABSOLUTE <expr_value_expr>	a 4 byte integer value expressing the absolute number of the row to be retrieved. If it is not a 4 byte integer, the values are automatically converted.
NEXT	the next row in the rowset in relation to the current row.

RELATIVE <expr_value_expr>
: a 4 byte integer value that specifies the row number to be retrieved relative to the current row (it may be positive or negative). If the value is not given as a 4 byte integer, the value is automatically converted.

notes

The rowset has no particular order, unless sort_list was specified in the SELECT statement that created the rowset.

When a rowset is first generated, the current row is the one just before the actual first row, so that a FETCH NEXT statement then moves it to the first row.

There is no current row of the rowset between the execution of an SELECT and the first FETCH statement; when FETCH moves on past the last row of the rowset; and if the current row is deleted.

If you attempt to fetch past the end of the rowset with either FETCH FIRST, LAST, NEXT or PREVIOUS, a $sqlcode value of $sqlnotfound is caused; for FETCH ABSOLUTE or RELATIVE, an error code is returned.

FOR

function

This statement performs repetitive execution initialised by an assignment, causing an expression to be evaluated, which if resulting as TRUE allows the specified statement to be executed. Then a loop reinitialisation assignment is called, and the whole process begins again. This cycle continues until the expression evaluates to FALSE. Flow then passes on to the next statement after the FOR statement.

syntax

```
FOR '(' [<init_assign>] ';' [<expression>] ';' [<init_assign>]
')' <statement>

<init_assign>::= <var_name> '=' <expression> | <var_name> '++'
| <var_name> '--'
```

parameters

FOR
: keyword: indicates that cycle of initialisation/expression/statement/reinitialisation follows.

	<init_assign>	an initialisation assignment, made up of a variable name and an expression.
	<var_name>	a variable name.
	<expression>	any valid expression.
	<statement>	any valid statement (usually a compound statement) to be executed if the expression is evaluated as TRUE.
notes	The initialisation assignment is always executed exactly once. If the expression produces a FALSE result at the first evaluation, then <statement> and the second initialisation assignment will never be evaluated.	

FOR EACH

function — This statement performs repetitive execution for each row in a rowset. This statement is a more convenient alternative to a WHILE loop containing a FETCH statement and $sqlcode testing. A specified statement is then executed for each row in the rowset, which is identified by the <cursor>.Cursor-based column references in the statement can be used to obtain the values of the columns in the current row for the iteration statement.

syntax

```
FOR EACH [<cursor>] <statement>
```

parameters	FOR EACH	keywords: indicate that a statement follows that is to be applied to each row of the rowset.
	<cursor>	the name of a cursor, identifying the rowset. This clause is optional.
	<statement>	any valid statement to be applied to each row in the rowset.

notes — The CONTINUE statement can start the next loop iteration prematurely.

If the retrieval of any row in the rowset fails, the FOR EACH iteration is terminated with an error. A BREAK statement can also prematurely terminate a FOR EACH iteration statement.

GOTO

function — This statement changes the flow of control in a program unconditionally. It is used in direct combination with the LABEL statement in that it causes an immediate fork to the particular statement labelled by the given statement label. (The statement label is defined in the LABEL statement.)

Both the GOTO and the LABEL statement cause a direct leap to a separate section of the program in order to perform any kind of processing. Please beware that overuse of the GOTO statement can mean that a program becomes needlessly complex, and difficult to understand and debug. Where possible, you should use the SWITCH statement.

syntax

```
GOTO <var_name> ';'
```

parameters

GOTO	keyword: indicates that a branch in the order of flow of program follows.
<var_name>	the identifier of a statement_label, previously specified by the LABEL statement.

notes — The GOTO statement may only appear within a procedure definition, and is also not permitted in the outer block. Please note that this is not a limitation in PrimeBase DAL. This is only limited by DAL.

A GOTO statement may leap out of a FOR, WHILE, DO, or SWITCH statement to break the flow of control. It is not recommended to leap into the middle of a FOR, WHILE, or DO loop, or a SWITCH statement.

IF

function — This statement executes DAL statements depending on a test condition. When the statement is executed, the specified expression is evaluated. If it produces a TRUE value, then the conditioned statement is executed. If not, the statement conditioned by the ELSE clause is executed.

syntax

```
IF '(' <expression> ')' <statement> [ELSE <statement>]
```

parameters	IF	keyword: indicates that a conditional statement follows.
	<expression>	an expression that is evaluated as the test condition.
	<statement>	the statement to be executed is the <expression> is evaluated as TRUE.
	ELSE <statement>	the statement to be executed is the <expression> is evaluated as FALSE. Note: an expression that is evaluated as NULL is FALSE.

INSERT

function This statement inserts a new row or new rows into a table.

syntax

```
INSERT [INTO] <table_reference> [(<column_group>)]
(<values_spec> | <query_spec> ) [';']

<values_spec>::= VALUES '(' [ <expression> | NULL ] { ',' [
<expression> | NULL ] } ')' [ <return_row> ]

<return_row>::= RETURNING [<column_group>] [INTO <cursor>]
```

parameters	INSERT [INTO]	keywords: indicate that new rows are to be added to the specified table. INTO is an optional keyword.
	<table_reference>	the qualified name of the table to receive the new rows.
	<column_group>	the group of columns to which new data is to be added. This is an optional part of the syntax. If it is omitted then all columns of the table are assumed. The order in which the data values are listed in the VALUES clause must then follow the order in which the column were created in CREATE TABLE.
	<values_spec>	clause in which the new data value are listed.
	VALUES	keyword: indicates that new values follow.

<expression>	any valid expression. If the value conflicts with the definition of the column, then an error will occur.
NULL	a NULL value
<return_row>	with <return_row>, you can retrieve a copy of the selected columns of the inserted row into a cursor. If the update was successful, the cursor will contain one row of data.
RETURNING	the keyword indicating that you want to retrieve a copy of the selected columns.
<column_group>	a comma-separated list of those columns you would like retrieved.
INTO	keyword indicating that the selected columns are to be put into a cursor.
<cursor>	the name of a cursor, into which the selected rows can be put.
<query_spec>	query specification. See the section entitled, "Query Specification" for full details as to syntax. Here, if present, indicates that the query should be carries out, and the results of the query be inserted into the specified table (<table_reference>). The number of columns returned by <query_spec> must correspond to the number of column specified in <column_group>.

notes

If <column_group> is omitted, all columns of the table are assumed in the order they were originally defined.

The number of expressions must correspond to the number of columns specified in <column_group>. Alternatively, the number of columns returned by <query_spec> must correspond to the number of columns specified in <column_group>.

$rowsaffected is set after a successfull INSERT, UPDATE or DELETE. The value is the number of rows effected by the query. If an error occurs $rowsaffected is set to zero.

LABEL

function — This statement, in combination with the GOTO statement unconditionally changes the flow of control in a program. It associates a statement label with a particular statement, thus serving as a branch destination.

syntax — `LABEL <var_name> ':' <statement>`

parameters

LABEL	keyword: indicates that a label for a particular statement follows.
<var_name>	identifier for statement label.
<statement>	any valid SQL/DAL statement.

notes — This statement, together with the GOTO statement may only appear within a procedure definition, and is not permitted in the outer block. Please note that PrimeBase DAL does not include this limitation, this is only a limitation by DAL.

The scope of a statement label is the procedure that contains it.

A statement label and local variable may share the same name without arising confusion.

PINKCTL

function — This statement controls the DAL compatibility of the PrimeBase DAL Server. It has been added to DAL in order that certain desirable features may be added to the PrimeBase DAL Server, without sacrificing compatibility with existing DAL Servers.

Each expression (in order from left to right) controls an aspect (parameter) of DAL compatibility. An expression may be omitted if the user wishes to leave the current compatibility setting unchanged. A value of zero always sets the compatibility parameter to 100% DAL compatible. Positive numbers (1,2,...) represent various modes of incompatibility depending on the parameter. See notes for details.

syntax `PINKCTL <expr_value_expr>`

parameters

PINKCTL	keyword: indicates that an expression follows.
`<expr_value_expr>`	that when evaluated defines an aspect of DAL compatibility.

notes The statement:

```
PINKCTL 0;
```

sets all compatibility parameters to zero (100% DAL compatible. This is the start-up setting for the compatibility parameters. Each of the compatibility parameters, and their various modes are described below.

PARAMETER 1

The first compatibility parameter controls the resulting rowset of a DESCRIBE DATABASES statement. It is possible to get further (useful) information from the describe databases statement when the parameter is set to a non-DAL compatible;

Mode 0: In this mode, DESCRIBE DATABASE returns the following rowset

Col#	Data Type	Name
1	VARCHAR[31]	name

Mode 1: In this mode, DESCRIBE DATABASE returns the following rowset:

Col#	Data Type	Name
1	INT	id
2	VARCHAR[31]	name

Mode 2: In this mode, DESCRIBE DATABASES returns the following rowset:

Col#	Data Type	Name
1	VARCHAR[31]	name
2	CHAR[3]	access

Mode 3: In this mode, DESCRIBE DATABASE returns the following rowset:

Col#	Data Type	Name
1	INT	id
2	VARCHAR[31]	name
3	CHAR[3]	access

In the rowsets above, the column “id” contains the database identifier, as given in the master database. The column “access” contains information concerning the user's access to that particular database. The column can contain one of the following values:

DBA	The user has DBA privileges in the database.
USR	The user has normal user status in the database.
NA	No access. The user is not a user of the database, and may not open the database.

In the case of DBA and USR, the user may open the database. In the case of NA, the user will not be allowed to open the database.

PARAMETER 2

Compatibility parameter 2 controls the formatting of DATETIME (TIMESTAMP) values. Standard DAL DATETIME formatting suffers from the problem that in the format string ($tsfmt) the symbol “MM” is ambiguous. It could mean either months or minutes.

Mode 0 — In the DAL compatible mode, this ambiguity is solved by expecting the date format to appear before the first space, and the time format to appear after the first space. The expression of DATETIME values is therefore fairly limited. It probably would have been good enough, in this case, to have specified that DATETIME values are formatted using $datefmt (DATE format) and $timefmt (TIME format) strings concatenated together. In fact, the PrimeBase DAL Server will do this if the $tsfmt string is set to NULL ("" or NULL).

Mode 1 — The ambiguous symbol, "MM" is clarified by designating the upper-case version ("MM") as months, and the lower-case version ("mm") of the symbol as minutes. In this way, the time and date symbols, in the $tsfmt string, may be freely mixed.

PARAMETER 3

Compatibility parameter 3 is used to make PrimeBase more compatible with P.INK SQL 1.x. There are 3 options for compatibility with P.INK SQL 1.x:

1. When turned on, this option will place the value zero (or empty string) into an inserted column whose value is not specified and the column has no default.

2. This optional string trims trailing spaces from all values returned through the API. In this case the stated length of the string will exclude the trailing spaces (for CHAR and VARCHAR values).

3. This option allows all possible settings of these three options:

Mode	1	2	3
Mode 0	-	-	-

Mode	1	2	3
Mode 1	ON	-	-
Mode 2	-	ON	-
Mode 3	ON	ON	-
Mode 4	-	-	ON
Mode 5	ON	-	ON
Mode 6	-	ON	ON
Mode 7	ON	ON	ON

PARAMETER 4

This parameter allows you to define in which order the driver handles date and time values.

Mode 0 — In this mode, a timestamp value is ordered by time first and then date.

Mode 1 — In this mode, a timestamp value is ordered by date first and then time.

PARAMETER 5

Parameter 5 concerns the confusion regarding buffer length when retrieving a character string value; using this parameter, you can specify how the application should handle this exchange.

Mode 0 — DAL compatible mode: most, if not all, existing DAL applications seem to work efficiently if we assume that the size stated for the user buffer is one less than the actual size. In this mode, the PrimeBase DAL driver assumes that there is an extra byte, and always adds the zero terminator.

Mode 1	In this mode, the driver foes not assume that there is an extra byte, and nevertheless always adds the zero terminator if there is space. Be warned that this can result in the last byte of the string going missing.
Mode 2	In this mode, the driver does not assume that there is one extra byte, and only adds the zero terminator if there is space. This is contrary to DAL.

PARAMETER 6

This parameter determines the binary coded decimal (BCD) format for data types DECIMAL and MONEY. The application can choose between DAL format and DAM format for exchange of BCD numbers in decimal form.

Mode 0	DAL compatible mode: all BCD numbers are exchanged in the internal form used by the PrimeBase Client driver.
Mode 1	DAM compatible mode: all BCD numbers are exchanged in the format specified by the DAM. This format is described in “Inside Macintosh 6”, pages 8-36.

Note: When a new session is started, Mode 1 is the default mode.

PRINT

function — This statement outputs a series of data values as a single row of output.

syntax — `PRINT <expression> {',' <expression> }';'`

parameters

PRINT	keyword: indicates that an expression will follow.
<expression>	an expression whose value will become a column of the output row.

notes The statement creates a single output row with as many columns as there are expressions. The data type of the column is the data type of the corresponding expression. See the PRIINTCTL statement for details as to output data type mapping.

A maximum of 256 expressions can appear in the expression list.

PRINTALL

function This statement prints the entire contents of a rowset.It sends the table described by <cursor> to the user application. The output is modified according to the current PRINTCTL statement. PRINTALL statements are generated automatically by the db...() type functions which are batch oriented and do not make use of the program control statements.

syntax `PRINTALL [<cursor>] ';'`

parameters

PRINTALL	keyword: indicates that the contents of the current cursor (or a specific cursor) are to be printed.
[<cursor>]	the cursor identifying the rowset whose contents are to be printed. This clause is optional. If it is omitted, then the default cursor ($cursor) is assumed.

notes This statement creates zero or more output rows, depending on the contents of the rowset. The number of columns in each row is equal to the number of columns in the corresponding rowset. The data type of each column is the data type of the corresponding column in the rowset. See PRINTCTL for details on output data mapping.

Use the PRINTINFO statement to find out information on data types and names of the columns in the rowset.

There is no limit on the number of rows or columns in a rowset, however there is a limit on the transmission of data values to the client. A single row of output should not be more than 32,000 characters, and a single value of output should also not be greater than 32,000 characters.

PRINTCTL

function — This statement controls the data type mapping of output rows. You can specify the manner in which each data type is to be returned to the client application. There are three ways in which this can be done. Firstly you can specify a data type conversion according to position - see the data type codes and names in the table below. Secondly you can define a specific conversion - from a source data type to a destination data type, and thirdly you can specify a procedure that handles data type conversion for a particular source data type.

syntax

```
PRINTCTL <conversion_spec> {',' <conversion_spec>} ';'

<conversion_spec>::= <expr_value_expr> ['='
( <expr_value_expr> | <procedure_name>) ] | <procedure_name>
```

parameters

PRINTCTL	keyword: indicates that specifications for data type mapping follow.
<conversion_spec>	specifications as to how data types are to be converted.
<expr_value_expr>	any valid expression that states how a data type is to be converted.
<procedure_name>	a previously declared procedure that specifies the conversion of a source data type.

notes — Data types can be referred to either by their data type code, or by their system variable name. The data types, their codes and names are listed in the following table:

Type	Code	System variable name
BOOLEAN	1	$boolean
SMINT	2	$smint
INTEGER	3	$integer
SMFLOAT	4	$smfloat

Type	Code	System variable name
FLOAT	5	$float
DATE	6	$date
TIME	7	$time
TIMESTAMP	8	$timestamp
CHAR	9	$char
DECIMAL	10	$decimal
MONEY	11	$money
VARCHAR	12	$varchar
VARBIN	13	$varbin
LONGCHAR	14	$longchar
LONGBIN	15	$longbin

Using this table, you can specify an integer expression, whereby, for example the expression for, say, code number 7 (TIME data type) produces the result value 6 - meaning that TIMESTAMP is converted to DATE.

There is a special code - namely 0, (zero), which specifies mapping to VARCHAR (type code 12) and where NULL values are returned as the text string “$NULL”. For convenience´ sake, the statement:

```
PRINTCTL 0,0,0,0,0,0,0,0,0,0,0,0:
```

can be abbreviated to:

```
PRINTCTL 0;
```

If the name of a procedure appears, the DAL subsystem is directed to call the named procedure before sending it back to the client application. The procedure should be declared to take a single argument and to produce a single value. The DAL subsystem sends the return value from the procedure back to the client application.

The PRINTCTL governs output data mapping until another PRINTCTL statement supersedes it.

This statement affects the actual data returned by the API to the client application.

The PRINTINFO reports the DAL data type of each column in a rowset - that is the unmapped types. If the contents of a rowset are printed with a PRINTALL or PRINTROW statement, however, the client application will map the received data.

The PRINTCTL statement must be used before values can be received in binary form.

PRINTF

function — This statement puts data items into an output-formatted string.

syntax — `PRINTF '(' <expression> { ',' <expression> } ')' ';'`

parameters

PRINTF	keyword: indicates that specifications follow for an output format string.
<expression>	an expression, whose value is a format string to control formatting. See the description of the $format() function in the section on Built-in Function for the syntax of a format string.

notes — This statement creates a single output row containing a single output column of type VARCHAR. This is, of course, subject to PRINTCTL - output data mapping.

This statement is especially useful for client applications that have little or no tabular-processing capability and that are text-oriented.

The $format() function provides the same formatting features as PRINTF, but produces a VARCHAR return value instead of generating output.

PRINTINFO

function — This statement returns a description of a rowset to the client application.

syntax — `PRINTINFO [ <cursor>] ';'`

parameters

PRINTINFO	keyword: indicates that a description of the current rowset is to be returned.
<cursor>	an optional clause. Specify which rowset is to be described. If this clause is omitted, the default cursor ($cursor) is assumed.

notes — This statement generates a single row for each column of the specified rowset. The table returned into <cursor> is as follows:

Col#	Data Type	Name
1	SMINT	number
2	VARCHAR[255]	name
3	SMINT	type
4	SMINT	length
5	SMINT	scale
6	SMINT	width

Similar information to that generated by the PRINTINFO statement can be obtained through the following system functions: $colname(), $coltype(), $rows(), and $cols().

PRINTROW

function — This statement prints the current row of a cursor.

syntax `PRINTROW [<cursor>] ';'`

parameters

PRINTROW	keyword: indicates that the current row of a cursor is to be printed.
[<cursor>]	an optional clause. The name of the cursor in which the row is to be found that you want printed.

notes This statement creates a single output row with as many columns as there are columns in the current cursor. The data type is the type of the corresponding column of the rowset. See PRINTCTL for details as to output data mapping.

Use the PRINTINFO statement to find out information about the names and data types, and other related information, of the columns of a rowset

QUERY SPECIFICATION

function A query specification is the part of a data manipulation statement, where you specify exactly what data you want retrieved from the database. The syntax for a query specification is used in the SELECT statement, and also in the INSERT statement.

syntax

```
<query_spec>::= SELECT [ ALL | DISTINCT ] <select_list>
<table_expr>

<select_list>::= <select_item> { ',' <select_item> }

<select_item> ::= '*' | ( <table_alias> '.' '*' ) |
<column_reference | <search_condition> [ [ AS ] <column_alias>
]

<table_expr>::= <from_clause>
[ <where_clause> ]
[ <group_by_clause> ]
[ <having_clause> ]

<from_clause>::= FROM <table_specification> { ','
<table_specification> }
```

```
<table_specification>::= <table_reference> [ [ AS ]
<table_alias> ] | <subquery> [ AS ] <table_alias>

<where_clause>::= WHERE <search_condition>

<group_by_clause>::= GROUP BY <column_reference> { ','
<column_reference> }

<having_clause>::= HAVING <search_condition>
```

parameters

SELECT [ALL | DISTINCT]	statement identifying keywords: indicate that data is to be retrieved from the database. The clause ALL | DISTINCT is an optional clause. The default is DISTINCT.
<select_list>	The select_list is a comma separated list of select_items. It is known as a "projection".
*	represents all the columns for the specified relation(s).
<table_alias>.*	an all-columns-of-table specification. The table alias obviously refers to a table in the FROM list.
<column_reference>	the name of the column, or columns from which you wish to select data.
<search_condition>	any valid <search_condition> (chapter Search Conditions in the User Manual). A query specification expressed through a search condition is a calculated column.
[[AS] <column_alias>]	an optional clause. This is best used in conjunction with the search condition: the calculated column can be given an alias name for the purposes of the current session.
<table_expr>	species from which table the data is to be retrieved. Included in the table expression are a number of optional clauses, where it is possible to focus in on specific data within the table.
<from_clause>	specifies from which table the data is to be retrieved.
FROM	clause identifying keyword.

<table_specification>	a table can be referenced, either by entering the qualified name of the table, or through a subquery.
<table_reference>	qualified name of the table.
[[AS <table_alias>]	an optional clause: the table you are retrieving from can be given an alias, which is then valid for the duration of the current session.
<where_clause>	restricts the rows of the projection (of the <select_list>).
WHERE	clause identifying keyword´.
<search_condition>	any valid <search_condition> (chapter Search Conditions in the User Manual). A query specification expressed through a search condition is a calculated column.
<group_by_clause>	summarises the rows of returned data into summary rows. All rows that have identical data values in one or more specified columns are divided into groups. These groups are then summarized by a single summary row for the group.
GROUP BY	clause identifying keywords.
<column_reference>	the qualified name of the column or columns.
<having_clause>	This clause is used to restrict the summary rows created by GROUP BY, just as the WHERE clause restricts those rows that a re subject to grouping.
HAVING	keyword: indicates that a search condition follows, which is used as a measure to restrict the rows subject to grouping.
<search_condition>	any valid search condition. Please see section on search conditions for details as to syntax.

RETURN

function — This statement is used to return values from a PrimeBase procedure. It ends execution of a called procedure. Execution then continues with the statement that immediately follows the CALL statement in the calling procedure. One or more return values can optionally be returned to the calling procedure. The evaluation of the expression list in the RETURN statement obtains the return values.

syntax — `RETURN [<expression> { ',' <expression> } ] ';'`

parameters

RETURN	keyword: indicates that return values are to be obtained through the evaluation of the expressions that follow.
<expression>	any valid expression.

notes — Items in <expression> may evaluate to NULL, producing a NULL return value.

ROLLBACK

function — This statement aborts whatever transaction is currently in process. Any updates that have been made to the database during such a transaction are rolled back, and a new transaction is begun.

syntax — `ROLLBACK [ WORK | TRANSACTION | TRANS ] ';'`

parameters

ROLLBACK	keyword: indicates that transaction should be rolled back.
WORK | TRANSACTION | TRANS	These keywords are optional: only include them for readability.

SELECT

function — The SELECT statement retrieves data from open tables in the database. The data is returned into a rowset.

syntax

```
<query_spec>

[ORDER BY <sort_spec> {',' <sort_spec>} ]
[INTO <cursor> ]
[FOR (READONLY | <scrolling_mode> | <update_mode> | EXTRACT )
] [ ';']

<sort_spec> ::= ( <column_reference> | <integer_literal> ) [
ASC | DESC ]

<scrolling_mode> ::= SCROLLING [ <update_mode> ]

<update_mode>::= UPDATE [ <column_group> | OF <column_name> {
',' <column_name> } ]
```

parameters

<query_spec>	see the section called “Query Specification”.
ORDER BY	This clause is used to order the data returned by a SELECT statement. Data can be ordered in an ascending or descending order, specified by the words ASC and DESC. If neither of these words are specified, then ASC is assumed by default.
<sort_spec>	the specifications as to how the rowset is to be ordered.
<column_reference>	the qualified name of the column to be ordered by.
<integer_literal>	the numerical position that is given to each separate column in the select_list, in the ORDER BY clause. From left to right, the columns are numbered from lowest to highest.
ASC | DESC	ascending or descending. If neither is specified, then ascending is assumed by default.

INTO <cursor>	the rowset can be loaded into a cursor. Cursors are declared with the DECLARE statement. See the section called “DECLARE” for more details.
FOR	this keyword indicates that a specification as to what is to happen to the returned data.
READONLY	specifies that the rowset resulting from the query will be read sequentially, using FETCH NEXT statements only, and that the cursor will not be used to update the database. READONLY is the default if no other update mode is specified.
SCROLLING	specifies that the rowset resulting from the query will be processed by means of FETCH motions (NEXT, PREVIOUS, FIRST, LAST, etc.) and that the cursor will not be used to update the database.
EXTRACT	specifies that the rowset resulting from the query will be processed by means of any of the possible FETCH motions and that the cursor will not be used to update the database. With this mode the number of rows in the rowset can be determined after the SELECT statement is complete.
UPDATE	specifies that the rowset resulting from the query will be read sequentially, using FETCH NEXT statements only, and that the cursor may be used to update the database in subsequent positioned UPDATE or positioned DELETE statements. If UPDATE is specified, the FROM clause of the query specification must name a single table.

notes

Update modes

The SELECT statement update modes (READONLY, UPDATE, SCROLLING, and EXTRACT) control the detailed operation of the SELECT statement and the subsequent FETCH and FOR EACH statements that reference the rowset it creates.

The simplest mode is READONLY. In this mode, the Server fetches data from the database row by row, depending on the specifications in the subsequent FETCH or FOR EACH statements. The motion supported is a forward sequential direction; those rows that were previously retrieved are discarded by the Server and are no longer available without performing another query. As the READONLY mode only retrieves rows from the database on request, the total number of rows in the resulting rowset cannot be predicted until all rows have been retrieved by the FETCH statement. The $rowcount system variable is set to NULL immediately after the SELECT statement, thus is this mode appropriate for those queries in which the data is read into the client application once, and is then processed exclusively within the client application itself.

The SCROLLING update mode is slightly more complex. As with READONLY, the Server fetches data from the database as specified by either the FETCH or FOR EACH statements, however, in addition to making the row available to a DAL program for processing, the Server adds the row to a shadow copy of the rowset, which it then maintains. Thus FETCH motions for previously retrieved (skipped) rows can be read from the shadow copy.This shadow copy of the rowset is stored on the client.

The size of the rowset in the SCROLLING mode cannot be predicted until all rows have been retrieved - as in READONLY. The $rowcount system variable is set to NULL immediately after the SELECT statement. This mode is appropriate for queries in which the client application anticipates a large rowset and uses the Server to support large scale scrolling through the data. This flexibility is not without its price: additional mass storage requirements and I/O overhead on the client.

PrimeBase has an extra mode - SCROLLING for UPDATE. In effect, it is similar to the normal SCROLLING mode, except that once your rowset has been retrieved, you can indicate your intention to update the rowset. In the SELECT statement, you must include the primary key in the select_list. To actually update the rowset, use the UPDATE command and refer to the cursor, in which the rowset was stored. These changes are written straight to the database on the Server. You will not see these changes reflected in the rowset until you have performed another SELECT query. This method of updating has its restrictions:

only simple columns may be selected (no aggregate functions or calculated columns): GROUP BY and HAVING are not allowed - in contrast to DAL. ORDER BY is permitted.

The OF clause in <update_mode> is alternative syntax for the normal <column_group> syntax, provided for compatibility with DAL.

The ';' is optional, above. There is, however, a problem with this: if the ';' and the FOR clause are omitted and the SELECT statement is by FOR or FOR EACH loop (see <for_stat> and <for_each_stat>) the compiler will mistake these statements for the FOR clause.

SET

function — This statement assigns a value to a variable. The statement has three forms. In the first form, the specified expression is evaluated, and its value is assigned to the given variable. The second and third form increase and decrease respectively the specified variable.

syntax

```
[SET] <var_name> '=' <expression> ';' |

[SET] <var_name> '++' ';' |
[SET] <var_name> '--' ';'
```

parameters

SET	keyword: indicates that a value for a variable is being set. This word is optional.
<var_name>	the identifier of the variable that is to be assigned a value.

notes — The expression may contain literals, variables, cursor-based column references, as well as arithmetic and comparison operators. Subqueries and aggregate functions are not permitted.

The data type of the expression will be automatically converted to that of the var_name, if this is possible, according to the data type conversion rules.

A NULL expression value assigns the NULL value to the variable.

SWITCH

function

The SWITCH statement acts as a junction to multiple paths in a program´s flow of control. In many ways it is like the IF statement, except that numeric or string values are compared to match the head expression, rather than the execution of a simple TRUE/FALSE test.

The head of the SWITCH statement is an expression which is evaluated to produce a result value. The body of the SWITCH statement contains one or more CASEs, which each specify a constant integer of string value. These are the comparison expressions. If one of these comparison_expressions matches the expression in the head on the SWITCH statement, then that particular CASE statement is then executed.

If no case values match the head expression, then the default statement is carried out - if such a default statement is present.

Each CASE statement_list should end with the BREAK statement, otherwise the program flow continues execution with the next CASE statement_list. Only in the CASE statement_list preceeding the default case need there be no BREAK statement.

syntax

```
SWITCH '(' <expression> ')' '{'

{ CASE <literal> ':' <statement_list> }
[DEFAULT ':' <statement_list>]
'}'
```

parameters

SWITCH	keyword: indicates that head expression follows.
<expression>	an expression that is evaluated to control the switch.
CASE	indicates that a value follows to be used in comparison with the control expression.
<literal>	a constant expression that specifies the matching value for this case.
<statement_list>	the sequence of statements to be executed if the value of the <expression> matches the case <literal>.

	DEFAULT	keyword: indicates that the statement_list that follows is to be executed should none of the case literals match the head expression.

notes

The DEFAULT case may appear only once in the SWITCH statement, but can be placed anywhere near to other CASEs.

As mentioned before, the CASEs themselves do not alter flow of program; flow continues sequentially until all statements have been executed. Interrupt this flow with BREAK, RETURN, GOTO or similar statements. The most common to use is the BREAK statement.

Data types are converted for comparison, just as they would be for a comparison expression (<expression> == <comparison_expression>).

The system variable $switch is set to the value of <expression> for the last SWITCH statement that is executed when the body of the SWITCH statement is entered.

UPDATE (POSITIONED)

function

The positioned UPDATE statement updates data in the current row of a cursor. The SET clause describes how a column is to be updated.

syntax

```
UPDATE <table_reference> SET <set_clause> {',' <set_clause>} [
WHERE CURRENT OF <cursor> ] [';']

<set_clause>::= <column_name> '=' <search_condition>
```

parameters

UPDATE	keyword: indicates that data is to be modified.
<table_reference>	the qualified name of the table to be updated.
SET	keyword: indicates that specifications as to how the data is to be altered follow.
<set_clause>	an expression that describes how a column is to be updated. No subqueries, or aggregate functions are allowed in the <search_condition> in this clause.

<column_name> name of column you are wanting to update.

<search_condition> an expression whose value is assigned to <column_name>. This expression may include references to the columns of the update table. The <search_condition> in the SET clause is evaluated by the Server.

WHERE CURRENT OF

keyword: indicating the current row of the specified cursor.

<cursor> the name of an active cursor with a valid current row that is then updated by this statement.

notes

The table reference must specify an updatable table or view. In the case of updating a view, this is only possible if the view is derived from a single table.

The cursor must specify a rowset created by a SELECT statement that included an update mode (see the SELECT statement for more details). If the <update_mode> specifies a list of updatable columns, then every column_name in the SET clause must be present in the <update_mode> list.

According to the definition of the column (from CREATE TABLE), the values that can be updated may be restricted. Attempts to insert a row in violation of these restrictions will result in an error.

If an expression in the <set_clause> includes a reference to a column being updated, the value used in computing the expression is the value of the column before any updates are performed on the row.

$rowsaffected is set after a successfull INSERT, UPDATE or DELETE. The value is the number of rows effected by the query. If an error occurs $rowsaffected is set to zero.

UPDATE (SEARCHED)

function — This statement modifies data in one or more rows of a table based on a row-selection criteria. The search condition in the WHERE clause identifies the rows to be updated. These columns are then updated according to the specifications in the SET clause.

syntax

```
UPDATE <table_reference> SET <set_clause> {',' <set_clause>} [
WHERE <search_condition>] [';']

<set_clause>::= <column_name> '=' <search_condition>
```

parameters

UPDATE	keyword: indicates that data is to be modified.
<table_reference>	the qualified name of the table to be updated.
SET	keyword: indicates that specifications as to how the data is to be altered follow.
<set_clause>	an expression that describes how a column is to be updated. No subqueries, or aggregate functions are allowed in the <search_condition> in this clause.
<column_name>	name of column you are wanting to update.
<search_condition>	an expression whose value is assigned to <column_name>. This expression may include references to the columns of the update table. The <search_condition> in the SET clause is evaluated by the Server.
WHERE	keyword: indicating which rows are to be updated. If this clause is left out, then all rows of the specified table are updated.

notes — The table reference must specify an updatable table or view. In the case of updating a view, this is only possible if the view is derived from a single table.

According to the definition of the column (from CREATE TABLE), the values that can be updated may be restricted. Attempts to insert a row in violation of these restrictions will result in an error.

If an expression in the <set_clause> includes a reference to a column being updated, the value used in computing the expression is the value of the column before any updates are performed on the row.

$rowsaffected is set after a successfull INSERT, UPDATE or DELETE. The value is the number of rows effected by the query. If an error occurs $rowsaffected is set to zero.

WHILE

function — This statement executes repeatedly and is terminated by a pretest. With each repetition, the WHILE statement evaluates a specified test expression. If this evaluates as TRUE then the statement is executed and a cycle of expression evaluation followed by statement execution begins. This continues until the expression evaluates as FALSE, when flow then passes on to the next statement after the WHILE statement.

syntax — `WHILE '(' <expression> ')' <statement>`

parameters

WHILE	keyword: indicates that an expression is to be evaluated; if this expression evaluates as TRUE, then the statement that follows may be executed.
<expression>	any valid expression.
<statement>	any valid statement.

notes — The <statement> is usually a compound statement.

Remember that if the <expression> is evaluated as FALSE the first time, then the statement will not be executed at all.

An expression that evaluates to NULL is FALSE.

API FUNCTIONS

The API functions are the interface between the client application and the DAL runtime environment. The client application calls the API functions to perform the following tasks.

- begin and end communication with a DAL server.
- send fragments of DAL programs to the runtime environment.
- request execution of a program fragment.
- determines the status of execution of a DAL program.
- retrieve data passed back from the DAL program.
- determines the type of data passed back from the DAL program.
- force a break or abnormal termination of the DAL program.
- collect details of any error that occurs during the execution of a DAL program.

API FUNCTION GROUPS

The API functions described in this section may be divided into the following function groups:

Session-control functions

- CLInit() Begins a DAL session
- CLEnd() Ends a DAL session
- CLConInfo() Describes a DAL session
- CLGetSn() Returns the session number

Program-execution functions

- CLSend() Sends DAL program text to the runtime environment
- CLSendItem() Sends binary data to the runtime environment
- CLExec() Executes previously sent DAL programs
- CLState() Checks DAL program execution status
- CLGetErr() Retrieves error codes and messages
- CLBreak() Interrupts and terminates DAL program execution

Results-processing functions

- CLGetItem() Describes and/or retrieves the next data item
- CLUnGetItem() Unretrieves the previous data item

RETURN VALUES

All the API functions return integer values to indicate either successful or unsuccessful completion of the call, or other types of status conditions. See the following tables:

Integer Values for Return on Codes

Symbol	Value	Description
A_NULL	1	The requested data item is a NULL, therefore no data was returned.
A_OK	0	Function successfully completed
A_VALUE	0	When returned by an information request call, the DAL program produces output data that is awaiting retrieval by the client application; when returned by a data retrieval call, the data item was successfully retrieved into the supplied buffer.
A_ERROR	-1	Program execution ended in an error condition. The error details can then be retrieved with CLGetErr().
A_READY	-2	The runtime environment is ready to accept program text. If a DAL program fragment was being executed, it has finished successfully and there is no more output to be processed by the client application.
A_BADTYPE	-3	The item requested is not of the data type expected. No data has been returned.

Symbol	Value	Description
A_BREAK	-5	An API results-processing function was interrupted on request from the client application, or a timeout has occurred.
A_EXEC	-6	A DAL program is currently being executed. There is no output data waiting to be processed by the client application.
A_NOTCONN	-7	Specified session ID parameter is invalid.
A_SESSMAX	-8	Specified session number parameter is out of range.
A_INUSE	-9	Session is in use by another client application
A_NOHOST	-10	Session is open, but not connected to a host.

Integer Values for Data Type Codes

Symbol	Data Type	Value	len	places
A_BOOLEAN	BOOLEAN	1	No	No
A_SMINT	SMINT	2	No	No
A_INTEGER	INTEGER	3	No	No
A_SMFLOAT	SMFLOAT	4	No	No
A_FLOAT	FLOAT	5	No	No

Symbol	Data Type	Value	len	places
A_DATE	DATE	6	No	No
A_TIME	TIME	7	No	No
A_TIMESTAMP	TIMESTAMP	8	No	No
A_CHAR	CHAR	9	Yes	No
A_DECIMAL	DECIMAL	10	Yes	Yes
A_MONEY	MONEY	11	Yes	Yes
A_VCHAR	VARCHAR	12	Yes	No
A_VBIN	VARBIN	13	Yes	No
A_LCHAR	LONGCHAR	14	Yes	No
A_LBIN	LONGBIN	15	Yes	No

Results Processing

The API functions, CLGetItem(), and CLGetUnItem(), allow the client application to process output data generated by the execution of a DAL program.Output data is generated by the various “print” statements: PRINT, PRINTROW, PRINTALL, PRINTF, and PRINTINFO. The “print” statements generate an output data stream with an implicit row/column structure for each statement.

Output from the DAL program is treated as one continuous stream of data items by the DAL API. The CLGetItem() function retrieves each data item in sequence, operating from the client application. The data items are returned to the client application each with a set of flags. The flags are used to indicate NULL values, end-of-row boundaries, and whether the output has been formatted by the DAL program.

The CLUnGetItem() function can be used to return a data item, that has just been retrieved, back to the stream, so that it can be retrieved again later. When no more data is there, i.e. when the end of the DAL program´s output data stream is reached, the API returns a A_READY result.

Another function of CLGetItem(), is to obtain information about the data item from the API. For example, you may want to find out information on the data type and length of the next data item, and based on this information then allocate a buffer. You can then call the API a second time, and actually retrieve the data item, and place it in the buffer. In this way the client application can adapt to the returned data.

API Functions and NULLs

A client application should be prepared to handle NULL values in the output data stream from the Server. The return value A_NULL, from the CLGetItem() function indicates a NULL value.

If the client application cannot handle NULL values, you can use the PRINT statement:

```
PRINTCTL 0;
```

All data is converted by this statement to VARCHAR strings; NULL values are represented by the string “$NULL”.

CLBREAK() FUNCTION

function This function interrupts the runtime environment, by resetting, or aborting it.

syntax
```
CLBreak (sessid, abort)

long sessid;
int abort;
```

parameters

CLBreak	The name of the function call.

	sessid	The session ID that specifies which session is to be re-set or aborted. CLInit() returns the session ID.
	abort	This requests that the current runtime environment be aborted if non-zero. If zero, it requests that program execution be halted, and that the runtime environment be reset.
return values	A_OK	The session has been successfully reset or aborted.
	A_ERROR	The reset, or abort attempt was unsuccessful.

notes If the runtime environment is blocked, while waiting for a reply from the host, any attempt to reset the environment will be unsuccessful. The reason for this is that the runtime environment cannot process this function until the host returns control to it.

This function can also be used to break an infinite loop or stop runaway output from the program that is currently executing.

CLCONINFO() FUNCTION

function This function returns information about a specified session.

syntax
```
CLConInfo (sessid, sessnum, outid, vrsid, host, user, network,
connstr, start, statep)

long sessid;
int sessnum;
long *outid;
long *vrsid;
char *host;
char *user;
char *network;
char *connstr;
long *start;
long *statep;
```

parameters	CLConInfo	The name of the function call.
	sessid	The session ID specifying the session about which you want information. If sessnum is used instead to identify the session, then sessid must be zero.
	sessnum	The session number specifying the session about which you want information. If sessid is used instead to identify the session, then sessnum must be zero.
	outid	This is used to return the corresponding sessid from a call where only the sessnum was specified.
	vrsid	This is used to return the client API version number as a sequence of 4 bytes.
	host	This is used to return the host parameter that was specified in the CLInit() call that created the session. This is the name of the Server.
	user	This is used to return the user parameter that was specified in the CLInit() call that created the session.
	network	This is used to return the name of the type of network connection that is being used by the session.
	connstr	This is used to return the connection string parameter that was specified in the CLInit() call that created the session.
	start	This is used to return a unique TIMESTAMP value that is associated with the start of a session.
	statep	This returns the state of the session as it would be reported by the CLState() call. The possible return values are the same as the CLState() return values.
return values	A_VALUE	Information for an active session has been retrieved successfully.
	A_NULL	The input session id or number refers to an inactive session.

	A_SESSMAX	The specified sessnum is out of range.

notes Normally, this function should be used with a non-zero sessid obtained from a previous CLInit() call (where sessnum is set to 0), in order to obtain information about a session that was initiated by the client application.

For testing purposes, sessnum can be used to find out information on all the available sessions - whether or not they were initiated by the client application.

The parameter statep can also return the value A_EXEC, which means that the requested session was already in use by another user at the time when the CLConInfo() call was made.

CLEXEC() FUNCTION

function This function requests the execution of a program fragment that has previously been passed by one or more CLSend() or CLSendItem() calls. The run-time environment then begins execution of the fragment, and control is immediately returned to the client application.

syntax
```
CLExec (sessid)

long sessid ;
```

parameters	CLExec	The name of the function call.
	sessid	The session ID specifying the session for which the program fragment should be executed. The session ID is that which is returned by the CLInit() call.
return values	A_OK	Execution of program fragment has begun successfully.
	A_ERROR	An error occurred while trying to begin execution.

notes The client application is free to continue work, while the runtime environment begins execution.

This call is asynchronous, error messages may not be available at completion.

You can determine the current status of execution with the CLState() function call.

CLGETERR() FUNCTION

function — This function call retrieves error codes and messages after an error has occurred.

syntax

```
CLGetErr (sessid, perr, serr, itm1, itm2, msg)

long sessid;
long *perr;
long *serr:
char *itm1;
char *itm2;
char *msg;
```

parameters

CLGetErr	The name of the function call.
sessid	The session ID that is returned by CLInit().
perr	The buffer in which the primary error code is placed.
serr	The buffer in which the secondary error code is placed.
itm1	The buffer for the zero-terminated string that identifies the first object of the error message.
itm2	The buffer for the zero-terminated string that identifies the second object of the error message. The maximum number of bytes returned is 256 (including zero terminators).
msg	The buffer for the returned error message. The maximum number of bytes returned is 256 (including zero terminators).

return values

A_OK	The error information has been retrieved successfully.

A_ERROR An error occurred while retrieving the error information!

notes This call must be made following a function call that returns an error status without any intervening API calls. This must be so in order to retrieve the error information directly associated with the returned error status.

Any of the pointers may be NULL is the user is not interested in the data that may be returned

When used after an unsuccessful CLInit() call, this function may be used to determine why it was unsuccessful.

The message in msg always begins with the name of the program source, as well as the line in which the error occurred. This information is followed by a colon, (:), and the error message itself.For program fragments sent by the CLSend() API function, the program source is identified as “network”. It is often convenient to use the colon delimiter to parse the message so that only the latter half is revealed to the user of the client application.

CLGETITEM() FUNCTION

function This function can either retrieve the next data item from the runtime environment´s output data stream, or it can obtain information about the type and size of the next data item.

This function call is an information request call when made with a NULL (0) buffer pointer, The next data item in the data stream is examined, and data type, length, number of decimal places and flags are returned to the client application, however, the item itself is not retrieved. With the help of the retrieved information, you can allocate a buffer to receive the next data item. Remember, however, that an information CLGetItem() call does not advance the API to the next data item in the data stream.

Data can be retrieved with a non-NULL buffer pointer. Again, the data type, length, number of decimal places, and flags for the next data item are returned in addition to the data item itself. In contrast to an information request call, the retrieval of a data item, with CLGetItem(), does advance the API to the next data item in the data stream.

By passing a data type code to the CLGetItem() function, the client application can tell the API what type of data item it expects to receive. If the type is different to the type of the next item in the output stream, an error code is returned. If the client application is prepared to receive any type of data, then the data-type code, A_ANYTYPE must be passed to the CLGetItem() function.

syntax

```
CLGetItem (sessid, timeout, typep, lenp, placesp, flagsp,
buffer)

long sessid;
int timeout;
short *typep;
short *lenp;
short *placesp;
short *flagsp;
char *buffer;
```

parameters

CLGetItem()	The name of the function call.
sessid	This specifies the session ID call, that is returned by CLInit().
timeout	This specifies the length of time in hundredths of seconds that CLGetItem() should wait for an available data item before returning the code A_BREAK. If you do not wish to specify a time - and therefore allow an unlimited wait for the next data item, then you must pass the value AW_FOREVER.
typep	The typep pointer points to a short value (a 2 byte signed value) that contains the expected data type of the retrieved data item. On return, the value is set to the actual type of the data.

	lenp	The lenp pointer points to a short value (a 2 byte signed value) that contains the length of the buffer pointed to by buffer. The range allowed by this short value is between 0 and 32767 bytes. Once the data item has been retrieved, the value is set to the actual length of the data item.
	placesp	The short value pointed to by placesp is set to the number of decimal places in the retrieved data item. If the item is not DECIMAL or MONEY type, then a zero is returned.
	flagsp	the short value pointed to by flagsp contains a series of bit flags that describe the retrieved data item. The flag AF_ISNULL indicates that the data item is a NULL value. The flag, AF_RECEND, specifies that the retrieved data item is in the last column of an output row. The client application can use this flag to test for an end-of-row condition.
	buffer	To retrieve information about the next item, this parameter must be set to NULL. For data retrieval, it must be a pointer to a buffer where the retrieved data can be placed. If the item is NULL, no data will be placed in the buffer.
return values	A_VALUE	A non-NULL data value was retrieved.
	A_NULL	A NULL data value was retrieved.
	A_ERROR	The execution ended in an error. There is no more data.
	A_READY	Execution ended successfully; there is no more data.
	A_BADTYPE	The data item is not of the type expected.
	A_BREAK	The time limit specified by the timeout parameter ran out while waiting for data from the runtime environment.
notes	The statements PRINT, PRINTF, PRINTALL, and PRINTINFO generate the data items that are retrieved by this function call. The data values form continuous data stream, from which CLGetItem() retrieves each item in sequence.	

For VARCHAR and CHAR types, the length returned is the number of data characters. The returned data also includes a NULL termination byte, which is not counted as a data byte. Therefore when such data types are expected, your buffer size should allow for the "returned length + 1".

In order to skip the following data item in the data stream, call CLGetItem() with a NULL buffer, and set typep to A_DISCARD.

A good use of the timeout feature in the CLGetItem() call is to supply a short timeout value and thus allow the client application to regain control and continue with other work, and not have to wait, should no data item be currently available. While the client application is continuing with other work, the function call CLState(), (page 168), can be called to determine when a data item becomes available. An alternative to this procedure is to set the client application to loop, calling CLGetItem() repeatedly with a relatively short timeout value, then checking for other events between calls.

CLGETSN() FUNCTION

function — This function call retrieves the internal DAL session number for a particular session ID.

syntax

```
CLGetSn(sessid)

long sessid;
```

parameters

CLGetSn	The name of the function call.
sessid	The session ID, that can be obtained by CLInit(), for the session number that you are requesting.

return value — This call returns a session number, which is an integer value of 1 or more.

notes — This function call is really only provided for testing and analytic purposes, and is rarely used by client applications. It is then most often used together with CLConInfo(), in order to retrieve information on all available sessions, whether or not they have been initiated by the client application.

The session number returned can be used as the sessnum of the CLConInfo() function call.

CLINIT() FUNCTION

function

This function call initiates a session for the client application. The call connects it the session to a Server, and thus creates a runtime environment on the host system (in this case PrimeBase). This function must be called before any other API function.

If you are using System 6, DAL uses a connection definition in a file in the System Folder, called "hosts.cl1", and if you are running System 7, the same connection definition is written in the file in the System Folder, called "DAL Preferences". This definition tells DAL how to connect to a particular host system. It specifies the type of network to use, how to find the target host system, and how to log on. See the installation manual for more information. Each connection definition in these files is given a unique name. You supply this name in using CLInit().

syntax

```
CLInit (sessidp, host, user, passwd, connstr)

long *sessidp;
char *host;
char *user;
char *passwd;
char *connstr;
```

parameters

CLinit	the name of the function call.
sessidp	an integer buffer in which the sessnum returned by this function call is stored.
host	the name of the entry in the file (either hosts.cl1, or DAL Preferences, depending on which System you are running).

	user	the user name under which the runtime environment executes.
	passwd	the password for that user name.
	connstr	a connection string, which is passed as a parameter to the underlying network.
return values	A_OK	the session has been successfully initiated.
	A_ERROR	an error occurred while initiating the session.

notes A long-integer session ID is always returned to the buffer at sessidp, even when CLInit() returns an error. This enables you to pass the session ID to CLGetErr(), in order to determine the cause of the error.

The CLEnd() should always be called with the session ID returned by CLInit(), so that all resources to do with that session can be freed. This must still be done, even if CLInit() returned A_ERROR.

CLSEND() FUNCTION

function This function call sends program text to the runtime environment. Text is then added to any previously passed text, to assemble a program fragment. This program fragment is not executed until the client application has called the CLExec() function call.

syntax
```
CLSend (sessid, text, len)

long sessid;
char *text;
int len;
```

parameters	CLSend	The name of the function call.
	sessid	The session ID for which the program fragment is being sent.
	text	A character string containing program text.

	len	The length of the text.
return values	A_OK	The text was accepted successfully.
	A_ERROR	An error occurred while sending the text.

notes

Program text can be passed in units as small as a single character. There is no requirement that complete statements be sent in a single function call.

In PrimeBase DAL a CLSend() cannot obtain part of a token. Each token (keywords, numbers, strings etc.) must be contained completely within the text of a CLSend(). White space and semi-colons must be explicitly included in the passed text.

It is possible to embed new lines and carriage returns in the passed text, in order to divide it into lines. These characters are ignored syntactically. Remember, thought, that DAL error messages reference the line number where the error occurred. Line numbers begin at line 1 for the first text sent via CLSend() after a CLInit() or CLExec() call.

CLSENDITEM() FUNCTION

function

This function call sends a binary data item to the runtime environment. This data item becomes part of the current program fragment that has been put together by a series of CLSend() and CLSendItem() calls.

syntax

```
CLSendItem (sessid, type, len, places, flags, buffer)

long sessid;
int type;
int len;
int places;
int flags;
char *buffer;
```

parameters	CLSendItem	The name of the functions call.

	sessid	The session ID (returned by CLInit()) for which the data is being sent.
	type	The data type code for the data item.
	len	The length of the data item. This is ignored if type specifies a data type that has a self-defining length.
	places	The number of decimal places in the data item. This is ignored for all data types other than DECIMAL and MONEY.
	flags	A word used to pass a flag value - in the same format as that returned by the CLGetItem() call. AF_ISNULL sends a NULL data item; to send non-NULL data items, pass a zero (0).
return values	A_OK	The data item was accepted successfully.
	A_ERROR	An error occurred while sending the data item.
notes	Data items retrieved in binary form by CLGetItem() can later be sent back to the runtime environment with CLSendItem().	

CLSTATE() FUNCTION

function	This function call returns the status of the runtime environment.	
syntax	`CLState (sessid)`	
	`LONG sessid;`	
parameters	CLState	The name of the function call.
	sessid	The session ID of the session for which the status is being requested.
return values	A_EXEC	A program is currently being executed.
	A_VALUE	Output data is available for the client application.

	A_NULL	Output data (a NULL) is available.
	A_READY	Execution was successful; the runtime environment is ready for text.
	A_ERROR	Execution ended in an error.

notes CLState() does not suspend or block user activity.

The CLState() is not reset by CLSend(), but only by CLExec().

CLUNGETITEM() FUNCTION

function This function call basically undoes the effect of the last CLGetItem() data retrieval call. The data item that was retrieved by the immediately preceeding CLGetItem() call is returned to the data stream, and can be re-retrieved the by the next CLGetItem() function call!

syntax
```
CLUnGetItem (sessid)

long sessid;
```

parameters		
	CLUnGetItem	The name of the function call.
	sessid	The session ID, as returned by CLInit().
return values	A_VALUE	The undoing of the last CLGetItem() call was successful.
	A_ERROR	An error occurred during the “undoing”.

notes Only a single data value can be undone; you cannot step back through previously retrieved data,

This function comes in handy when your allocated buffer is not large enough to handle the item and must then reallocate a larger buffer and retrieve the item again.

HYPERCARD XCMDS & XFCNS

HyperCard uses interface functions similar to the API functions implemented by DAL. The HyperCard external functions and commands are used to send DAL language statements to the Server to receive result data.

The HyperCard XFCNs and XCMDs can be split up into the following groups:

Session Control

- CL1Init — Begins a DAL session.
- CL1End — Ends a DAL session.

Program Execution

- CL1Send — Sends DAL text to the runtime environment.
- CL1Exec — Executes previously sent text.
- CL1State — Returns the status of the host: PrimeBase.

Results Processing

- CL1Getval — Retrieves the next data item.
- CL1Getstat — Checks the state of execution.
- CL1Putval — Calls CL1Getval in order to put values into fields.

Global Variables

Global variables are used by HyperCard to provide status and error information for stacks that use DAL. These variables should not be altered, as they have been set by the DAL XCMDs and XFCNs.

- cl1_id — This variable contains the session ID of the current session.
- cl1_error — This variable contains the error-return string. This value is reset after and error has occurred. The string depends upon which command or function has caused the error.
- cl1_status — This variables contains the primary DAL error number. Every time the value of cl1_error is set to “Error”, the value of cl1_status is reset.
- cl1_status2 — This variable contains the secondary DAL error number. Again, this value is reset each time cl1_error is set to “Error”.
- cl1_message — This variable contains a message that corresponds to the error number in cl1_status. As above, this value is reset every time cl1_error is set to “Error”.

CL1END XCMD

function	This command ends a DAL session, and breaks the connection to the Server.	
syntax	`cl1end`	
return values	empty	the session was terminated successfully.
	“Error”	an error occurred in ending the session.
	“No cl1init was done”	the session was not initiated.

notes — You must set cl1_id to the session ID of the session you want to terminate. You need not specify parameters. Any errors are returned in the result field.

Remember that if you do not use CL1End, you run the risk of leaving sessions running on the Server.

This function is similar to the API function, CLEnd().

CL1EXEC XCMD

function — This command executes previously passed program fragments. You can check the status of the executing program after you´ve sent CL1Exec, with CL1Getstat, or CL1State.

syntax — `cl1exec`

return values

empty	the execution began successfully.
“Error”	an error occurred while beginning execution.
“No cl1init done”	the session was initiated.

notes — There is no need for parameters to be specified, and no error information is returned.

This function is similar to the API function, CLExec(), page 159.

CL1GETLIST XFCN

function — This function retrieves a complete series of data items from the Server´s output data stream. All data items are concatenated into a single string, separated by Return characters, and the string is then returned to the result field.

syntax — `cl1getlist()`

notes — This function has no provisions for indicating successful completion, therefore the global variable, cl1error, is cleared upon success, and set upon error.

CL1GETSTAT XFCN

function — This function returns the status of the next available data item int he Server´s output data stream. This statement is normally called before a CL1Getval function, to fund out whether or not a data item is available. If a program fragment is still being executed, and no output data is available, CL1Getstat blocks further execution. This function returns a value when a data item has been produced (by one of the PRINT statements), or when execution finishes successfully, or when an error occurs.

syntax — `cl1getstat()`

return values

empty	a data item has been produced by the CL1Getval function, and is ready for retrieval.
"Ready"	there is no more data. The execution of a DAL program has ended successfully.
"Error"	there is no more data. Execution ended in error, as indicated by the cl1_status, cl1_status2, and cl1_message variable values.
"Timeout"	the time specified for waiting for the next data item has run out.

notes — This function provides the same information as the CL1State function, except that CL1State returns immediately with a "Busy" message, if the Server is in mid-execution, and CL1Getstate will not return values until data is produced. or execution ends.

CL1GETVAL XFCN

function — This function returns the next data item from the output data stream. The value is returned as a variable-length text string, which can be placed in a HyperCard container or in an expression.

syntax — `cl1getval()`

return values	"Error"	there is no next data item. An error occurred during execution.
	"Ready"	there is no next data item. Execution ended successfully.
	"No cl1init done"	the CL1Init XCMD was not executed.
	value	the retrieved data item.

notes This function is the same as the API function, CLGetItem(), when CLGetItem() is used as a data retrieval call, (page 161).

Before using the CL1Getval XFCN, you should check the availability of the next data item with CL1Getstat.

This function blocks waiting-for-output until the Server produces an output data item, or until execution of a program fragment ends.

NULL values are returned as "$NULL" and BOOLEAN values are returned as "$TRUE" and "$FALSE".

CL1INIT XCMD

function This command initiates a session for your HyperCard stack, and makes a connection to the Server. You must connect using this command before executing any other commands or functions. The completion status of the command is returned to the results field.

syntax
```
cl1init [<host_name> [',' <user_name> [',' <password_ [',' <connstr> ]]]]
```

parameters	cl1init	the name of the external command.
	<host_name>	a string expression, the value of which is the entry in the configuration file ("hosts.cl1" or "DAL Preferences").
	<user_name>	a string expression, whose value is the user name, for access to the Server. The name must already have been an entry in the SysUsers table.

	<password>	a string expression, whose value is the password associated with the user name.
	<connstr>	a string expression, whose value is an optional connection string.
return values	empty	the connection was initiated successfully.
	“Error”	an error occurred during the initiation of the connection.
notes	This command is similar to the API function, CLInit, (page 165).	

CL1PUTVAL XCMD

function	This command retrieves a series of data items from the output data stream, and places them in card fields, global variables, and background fields. Where these items are to be placed is determined by a variable-length argument list.	
syntax	`cl1putval [<positive_number>] [',' <negative_number] [',' '”' <global_variable> '”'] [,...]`	
parameters	cl1putval	the name of the external command.
	<positive_number>	a valid card field number.
	<negative_number>	a valid background field number.
	“global_variable”	a global variable name.
return values	empty	the data was successfully retrieved.
	“Ready”	there is no data available, but execution ended successfully.
	“Error”	there is no data available, and an error occurred during execution.
	“Timeout”	the time specified for waiting for data ran out.
	“No cl1init done”	the CL1Init XCMD was not executed.

notes If an error occurs, the global variables, cl1_error, cl1_status, cl1_status2, and cl1_message contain information about the error.

If you are using any version of HyperCard previous to 1.2, you must initialise a global variable before using the command, in order to retrieve data into the global variable.

CL1SEND XCMD

function This command send DAL statements to the runtime environment, and appends it to any text that has previously been sent by other CL1Exec commands. The program is not, however, executed until you issue a CL1Exec command.

syntax `cl1send '"' <statement>'"'`

parameters	CL1Send	the name of the external command.
	<statement>	any valid statement fragment.
return values	empty	the statement was sent successfully.
	"Error"	an error occurred sending the statement.
	"No cl1init done"	the session was not initiated.

notes This command is similar to the API function, CLSend, (page 166).

CL1STATE XFCN

function This function determines and returns the status of the runtime environment. It is usual practice to call CL1STate after you have issued a CL1Exec command, in order to find out such information as to whether the program fragment is still executing, whether it has executed, or whether there is data available.

syntax `cl1state()`

return values	empty	there is output data available from the Server. Use CL1Getval to retrieve the data.
	"Ready"	there is no more data to send, and the runtime environment is waiting for more statements. This value signals the end of the data stream produced by your statements, it is returned immediately if the statements do not produce any output.
	"Error"	an error occurred at the completion of a program. The value of cl1_error, cl1_status. cl1_status2, and cl1_message are set when "Error" is returned.
	"No cl1init done"	a session was not initiated.
	"Busy"	the Server is still executing a program.

notes When the Server is executing, CL1State will return a "Busy" status. CL1Getstat, on the other hand, no matter how much you coax it will never return the status message "Busy".

SYSTEM FUNCTIONS

We divide system functions supported by PrimeBase into the following groups:

- functions that manipulate strings,
- functions that retrieve information and manipulate cursors,
- functions that retrieve information about variables,
- the $format function,
- functions that manipulate files,
- utility functions.

Several functions are not standard DAL, but PrimeBase specific extensions to DAL. These functions are marked as extensions.

STRING FUNCTIONS

String manipulation functions are those that trim strings, extract a substring from a string, and locate substring in a string. In addition to these, PrimeBase provides two functions for converting upper- to lower-case and back.

The string functions are:

- $left extracts a substring from the left of a string.
- $right extracts a substring from the right of a string.
- $substr extract a substring.
- $locate finds a substring occurrence.
- $trim trims blank characters from both sides of a string.

- $ltrim trims blank characters from the left side of a string.
- $rtrim trims blank characters from the right side of a string.
- $toupper convert a string to upper-case characters.
- $tolower convert a string to lower-case characters.

$left and $right

These functions have basically the same function as the $locate function, except that they extract portions of the input string rather than locating a position within it. The syntax for these functions is as follows:

```
<outstr>::= $left '(' <instr> ',' <pattern> [',' <count>]')'
```

```
<outstr>::= $right '(' <instr> ',' <pattern> [',' <count>]')'
```

parameters

<outstr>	the output string
<instr>	the string to be searched
<pattern>	the substring to be located
<count>	an optional argument. If included, a positive <count> value causes the search of <instr> to continue until <count> occurrences have been found. A negative <count> value causes the search to begin at the end of <instr>, and proceed from right to left. If it is omitted, the functions search for the first occurrence of <pattern> within <instr>.

Examine the following examples:

```
$left (“Golfers.Results.Winnings”, “.”, 2)
```

yields `“Golfers.Results”`

```
$left (“Golfers.Results.Winnings”, “.”, -1)
```

yields `“Golfers.Results”`

```
$left (“Golfers.Results.Winnings”, “.”, -2)
```

yields `“Golfers”`

```
$right (“Golfers.Results.Winnings”, “.”, 2)
```

yields `“Winnings”`

```
$right (“Golfers.Results.Winnings”, “.”, -1)
```

yields `“Winnings”`

```
$right (“Golfers.Results.Winnings”, “.”, -2)
```

yields `“Results.Winnings”`

$locate

This function locates a substring within a string and returns its position as an integer. The syntax is as follows:

```
<position>::= $locate '(' <instr> ',' <pattern> [ ',' <count>]
')'
```

parameters

<position>	an integer value for the position of the substring within the string.
<instr>	the string to be searched.
<pattern>	the substring to be located.
<count>	an optional argument. If included, a positive <count> value causes the search of <instr> to continue until <count> occurrences have been found. A negative <count> value causes the search to begin at the end of <instr>, and proceed from right to left. If omitted, the first occurrence of <pattern> within <instr> is located and its position is returned by the function.

```
$locate (“Golfers.Results.Winnings”, “.”)
```

yields 8

If <count> is specified, it instructs the function to continue past the first occurrence of <pattern> until <count> occurrences have been found:

```
$locate (“Golfers.Results.Winnings”, “.”, 2)
```

yields 16

```
$locate (“Golfers.Results.Winings”, “.”, 1)
```

```
yields 9
```

A negative <count> value instructs the function to begin its search at the end of <instr> and search backward through the string.

```
$locate (“Golfers.Results.Winnings”, “.”, -1)
```

yields 16

```
$locate (“Golfers.Results.Winnings”, “.”, -2)
```

yields 8

Please note that with a negative count value, even though the search proceeds from right to left, the returned position is always the position relative to the beginning of <instr>, with the first character as position 1.

$substr

This function is used to extract a substring from a string variable or constant. The syntax for this function is as follows:

```
<outstr>::= $substr '(' <instr>',' <position> [',' <length>]')'
```

parameters	<outstr>	the substring that is extracted from <instr>
	<instr>	the input string from which the output string is extracted

<position> an integer specifying the starting position within <instr>. The first character of <instr> is specified as position1. If position is negative, then it specifies a position relative to the end of <instr>. The final character of <instr> is designated by the value -1.

<length> this argument is optional, and specifies the number of characters to extract. If length is negative, then position specifies the last character of the extracted string, and extraction proceeds to the left instead of to the right. If omitted, the function extracts the entire remainder of <instr> up to its final character.

Examine the following examples:

```
$substr("HELLO", 3)
```

yields "LLO"

```
$substr("HELLO", 3, 2)
```

yields "LL"

```
$substr("HELLO", 3, 7)
```

yields "LLO"

```
$substr("HELLO", 3, -2)
```

yields "EL"

```
$substr("HELLO", -2, 2)
```

yields "LO"

```
$substr("HELLO", -2, 1)
```

yields "L"

```
$substr("HELLO", -2, -2)
```

yields "LL"

$trim, $ltrim, and $rtrim

There are three further string functions that are used to strip leading and trailing blanks from character strings. All three functions take a single string argument and return a string with the blanks removed. The $trim function removes both leading and trailing blanks; the $ltrim function removes leading blanks from its argument, and the $rtrim function removes trailing blanks, but leaves leading ones.

Examine these examples:

$trim(“ removes leading and trailing blanks “)

yields “removes leading and trailing blanks”

$ltrim(“ removes leading blanks “)

yields “removes leading blanks “

$rtrim(“ removes trailing blanks “)

yields “ removes trailing blanks”

The $rtrim function is especially useful for stripping trailing blanks from CHAR data that is retrieved from a database, thus converting it to a VARCHAR representation of the same string.

$toupper, $tolower - PrimeBase Extersion

These functions are used to convert the characters of a string to upper- or lower-case. The function $toupper convers ASCII extended characters to upper-case using the Macintosh character set. However, $tolower converts to lower-case as C programmers would expect, ignoring extended characters. $toupper can be used to perform case-insensitive comparisons in the same manner as PrimeBase when comparing database object names.

Some examples:

```
$toupper(“1) This is Upper-case!“)
```

yields “1) THIS IS UPPER-CASE!”

```
$toupper(“2) Special characters: ä, é, ù.“)
```

yields “2) SPECIAL CHARACTERS: Ä, É, ù.“

```
$toupper(“3) This IS UPPER-CASE!“)
```

yields “3) this is lower-case!“

```
$tolower(“4) SPECIAL CHARACTERS: Ä, É, ù.“)
```

yields “4) special characters: Ä, É, ù.”

Notice that `the character 'ù'` has no upper-case equivalent .

CURSOR FUNCTIONS

These functions return such information as the length of a column, the type of the column, the name of the column of a particular cursor. As extensions to standard DAL, PrimeBase provides functions to manipulate the actual contents of a cursor rowset. All functions operate on the system cursor. $cursor, if no cursor is explicitly specified.

Aside from passing a cursor identifier as one of the optional parameters, column functions can also take an integer or a string as a calling parameter. In this case, an integer would refer to the column's position, and a string would refer to the name of the column. The column functions are:

- $collen — returns the length, in number of bytes, of the data object that the column contains.
- $colname — returns the name of the column.
- $colplaces — returns the number of decimal places in the column if the data type is decimal or money.
- $coltype — returns a number from 1-12, indicating the type of the column.

- $colwidth returns the field width associated with the column of a cursor.
- $cols returns the number of columns contained in the cursor.
- $rows returns the number of rows contained in the cursor.
- $insertrow add a row to the cursor rowset at the current position.
- $updaterow alter the fields of the current row of a rowset.
- $deleterow delete the current row of a rowset.
- $currentrow return the absolute position of the current row of the cursor rowset.

The syntax of the column functions is explained as follows:

VARIABLE FUNCTIONS

These functions return information on DAL variables

Aside .. are:

- $typeof returns the data type of the selected variable.
- $len returns the length the same way as $collen, but is The syntax of the column functions is explained as follows:

$len

This function returns the length of the argument, which must have a string data type: CHAR, VARCHAR, VARBIN. The function then returns the number of characters or bytes in the string. The example shows the use of this function:

```
declare char greeting1 = “Good ”;

declare char greeting2 = “Morning”;
```

```
declare varchar greeting3;

greeting3 = greeting1 + greeting2;

print $len(greeting3);
```

This example outputs the integer 11, which is the number of characters in the concatenated string.

$typeof

This function returns the data type of its argument. This can be an expression of any of the valid data types. It returns an INTEGER value which is the DAL code for the data type. A list of the DAL data type codes can be found in chapter Data Types in the User Manual.

```
declare date mydate = “13/9/1993”;

declare varchar new = “Birthday”;

declare generic gen1;

gen1 = mydate;

print $typeof(mydate), $typeof(new), $typeof(gen1);
```

The example outputs the three integers 6 (for DATE), 12 (for VARCHAR) and 6 (for DATE).

$FORMAT FUNCTION

This function takes a variable number of arguments, and combines them into a VARCHAR string according to a user-specified format. The syntax is as follows:

```
<outstr>::= $format '(' <fmtstr> { ',' <data> } )'
```

parameters <outstr> the output string

$format	the name of the function
<fmtstr>	a string expression specifying the format to be used. The specific contents of <fmtstr> determine the number of data items.
<data>	any number of data items.

The Format String

The format string (<fmtstr>) may consist of three formatting specifications. They are interpreted in a left to right order. These formatting specifications are:

1. Ordinary characters: They simply stand for themselves, and are appended to the end of the out string (<outstr>).

2. Special characters: They are non-printing characters - for example a tab. Special characters consist of a two character sequence: a backslash and a second character.

3. Conversion specifications: These are specifications that take the following data argument and convert it into a sequence of characters that are appended to the end of the out string.

Special Characters: Two-Character Sequences

• \ b	generates a backspace character.
• \ n	generates a newline character.
• \ r	generates a return character.
• \ t	generates a tab character.
• \ \	generates a backslash character.
• \ %	generates a percent character.

Conversion Specifications

The percent character (%) introduces all conversion specifications and is concluded by a conversion character (<fmtchar>). See the syntax below:

```
<conversion_spec>::= '%' ['-'] [<width>] ['.' <precision>] <fmt-
char>
```

Minus Sign

The minus sign, when included, justifies the data to the left upon conversion. Otherwise the data is right justified. Justification is, of course, relative to width and precision - which specify the width of the converted data string.

Width and Precision

Specify the width and precision of the converted data string in the fields <width> and <precision>. Remember to include the period (.) preceeding the value you give as the <precision>. You can specify width and precision as the special character, asterisk (*). Then the numeric value for the parameter is taken from the next data argument to the $format() call.

Format Character Values

The values that you can use in <fmtchar> are as follows:

- d or u — This is an integer data argument. It converts to a string, whose value is given in the integer literal format.
- p — This is a decimal or money data argument. It converts to a string, whose value is given in the decimal literal format.
- c — This is a single character data argument. It copies the character.

- f — This is a floating-point data argument. It converts to a string, whose value is given in the floating-point literal format.
- s — This is a string data argument. It copies to a string, according to the other conversion parameters.
- x or X — This converts its data argument to a sequence of characters that are the unsigned hexadecimal representation of the data item. A capital X results in uppercase hex digits (A-F); a lowercase x results in lowercase hex digits (a-f).

When the <fmtchar> is preceded by:

- ^ (caret) — the output field contents are to be converted to uppercase characters.
- ! (exclamation point) — the output field contents are to be converted to lowercase characters.

For every call to the $format() function, the number of conversion specifications in <fmtstr> and the number of data arguments must match. If the number of data arguments is less than the number of conversion specifications, then errors may result.

FILE FUNCTIONS - PRIMEBASE EXTENSIONS

These functions allow the user to directly read and write files from a DAL program. The main purpose of these functions is to import and export data to and from PrimeBase in text format.

The file manipulation functions are:

- $open — open a file.
- $close — flush and close a file.
- $readline — returns the next line.

- $writeline writes a line to a file.

$open

Open a text file for reading and writing. This function takes as input a string value, that represents the path of the file to be opened. Returned is an integer value which is to be used as a file handle for subsequence read and close operations. Note that there is no special file type, and normal integer is used as file handle. Files opened are globally accessable, and as a result, files opened in procedures are not automatically closed on return from the procedure. Currently, a maximum of 10 files may be opened. Valid file handles may range from zero to 9. When a session is closed, all open file are close as well.

$close

This function takes as argument an integer value which was returned by $open. The file associated with the integer is closed, and resources for accessing the file are freed.

$readline

This function returns each line of an open file. It takes as argument an integer value which was returned by $open. It returns a VARCHAR value that is the next line of text of the file. The end-of-line indicator is NOT returned as part of the string. There is no limit to the length of a line. $readline return $NULL when the end of file is reached. This function will read text files created by Macintosh, UNIX or DOS machines.

$writeline

This function writes a line to a text file. The syntax is as follows:

```
$writeline '(' <handle> ',' <line> )'
```

parameters

$writeline	the name of the function
<handle>	the integer handle returned by $open
<line>	the line to be written.

The function writes the line to the file ands the appropriete end-of-line characters.

UTILITY FUNCTIONS - PRIMEBASE EXTENSIONS

Utility functions are useful extensions to DAL. These functions include:

- $now returns the current time.
- $errorstring returns the string associated with an error number.

$now

This function returns the current time as a TIMESTAMP value. It takes no arguments. Note that this is the time on the client machine, not the server. Server time can be obtained after login (see system variable $logintime).

$errorstring

This function takes a single integer arguments, and returns the associated error string. The syntax is as follows:

```
<outstr>::= $errorstring '(' <errno> )'
```

parameters	<outstr>	the output string error string
	$errorstring	the name of the function
	<errno>	an error number.

SYSTEM VARIABLES

Date and Time Formats

The date and time system variables have been extended to include more options for the formatting values of these types. This has been done without losing any compatibility with formats supported by DAL.

Date

• •$month

The system variable, $month, is a string of 12 month names, each separated by a '+'. When a session is created, the following is true:

$month = “Jan+Feb+Mar+Apr+May+Jun+Jul+Aug+Sep+Oct+Nov”

Please note that spaces are significant in this string.

• $day

The system variable, $day, is a string of 7 day names separated by a `+´. If the $day string is empty (i.e. the empty string, or NULL), a 3 digit Julian day (number of the day in the year) is returned. On session startup, $day contains an empty string in order to be compatible with DAL.

Please note that spaces are significant in this string.

• $datefmt

The following tokens are recognised in $datefmt. Case is NOT significant.

YYYY	a 4-digit year.
YY	a 2-digit year.

MMM	substitute $month.
MM	a 2-digit month.
M	a 1 or 2-digit month.
DD	a 2-digit day.
D	a 1 or 2-digit day.
DDD	substitute $day.

For DAL compatible startup, $datefmt = “MM/DD/YYYY”.

TIME

- $ampm

The system variable, $ampm, is a string containing two names separated by a '+'. The first name is the sign for morning (AM), and the second is the sign for evening (PM). For DAL compatibility, set $ampm to “AM+PM”.

- $timefmt

The following tokens are recognised in $timefmt, Case is NOT significant, except for “xm” and “XM” (see below).

HH	2-digit hour: 24 hour time by default; 12 hour time if xm or XM are included in the time format.
H	1or 2-digit hour.
MM	2-digit minutes.
M	1 or 2-digit minutes.
SS	2-digit seconds.
S	1 or 2-digit seconds.
HU	2-digit hundredths of a second.
T	1-digit tenths of a second.

XM | AM | PM substitute uppercase $ampm.

xm | am | pm substitute lowercase $ampm.

For DAL compatible startup, the following format is valid: $timefmt = “HH:MM:SS:hu”

Date and Time

• $tsfmt

This format string includes both date and time tokens. Without losing compatibility with DAL, PrimeBase allows the $tsfmt variable to be empty (the empty string: “”), or NULL. In this case, the $datefmt and $timefmt are used, in place of a $tsfmt, in that order and a space is placed between the two.

There are some problems with the DAL definition of this format, yet these are solved with a non-compatible PrimeBase extension, which can be turned on or off:

In DAL compatible mode the PINKCTL parameter 2 is set to zero. For example,

```
PINKCTL , 0;
```

In this mode, the format before the first space in the timestamp format must be a date format, and the format after the first space must be a time format. Note that in this way, “MM” or “M” appearing before the first space represent months, and “MM” and “M” appearing after the first space represent minutes.

PrimeBase EXTENSION: This extension is not DAL compatible because it makes the month and minute tokens case-sensitive in the timestamp format. For example,

```
PINKCTL , 1;
```

both date and time tokens may be freely mixed. Month and minute tokens are distinguished by the case of the first “m”. Upper-case indicates months and lowercase indicates minutes. Thus “MM”, ““Mm” and “M” represent months, and “mm”, “mM” and “m” represent minutes. Please note that the “MM” token is still case-insensitive in the $timefmt and $datefmt system variables.

Decimal and Money Formats

The decimal and money format variables describes here are not part of standard DAL; the system variables described in this section are extensions to DAL.

• $decfmt

The decimal format system variable will accept a string with the following format:

```
<decfmt>:: = [ '[' ] '9' [ <tchar> ] '999' [ ']' ] [ '[' ]
<dchar> <fchar> [ <ichar> ] [ ']' ]
```

In this description, ' [' and '] ' indicate the string elements that are optional. The characters enclosed in single quotation marks (' ') are meant literally. The syntax, <tchar>, <dchar>, <fchar> and <ichar> are all single characters, that you select. The various elements of the <decfmt> string have the following meaning:

<tchar> The user defined character, <tchar>, is the thousands separator. If <tchar> is omitted, no thousands separator is used.

<dchar> The user defined character, <dchar>, is the decimal point. This character must be specified, and must be different to <tchar>.

<fchar> The <fchar> character indicates how to display the character immediately after the decimal point. The characters `9´ and `0´ indicate that the digit should always be displayed. The underscore character indicates that the character is not to be displayed if the fractional part of the number is zero. The characters, ' ' (a space), and ' * ' (the asterisk), may also be used if the fractional part of the number is zero.

<ichar> The <ichar> character is the insignificant fraction digits character. Insignificant fraction digits are the trailing zero digits of the fraction: for example, in the value 1.2000, the

characters '000' are the insignificant zero digits of the fraction. The number of such digits depends on the scale of the decimal number (and the number of significant fraction digits). These digits are printed as the user defined character <ichar>. (<ichar> must be set to '0' to print the insignificant fraction digits as zeros!) <ichar> may also be omitted, or the underscore character can be set; in this case, insignificant fraction digits are not displayed. <ichar> can also be: ' ' (space), '0', or '*'.

The first optional '[' ']'

The first optional set of brackets indicate whether the whole number part of the number should be displayed, if it is zero. By placing the whole number part between the brackets, you indicate that it is optional and therefore not displayed when the value is zero.

The second optional '[' ']'

By placing the fraction between the brackets (including the decimal point), you indicate that this part is optional, and should not be displayed when the value is zero.

Note: Both the whole number and fractional parts are optional, a zero will still be printed as '0'.

For DAL compatible startup, the following format is valid: $decfmt = “[9999].9”

• $moneyfmt

The system variable, $moneyfmt accepts an ASCII string of the following form:

```
<moneyfmt>::= [ '(' ] [<cstring>] [ '(' ] [ '-' ] <decfmt> [
')' ] [ <cstring> ] [ ')' ]
```

Parentheses: '(' ')' All parentheses are optional. If they appear, they indicate that negative money values should be displayed in parentheses. You can indicate whether the brackets

should include the currency string (<cstring>) or not. When parentheses appear the negate sign is not permitted, see below.

<cstring> — The <cstring> is a user defined string of characters, which may optionally be enclosed in double or single quotation marks. The string indicates which currency symbol is to be used. Spaces are significant in this string. Either the first or second <cstring> must appear, but not both. The position of <cstring> indicates the position of the currency symbol: whether it comes before or after the currency value.

The negate sign: '-' — The negate sign is optional. When it appears, it indicates that negative values are to be displayed with a leading `-´character. Please note that either the optional parentheses can be specified, or the negate sign, but never both. If neither is specified, then the negate sign is assumed by default.

<decfmt> — This is a decimal format string, as has already been described under the heading $decfmt. Please note the following example in this regard: “9,999.9 DM”. In this example, it is impossible to tell whether the space after the 9 is meant to be the <ichar> (the insignificant fraction digits character) or the first character of <cstring> (the currency symbol). Please use the optional quotation marks for the currency symbol in the case as follows:
“9,999,9' DM´”
It is now clear that the currency symbol has a leading space and that insignificant fraction digits are not displayed.

For DAL compatible startup, $moneyfmt = “$[9999].0”

DAL System Variables

- $sqlcode — This variable is automatically set after each execution of each data manipulation statement. Its value is zero if no error occurred and negative for errors.
- $switch — This system variable has the value of the expression of the last SWITCH statement executed. It is unitialised before the first switch statement is executed.
- $maxrows — This variable is initially set to NULL, and can be modified by the SET statement. If it is set, it limits the number of rows in the rowset created by a SELECT statement.
- $cursor — After a successful SELECT statement that has no cursor explicitly specified as destination (i.e. no INTO clause), this variable is set to identify the rowset just created.
- $colcnt — After each successful SELECT statement, this variable is automatically set to the number of columns that have just been created in the rowset.
- $rowcnt — After each successful SELECT statement, this variable is set to the number of rows in the rowset just created, if the number of rows is known. If the number of rows is not known $rowcnt is set to NULL. Note that if the mode of the SELECT statment is not EXTRACT, the number of rows is not necessarly known. Only after a SELECT for EXTRACT, have all rows always been loaded by the client, and as a result the number of rows is known.

DAL System Constants

- $null — a NULL value.
- $sqlnotfound — the value to which $sqlcode is set when a cursor is fetched past the end of rowset.

- $version — a string containing the PrimeBase version of the client software.
- $true — the boolean value *true*.
- $false — the boolean value *false*.
- $boolean — the datatype number (as returned by $typeof) of type *boolean*.
- $smint — the datatype number of type *smint*.
- $integer — the datatype number of type *integer*.
- $smfloat — the datatype number of type *smfloat*.
- $float — the datatype number of type *float*.
- $date — the datatype number of type *date*.
- $time — the datatype number of type *time*.
- $timestamp — the datatype number of type *timestamp*.
- $char — the datatype number of type *char*.
- $decimal — the datatype number of type *decimal*.
- $money — the datatype number of type *money*.
- $varchar — the datatype number of type *varchar*.
- $varbin — the datatype number of type *varbin*.
- $longchar — the datatype number of type *longchar*.
- $longbin — the datatype number of type *longbin*.

Lock Settings - PrimeBase Extension

Global variables are used to control various aspects of a locking. The current setting of these variables determines the parameters of the next transaction to be started for the session. A transaction can be started explicitly with the BEGIN command, or implicitly when a SELECT, INSERT, UPDATE or DELETE is executed.

- $locktimeout — This variable contains the time in hundredths of seconds that the system should wait for a lock. When set to zero, the Server will return an error to the user as soon as the user (user´s transaction) attempts to acquire a lock on data that conflicts with locks held by other transactions. If the variable is set to NULL the Server will wait as long as it takes to acquire a lock. Any other positive value causes the Server to wait that number of 100th of a second before returning a locked error. The default for this variable is zero.

- $aborttime — The abort time is the time that the server allows a transaction to remain idle. A transaction is considered idle when it is not reading, writing or sorting. Abort time is specified in seconds. The upper-limit (and default value) for abort time stored per user in the SysUsers table in each database. If more than one database is open on a particular connection, then the upper-limit is the minimum of all default values. It is recommended that abort time be set to, at least, 2 seconds. Normally abort time is quite high, for example 10 minutes. The system default is 30 minutes. The purpose of a abort time is to ensure that a transaction does not consume server resources and do nothing. This can occur if a complex deadlock has occurred (a deadlock involving more the 2 transactions), or if the client application has forgotten

about the transaction for some reason (for example, an application begins a transaction, and then the user switches to another application).

- $rowlocking — This variable is used to turn row locking on (set to $true) or off (set to $false). When row locking is off, the server gains table level locks for every table accessed by the transaction. When row locking is on, the server locks only the rows accessed by the transaction. Row locking should not be used in transactions that access a large number of rows (over 2000), or most rows in a particular table, because each row lock requires about 28 bytes of memory on the server.

DBMS Lookup Parameters - PrimeBase Extension

These global variables are used to control the behaviour of the DESCRIBE DBMS statement. PrimeBase equates a server with the DAL concept of a DBMS. As a result, the DESCRIBE DBMS statement lists all servers/gateways that can be accessed by the client. If the variables below are set to NULL, then all possible servers are listed. The variables can be used to limit the search to server in a particular zone, etc.

- $dbmszone — If not NULL, then it specifies the AppleTalk zone to be searched for servers by the DESCRIBE DBMS statement.
- $dbmsbrand — This variable can be used to search for a particular server/gateway type. For example, when set to 'Sybase', the DESCRIBE DBMS statement will search for PrimeBase gateways to Sybase. To lookup PrimeBase servers, set this variable to 'PrimeBase'.
- $dbmsprotocol — Servers and gateways may publish itself on one or more protocols, for example, ADSP (Apple Data Stream Protocol) and TCP/IP (Transport Control Protocol/Internet Protocol). When set to NULL, the client will search for all

servers published on protocols that the client is capable of. Setting this variable to 'adsp', would limit the search to servers capable of communicating using ADSP. Other possiblities are: 'tcp' (TCP/IP), 'ppc' (Macintosh Program-to-Program communications) and 'ipc' (UNIX interprocess-communication, using shared-memory). Note that an error will occur if the client is not capable of the specified protocol.

Login Information - PrimeBase Extension

After the client has successfully established a connection to a server certain information regarding the connection/login is made avialable to the client application. This information is stored in the following global variables:

- $logintime — The time on the server machine at login is recordedin this variable. This value can be used by the client to syncronise its time with that of the server after login.
- $connid — The server connection identifier is stored in the variable after login. The value can be used by the client to find it's connection entry in the SysConnections system table in the Master database.
- $user — Stored in this variable is the name of the user of the last successfull connection to a server. If a user name is not specified in subsequent connects (OPEN DBMS command), then the value stored in $user is used.

Cursor Information - PrimeBase Extension

The following variable are extensions to the DAL standard.

• $rowsperpage	This system variable allows you to control the number of rows in a page received from the Server. Normally the variable is set to zero or NULL, in which case the number of rows is estimated by the system. The effect of setting this variable is not visible from the DAL program itself. NOTE: The size of the pages to be used is only calculated when the SELECT is done; subsequent pages fetched for the SELECT are not resized according to $rowsperpage (i.e. make sure that $rowsperpage is set before the SELECT is done).
$rowsaffected	This variable is set after a successful INSERT, UPDATE, or DELETE. The value is the number of rows affected by the query. If an error occurs, $rowsaffected is set to zero.

SYSTEM PROCEDURES

System procedures are used to initiate an operation performed by the server. Master database specific objects such as devices, locations and partitions can be created and deleted using system procedures.

SYNTAX

Server procedures all have the same syntax, consisting of two keywords followed by parameters in parenthesis.

syntax

```
<identifier> <name> '(' [<proc_input_spec>] ')'
[<proc_output_spec>] ';'

<proc_input_spec> ::= <required_params> {<optional_param>}

<required_params> ::= <server_param> {',' <server_param>}

<server_parame> ::= <name> '=' <expression>

<optional_param> ::= ',' [ <name> '=' ] <expression>

<proc_output_spec> ::= <returning_variables> | ( INTO <cursor> )
```

parameters

<identifier>	an identifying keyword.
<name>	an identifying keyword or character expression.
<proc_input_spec>	specification of procedure input.
<proc_output_spec>	procedure output specification.

notes

Server procedures define a generic syntax for services performed by the server. The actual services provided depend on the server type, and version. There are a number of possibilities regarding the output of server procedures:

1. The procedure may return no values. In this case the <proc_output_spec>, if any is ignored.
2. The procedure returns values. If a <returning_variables> clause has been specified, then the values are placed in DAL variables. Additional values are ignored. If no <returning_variables> clause was specified, the values are printed to the output stream.
3. The procedure returns a rowset. If an INTO clause is supplied, then the rowset goes to specified cursor, otherwise into the default cursor ($cursor).

   In all cases, no output is generated if an error occurs during execution of the server procedure. Error details can be retrieved as all other DAL errors.

   All server procedures have a set of required parameters, and a variable number of optional parameters.The required parameters must be passed in the correct order before the optional parameters. Optional parameters must be identified using the <name> = syntax of server procedures.

   Optional parameters, in the case of *add* type procedures have default values which are used when the value is not specified. Optional parameters not specified in *alter* type procedures leave that particular aspect unchanged.

   All names of parameters, and values given as input to system procedures are case-insensitive, unless otherwise noted.

DEVICES

A device definition is the basic requirement of the server to access data residing on permanent storage. Device descriptions tell the server the type of the device, the location within the file system of the host, and any other information required to access data stored on the device. Device descriptions are stored in the SysDevices table in the master database. The following server procedures are provided for maintaining devices:

Add Device

Add a device description to the SysDevices table. The device can subsequently be used to create locations, specify partitions, or locate log files. Exactly which types of devices, and which options are supported, depends on the server version, and host platform.

Required parameters:

NAME	The name of the new device.
PATH	A string identifying the location of the device within the host filing system. This path may be relative to the location of the server application. This value may be case sensitive, depending on the host file system.

Optional parameters:

TYPE	(default: FileSystem) The type of device. Determines the internal device driver used by the server.
REMOVEABLE	(default: FALSE) True if the device contains removable media.
READONLY	(default: FALSE) True if the device is readonly.
RANDOMACCESS	(default: TRUE) False if the device does not support random seek operations.
USEABLESPACE	(default: $NULL) A value in bytes that determines the maximum amount of space the server may use on the device. $NULL means the space used is only limited by physical size of the device.
ACCESSSTRING	The access string is device specific information required by the server's device driver in order to access the device.

Alter Device

Alter a device definition. Note that certain aspects of a device cannot be altered.

Required parameters:

ID	The device identifier as it appears in the SysDevices table. When a device is added, it is automatically allocated a unique identifier by the system.

Optional parameters:

NAME	Change the name of the device.
REMOVEABLE	Determines if the device contains removable media.
READONLY	Determines if the device is readonly.
RANDOMACCESS	Indicates whether the device supports random seek operations.
USEABLESPACE	Determines how much space may be used by the server on the device.
ACCESSSTRING	Change the device specific access string.
PATH	Change the device path.

Remove Device

Devices cannot be removed if there are any databases/backups or logs currently located on the device. If so, an error is returned, indicating that the device is in use.

Required parameters:

ID	The device identifier.

LOCATIONS

Locations indicate storage and backup areas of various types of files used by the server. Current locations are searched for objects (databases, backups, log, etc.) already existing. Future locations are used when objects are created, to determine where the files should be located.

Add Location

Add a location that indicates the placement of various database and server control files.

Required parameters:

NAME	The name of the location.
FILETYPE	The type of file to be stored in this location. Current possibilities are: Data, Index, Blob, Log, Restart, VM.
FILEPURPOSE	Indicates the purpose of files stored (or to be stored) in the location. The following are valid purposes: CurrentStorage, FutureStorage, CurrentBackup, FutureBackup.
DEVICEID	The device on which the location resides.

Optional parameters:

GROUPNUMBER	(default: $NULL) The group number of the location. Location groups are only used for future locations. When creating new objects, if the modulus maximum group number of the group number equals the minor identifier of the new object, then the location is used to locate the new object.
ALLOCATION	(default: Automatic) Allocation determines the allocation strategy to be used by the server when creating objects using the given location. Possibilities are: *Automatic*: The location is automatically a candidate for the creating of new objects. *Manual*: The server never creates objects in this location, however it does search for existing objects in the area. *Default*: This location is only considered for creation of an object if there are no other candidate locations.
MEDIANUMBER	(default: $NULL) The media number of a specific media associated with the device.
COMMENTS	(default: "") Description of the location or any other user specific information.

Alter Location

Alter an existing location.

Required parameters:

ID	The location identifier of an existing location.

Optional parameters:

NAME	Change the name of the location.
FILETYPE	Change the type of files stored in the location.
FILEPURPOSE	The new purpose of the location.
DEVICEID	The device identifier of an existing device.
GROUPNUMBER	New group number.
ALLOCATION	Change the allocation strategy for the location.
MEDIANUMBER	The media number of a particular media accosiated with the device.
COMMENTS	Change the comments on the location.

Remove Location

Delete a location.

Required parameters:

ID	The location identifier of an existing location.

PARTITIONS

Partitions divide databases and backups into various locations. Each database/backup file will be located completely within a particular partition. Types of database files are: Data, Index and Blob (Binary large objects). Server partition are logical entities that have no physical affect on the device.

Add Partition

Add a parition to a particular database or backup.

Required parameters:

DATABASEID	The identifier of an existing database.
DEVICEID	The identifier of the device on which to place the partition.

Optional parameters:

BACKUPNUMBER	(default: $NULL) The backup number, if the devision is intended for a particular existing backup.
DATA	(default: $TRUE) True if data type files should be stored in this partition.
INDEX	(default: $TRUE) True if index type files should be stored in this partition.
BLOB	(default: $TRUE) True if blob type files should be stored in this partition.
MEDIANUMBER	(default: $NULL) The media number (if any) of a particular media associated with the device.
ALLOCATION	(default: Automatic) The allocation strategy used by the server when using this partition: *Automatic*: The location is automatically used to locate/create database files. *Manual*:- The server never creates files in this partion, however it does locate existing database files. *Default*: This partition is only used if there are no others.

Alter Partition

Change certain parameters of an existing partition. Changing the types of files that may be stored on a partition will not change the location of existing file on the partition. However, the server will no longer find certain files, depending on how the locations is altered.

Required parameters:

ID	The identifier of an existing partition.

Optional parameters:

DATA	Determines whether data type files can be stored in this partition.
INDEX	Determines whether index type files can be stored in this partition.
BLOB	Determines whether blob type files can be stored in this partition.
ALLOCATION	Alter the allocation method for this partion.

Remove Partition

Delete a partition. Deleting a partition does not delete the files on the device.

Required parameters:

ID	The identifier of an existing partition.

SYSTEM PARAMETERS

TransactionLimit

The transaction limit is the maximum number of transactions that the server can process concurrently. If the a transaction is begun, and the server has no transactions available, the user will be returned the error 'Too many active transactions'. There should be approximately one transaction available per connection. The minimum transaction limit is 32, and the maximum is 255.

SystemFileLimit

The system file limit is the number of system file handles the server will use. When the server has consumed this number of file handles, it will recycles its file handles on a least-recently-used basis. In addition to the file handles used by the server, one file handle is required to access the environment file, and one is required per 'execute file' command entered from the console. If this value exceeds the actual number of files available to the server (as provided by the system), it is possible that users will occasionally receive the 'Too many files open' error. If this occurs, set the SystemFileLimit down by one or two files. The default is 240 on all platforms.

LogBufferSize

This is the size in bytes of the log buffers. The log buffers cache the data to be written to the logs. Before the log is flushed, the contents of the log buffer is written to the log file. The log must be flushed when a transaction is committed, or

when the log buffer is full. Large log buffers can improve the performance of long running transactions. The transaction manager will allocate at most 2 log buffers, one for each online log file. The mimimum size for the log buffer is 32K.

LogThreshold

The log threshold is the point at which a new online log is created. It is a size in bytes. When the current active log (this is log with the highest number), reaches this size, an new log is created. The new log becomes the active log, and the old log becomes inactive. Transaction always begin writing to the active log. Transactions cannot change the log to which they write. This means that transactions that began when the now inactive log was active continue to write to the log when it becomes inactive. Both the active and the inactive logs are called the 'online logs'. When the inactive log has no more transaction writing to it, it is taken offline. An offline log is no longer required by the dataserver for recovery purposes, and if the offline log function is set to 'delete', then it will be deleted by the server. If the inactive log grows to 150% of its threshold it is forced offline. Transactions still writing to the inactive log when this occurs are aborted, and rolled back.

CheckpointThreshold

After CheckpointThreshold bytes have been written to the log, the server writes a checkpoint record to the log. The more frequently a checkpoint is written, the less time the server takes to restart after the server application was unexpectedly quit. The time taken to restart, however also depends on the size of the server disk cache memory (CacheSize). The more information cached when the server was quit, the more time required to restart. Writing a checkpoint record does noit take much time.

CacheSize

Cache size is the maximum amount of memory, in bytes, used by the server to cache records read from disk. Increasing the cache size improves the performance of the server, but may also increase time required to restart the server if the server is not shutdown correctly. Cache memory is taken from the total memory allocated (or available) to the server. If set too high, insufficient memory may remain for the correct operation of other parts of the server. To be safe about 256K should be available per connection, after cache memory and virtual cache memory have been subtracted from total available memory.

VirtualCacheSize

Virtual cache size is the maximum amount of memory in bytes used by the virtual memory manager. When the virtual memory manager has no more physical memory available to it, it begins to swap data out to disk. The servers virtual memory system is used to store intermediate result sets during the execution of queries. Data being sorted is also maimtained in server virtual memory. Certain queries will execute much faster when enough physical memory is available to the virtual memory manager.

OfflineFunction

This variable determines what happens to the inactive log when it is moved offline. There are only two permissable values: 'Delete' and 'Archive'. After installation, the ofline log function is set to 'Delete'. Offline logs are only required to restore a database from backup. If no backups have been made, the offline logs are not required. In this case, the offline log function can be set to 'Delete', which causes the server to automatically delete logs as they are moved offline. If the log function is set 'Archive', the offline logs are moved to the log archive location. If the server requires a log when restoring a database, it looks in the location in which the log was archived.

DataServerName

This variable contains the of the server used by client applications to access the server. This name is published, and is visible over the network on protocols such as ADSP and PPC. When set, the server changes the network visible name immediately. Connected clients are not affected by change of the server name.

ConnectionLimit

The connection limit is the current number of connections permitted by the server. This number may range from zero to ConnectionTotal (see below). Increasing this variable immediately makes more connections available for client. If the connection limit is zero, the server is not published (is not visible) over the network. Decreasing the value of connection limit will decrease the number of connection available to clients. If the number is decreased below the number of clients currently connected to the server, some of the client connections will be terminated by the server.

ConnectionTotal

This variable contains the maximum number of connections permitted according to the registration license of the server. The connection limit cannot be set to a value greater than the connection total. Connection total is set by the installation program, and may be increased later using the incrementor program. In each case, a serial number containing the number of connections permitted is required.

SerialNumber

The serial number of the server. This value must be provided upon installation.

ActivationKey

A valid activation key is required to register a server. Without a valid activation key the server runs in demostration mode. In demostration mode the server shuts down after 2 hours of operation.

ExpiryDate

The expiry date of the server determines how long the server will run as a registered server. Setting this variable to $NULL indicates that there is no expiry date. After the expiry date, the server runs in demostration mode. Before the expiry date, the server runs as a registered server.

IdentificationString

The identification string contains the characteristics of an installed server that are required for registration. The value of this variable, along with the serial number must be sent to SNAP Innovation in order to register a server.

InitialMemoryBlockSize

The initial memory block size is the size of the initial memory block allocated by the server in bytes. This is one of three variables used to control the amount of memory used by the server.

Memory is allocated by the server from the system in blocks. The memory in these blocks is then managed by the server itself. The server allocates memory from these blocks using a very fast best-fit algorithm. This memory management is faster and more effecient than any operating system memory management we have tested so far.

When the server if finished using the memory in a block it frees the memory to the system. The size indicated by this variable is the size of the initial block of memory allocated by the server. It may be larger the subsequent blocks allocated. In addition, the initial memory block is never freed to this system while the server is running. This block is managed like all other blocks, and therefore must be calculated as part of the block total (see below). Setting this variable has no immediate effect on the size of the initial memory block. Only when the server is started up again, will the new initial memory block size be used.

You should set this variable to the mimimum amount of memory you wish the server to use. On the macintosh, make sure that enough memory is allocated to the server application that the server can allocate the initial memory block. The server will not start if it cannot allocate this block.

By setting this variable to NULL, you indicate to the server that the memory variable should be automatically configured. On the Macintosh this is the optimal setting, as the servers memory parameters are then automatically set according to the memory limit given to the server application in the finder. Under UNIX the automatic memory configuration causes the system to allocate memory until the system says there is no more.

MemoryBlockSize

This is the size, in bytes, of all blocks of memory allocated by the server from the system besides the initial memory block (which has size InitialMemoryBlockSize). Changing this variable has no affect on blocks already allocated, but subsequent blocks will be allocated using the new size. The server only allocates memory blocks when cannot find a free segment of memory of the required size amongst the blocks that it already has. If the segment of memory that the server wishes to allocate exceeds the size of memory blocks the server will try to allocate a block of the required size from the system. This means that MemoryBlockSize is, in fact, a minimum block size. As a rule, however, the server does not require memory segments of much larger than 64K. One exeption to this are the log buffers, whose size may be set by the system administractor (see LogBufferSize).

MemoryBlockTotal

This is the total number of memory blocks (including the initial memory block) that may be allocated by the server. After the server has allocated this number of blocks, the server will report an 'Out of Memory' error. if the system does not allow the server to allocate all its memory blocks, the server may report a 'out of Memory' error sooner. The maximum amount of memory used by the server can be calculated as:

MemTot = InitialMemoryBlockSize + MemoryBlockSize * (MemoryBlockTotal - 1)

Under UNIX it is recommended that MemTot be set such that when allocated, all this memory will reside in physical RAM. The server has its own virtual memory management scheme, and as a result it is better if the memory actually used by the server is real and not virtual memory.

APPENDIX A: SYSTEM DATABASE

MODEL DATABASE

Domains

```
CREATE DOMAIN System.ObjectTypeCHAR(4), ORDER NOT APPLICABLE AS
CASE INSENSITIVE; /* 1 */
CREATE DOMAIN System.ObjectIDINTEGER;
CREATE DOMAIN System.DBObjectID(ObjectType, ObjectID);


CREATE DOMAIN System.ColumnIDSMINT;
CREATE DOMAIN System.DBColumnID(ObjectType, ObjectID, Colum-
nID);


CREATE DOMAIN System.ComponentIDSMINT;
CREATE DOMAIN System.DBComponentID(ObjectType, ObjectID, Compo-
nentID);


CREATE DOMAIN System.DBColCompID(ObjectType, ObjectID, Colum-
nID, ComponentID);


CREATE DOMAIN System.SysNameVARCHAR(31), ORDER NOT APPLICABLE AS
CASE INSENSITIVE;
CREATE DOMAIN System.DBName(SysName, SysName);


CREATE DOMAIN System.UserIDINTEGER;
CREATE DOMAIN System.DataTypeSMINT;/* 12 */


CREATE DOMAIN System.UserType CHAR(3), ORDER NOT APPLICABLE;
CREATE DOMAIN System.UGID (UserID, UserID);
```

```
CREATE DOMAIN System.KeyAction CHAR(4), ORDER NOT APPLICABLE;
CREATE DOMAIN System.PrivilegeID (ObjectType, ObjectID, UserID,
UserID);
CREATE DOMAIN System.ColPrivID (ObjectType, ObjectID, ColumnID,
UserID, UserID);
```

Tables

1. SysUsers

```
CREATE TABLE System.SysUsers
(
                ID           UserIDNOT NULL,
                Name         SysNameNOT NULL,
                CreatorID    UserIDNOT NULL,
                CreationTime TIMESTAMPNOT NULL,
                UserType     UserTypeNOT NULL,
                Resource     BOOLEANNOT NULL,
                CreatorName  SysNameNOT NULL,
                Password     VARCHAR(11)NULL,
                AbortTimeout INTEGERNULL,
                LoginCnt     INTEGERNOT NULL,
                LastLogin    TIMESTAMPNULL,
                OnlineTime   INTEGERNOT NULL
);
```

2. SysMembers

```
CREATE TABLE System.SysMembers
(
                UserID       UserID NOT NULL,
                GroupID      UserID NOT NULL,
```

```
                    UGID          (UserID, GroupID) UGID
);
```

3. SysDataTypes

```
CREATE TABLE System.SysDataTypes
(
DataType          DataType      NOT NULL,
Name              SysName       NOT NULL,
Scale             BOOLEAN       NOT NULL,
Length            BOOLEAN       NOT NULL,
Size              SMINT         NULL,
Comments          VARCHAR(120)  NULL
);
```

4. SysObjects

```
CREATE TABLE System.SysObjects
(
Type              ObjectType    NOT NULL,
ID                ObjectID      NOT NULL,
CreatorName       SysName       NOT NULL,
Name              SysName       NOT NULL,
CreatorID         UserID        NOT NULL,
CreationTime      TIMESTAMP     NOT NULL,
Comments          VARCHAR(120)  NULL,
DBID              (Type, ID) DBObjectID,
DBName            (CreatorName, Name) DBName
);
```

5. SysDomains

```
CREATE TABLE System.SysDomains
(
Type              ObjectType    NOT NULL,
ID                ObjectID      NOT NULL,
Primary           BOOLEAN       NOT NULL,
PrimaryKeyCnt     SMINT         NOT NULL,
ComponentCnt      SMINT         NOT NULL,
DataType          DataType      NULL,
Scale             SMINT         NULL,
Length            INTEGER       NULL,
Nulls             BOOLEAN       NULL,
Arithmetic        BOOLEAN       NULL,
Ordered           BOOLEAN       NULL,
SequenceType      ObjectType    NULL,
SequenceID        ObjectID       NULL,
DBID              (Type, ID) DBObjectID,
DBSequenceID      (SequenceType, SequenceID) DBObjectID
);
```

6. SysDomainComps

```
CREATE TABLE System.SysDomainComps
(
Type              ObjectType    NOT NULL,
ID                ObjectID       NOT NULL,
ComponentID       ComponentID   NOT NULL,
DataType          DataType       NULL,
Scale             SMINT         NULL,
Length            INTEGER       NULL,
DomainType        ObjectType    NULL,
DomainID          ObjectID       NULL,
```

```
DBID            (Type, ID) DBObjectID,
DBComponentID   (Type, ID, ComponentID) DBComponentID,
DBDomainID      (DomainType, DomainID) DBObjectID
);
```

7. SysTables

```
CREATE TABLE System.SysTables
(
Type            ObjectType   NOT NULL,
ID              ObjectID     NOT NULL,
ColumnCnt       SMINT        NOT NULL,
CompColCnt      SMINT        NOT NULL,
FileName        CHAR(8)      NULL,
CheckPending    BOOLEAN      NOT NULL,
ReorgPending    BOOLEAN      NOT NULL,
BackupPending   BOOLEAN      NOT NULL,
RowCnt          INTEGER      NULL,
AverageDirTime  SMFLOAT      NULL,
TotalDirTime    SMFLOAT      NULL,
DirectCnt       INTEGER      NULL,
AverageSeqTime  SMFLOAT      NULL,
TotalSeqTime    SMFLOAT      NULL,
SequentialCnt   INTEGER      NULL,
DBID            (Type, ID) DBObjectID
);
```

8.SysColumns

```
CREATE TABLE System.SysColumns
(
Type            ObjectType   NOT NULL,
```

```
ID              ObjectID     NOT NULL,
ColumnID        ColumnID     NOT NULL,
Name            SysName      NOT NULL,
Title           VARCHAR(64)  NULL,
ComponentCnt    SMINT        NOT NULL,
Comments        VARCHAR(120) NULL,
DomainType      ObjectType   NULL,
DomainID        ObjectID      NULL,
Nulls           BOOLEAN      NULL,
DataType        DataType      NULL,
Scale           SMINT        NULL,
Length          INTEGER      NULL,
DistinctValCnt   INTEGER     NULL,
SearchCnt       INTEGER      NULL,
DBID            (Type, ID) DBObjectID,
DBColumnID      (Type, ID, ColumnID) DBColumnID,
DBDomainID      (DomainType, DomainID) DBObjectID
);
```

9. SysColumnComps

```
CREATE TABLE System.SysColumnComps
(
Type            ObjectType   NOT NULL,
ID              ObjectID      NOT NULL,
ColumnID        ColumnID      NOT NULL,
ComponentID     ComponentID  NOT NULL,
ObjectType      ObjectType   NOT NULL,
ObjectID        ObjectID      NOT NULL,
CompColID       ColumnID      NOT NULL,
DBColumnID      (Type, ID, ColumnID) DBColumnID,
DBComponentID   (Type, ID, ColumnID, ComponentID) DBColCompID,
DBCompColID     (ObjectType, ObjectID, CompColID) DBColumnID
```

```
);
```

10. SysKeys

```
CREATE TABLE System.SysKeys
(
Type             ObjectType   NOT NULL,
ID               ObjectID     NOT NULL,
TableType        ObjectType   NOT NULL,
TableID          ObjectID     NOT NULL,
ColumnID         ColumnID     NOT NULL,
KeyType          CHAR(2)      NOT NULL,
ReferenceCnt     SMINT        NULL,
UpdateAction     KeyAction    NULL,
DeleteAction     KeyAction    NULL,
DBID             (Type, ID) DBObjectID,
DBTableID        (TableType, TableID) DBObjectID,
DBColumnID       (TableType, TableID, ColumnID) DBColumnID
);
```

11. SysReferences

```
CREATE TABLE System.SysReferences
(
Type             ObjectType   NOT NULL,
ID               ObjectID     NOT NULL,
ComponentID      ComponentID  NULL,
TableType        ObjectType   NULL,
TableID          ObjectID     NULL,
DBID             (Type, ID) DBObjectID,
DBComponentID    (Type, ID, ComponentID) DBComponentID,
DBTableID        (TableType, TableID) DBObjectID
);
```

12. SysDefaults

```
CREATE TABLE System.SysDefaults
(
Type            ObjectType   NOT NULL,
ID              ObjectID     NOT NULL,
ObjectType      ObjectType   NOT NULL,
ObjectID        ObjectID     NOT NULL,
ColumnID        ColumnID     NULL,
DefaultText     VARCHAR(512) NOT NULL,
DefaultType     CHAR(3)      NOT NULL,
Literal         VARBIN(128)  NULL,
SerialType      ObjectType   NULL,
SerialID        ObjectID     NULL,
DBID            (Type, ID) DBObjectID,
DBObjectID      (ObjectType, ObjectID) DBObjectID,
DBColumnID      (ObjectType, ObjectID, ColumnID) DBColumnID,
DBSerialID      (SerialType, SerialID) DBObjectID
);
```

13. SysRules

```
CREATE TABLE System.SysRules
(
Type            ObjectType   NOT NULL,
ID              ObjectID     NOT NULL,
ObjectType      ObjectType   NOT NULL,
ObjectID        ObjectID     NOT NULL,
RuleText        VARCHAR(1024)NOT NULL,
RuleBinary      VARBIN(1024) NOT NULL,
DBID            (Type, ID) DBObjectID,
DBObjectID      (ObjectType, ObjectID) DBObjectID
);
```

14. SysViews

```
CREATE TABLE System.SysViews
(
Type              ObjectType    NOT NULL,
ID                ObjectID      NOT NULL,
ColumnCnt         SMINT         NOT NULL,
CompColCnt        SMINT         NOT NULL,
Updateable        BOOLEAN       NOT NULL,
UpdateCheck       BOOLEAN       NOT NULL,
ViewText          VARCHAR(12228)NOT NULL,
ViewBinary        VARBIN(16384) NOT NULL,
BuildLastTime     TIMESTAMP     NULL,
BuildFrenquency   INTEGER       NULL,
BuildCnt          INTEGER       NULL,
BuildTime         INTEGER       NULL,
BuildRowCnt       INTEGER       NULL,
UsageCnt          INTEGER       NULL,
DBID              (Type, ID) DBObjectID
);
```

15. SysIndices

```
CREATE TABLE System.SysIndices
(
Type              ObjectType    NOT NULL,
ID                ObjectID      NOT NULL,
ObjectType        ObjectType    NOT NULL,
ObjectID          ObjectID      NOT NULL,
ComponentCnt      SMINT         NOT NULL,
IndexType         CHAR(3)       NOT NULL,
EqUsageCnt        INTEGER       NULL,
EqAvgElements     INTEGER       NULL,
```

```
EqAvgSearchTime   SMFLOAT      NULL,
EqAvgReductionPh  SMFLOAT      NULL,
MiUsageCnt        INTEGER      NULL,
MiAvgElements     INTEGER      NULL,
MiAvgSearchTime   SMFLOAT      NULL,
MiAvgReductionPh  SMFLOAT      NULL,
SiUsageCnt        INTEGER      NULL,
SiAvgElements     INTEGER      NULL,
SiAvgSearchTime   SMFLOAT      NULL,
SiAvgReductionPh  SMFLOAT      NULL,
DBID              (Type, ID) DBObjectID,
DBObjectID        (ObjectType, ObjectID) DBObjectID
);
```

16. SysIndexComps

```
CREATE TABLE System.SysIndexComps
(
Type              ObjectType   NOT NULL,
ID                ObjectID     NOT NULL,
ComponentID       ComponentID  NOT NULL,
TableType         ObjectType   NOT NULL,
TableID           ObjectID     NOT NULL,
ColumnID          ColumnID     NOT NULL,
DBID              (Type, ID) DBObjectID,
DBComponentID     (Type, ID, ComponentID) DBComponentID,
DBTableID         (TableType, TableID) DBObjectID,
DBColumnID        (TableType, TableID, ColumnID) DBColumnID
);
```

17. SysVariables

```
CREATE TABLE System.SysVariables
```

```
(
Type            ObjectType    NOT NULL,
ID              ObjectID      NOT NULL,
VariableText    VARCHAR(2048) NOT NULL,
VariableType    CHAR(3)       NOT NULL,
DataType        DataType      NULL,
Scale           SMINT         NULL,
Length          INTEGER       NULL,
Value           VARBIN(2048)  NULL,
DBID            (Type, ID) DBObjectID
);
```

18. SysPrivileges

```
CREATE TABLE System.SysPrivileges
(
ObjectType      ObjectType    NULL,
ObjectID        ObjectID      NULL,
GrantorID       UserID        NULL,
GranteeID       UserID        NULL,
CanInsert       BOOLEAN       NOT NULL,
CanDelete       BOOLEAN       NOT NULL,
CanSelect       BOOLEAN       NOT NULL,
CanUpdate       BOOLEAN       NOT NULL,
CanReference    BOOLEAN       NOT NULL,
CanExecute      BOOLEAN       NOT NULL,
GrantInsert     BOOLEAN       NOT NULL,
GrantDelete     BOOLEAN       NOT NULL,
GrantSelect     BOOLEAN       NOT NULL,
GrantUpdate     BOOLEAN       NOT NULL,
GrantReference  BOOLEAN       NOT NULL,
GrantExecute    BOOLEAN       NOT NULL,
DBID            (ObjectType, ObjectID) DBObjectID,
```

```
PrivilegeID      (ObjectType, ObjectID, GrantorID, GranteeID)
PrivilegeID
);
```

19. SysColumnPrivs

```
CREATE TABLE System.SysColumnPrivs
(
ObjectType       ObjectType   NULL,
ObjectID         ObjectID     NULL,
ColumnID         ColumnID     NULL,
GrantorID        UserID       NULL,
GranteeID        UserID       NULL,
CanSelect        BOOLEAN      NOT NULL,
CanInsert        BOOLEAN      NOT NULL,
CanUpdate        BOOLEAN      NOT NULL,
GrantSelect      BOOLEAN      NOT NULL,
GrantInsert      BOOLEAN      NOT NULL,
GrantUpdate      BOOLEAN      NOT NULL,
DBColumnID       (ObjectType, ObjectID, ColumnID) DBColumnID,
ColPrivID        (ObjectType, ObjectID, ColumnID, GrantorID,
GranteeID) ColPrivID
);
```

APPENDIX B: ERROR CODES

DATA DEFINITION ERRORS

All data definition and data manipluation errors can be trapped by the standard DAL ERRORCTL statement. Set `ERR0RCTL 1;` in your program if you wish to handle any of these errors in you DAL program.

Where DAL compatible errors are used, the DAL defined macro has been given.

Database related errors

-12040	Unknown database.
-12039	Database already exists.
-12038	Database is currently in use.
-12037	Database cannot be opened, restore/recover pending.
-12036	Maximum number of open databases exceeded.

Database alias related errors

-10211	(CEDBOPEN) There is no database open.
-923	(CENOCON) Database with alias '%s' does not exist.
-12035	Database alias '%s' already in use.

Database objects:

-601	(CEEXISTS) Database object '%s' already exists.
-12034	Unknown database object.
-12033	Table is currently in use.

Database users and groups:

-12032	Unknown user.
-12031	Unknown group.
-12030	A user with this name already exists.
-12029	A group with this name already exists.
-12028	The user is already a member of this group.

DATA MANIPULATION ERRORS

Error that can be specifically trapped (and are of particular interrest) during data manipulation.

-407	(CENONNUL) NULL assigned to NON-NULL column.
-803	(CEDUPROW) nsert or update would create duplicate row in a unique key column.
-10105	(CELOCK) The requested lock cannot be granted as a conflicting lock is already held by another ransaction. This result is only possible if the transaction has a wait time (the time to wai for a lock) of anything less than 'infinity', and is only returned after the specified wait time.

-913	(CETXFAIL) The calling transaction has been rolled back due to deadlock with one other transaction (only) mutual deadlock can be detected). This result is only possible if wait time is set to a value greater than zero.
-12060	Rule violation.

PRIVILEGE VIOLATIONS

Primary errosr

-551	(CEOBJPRV) Object level privilege violation on %s.
-552	(CEOPNPRV) Command level privilege violation.
-922	(CECONAUTH) Database access denied, unknown user.
-12100	Database access denied, invalid password.
-12099	Column level privilege violation.

Secondary errors

-12080	nsert privilege required on %s.
-12079	Select privilege required on %s.
-12078	Delete privilege required on %s.
-12077	Update privilege required on %s.
-12076	Reference privilege required on %s.
-12075	Grant privilege required on %s.
-12074	DBA privileges required.

-12073	DBA privileges or object creator required.
-12072	Resource privileges required.
-1207	SA privileges required.
-12070	Invalid privilege required for %s.

CALCULATION AND CONVERSION ERRORS

Errors that may occur during calculations and conversions.

-413	(CECNVOFL) Size overflow in conversion.
-10026	(CEUFLOW) Conversion underflow.
-10002	(CEDATLIT) Invalid date/time litera.
-103	(CENUMLIT) Invalid numeric literal.

Invalid literal (string) values in conversion

-12140	Invalid boolean literal.
-12139	Invalid decimal literal.
-12138	Invalid real value.
-12137	Date/time value out of range in conversion.
-12136	Invalid conversion of negative value to unsigned.
-12135	Binary value size mismatch in conversion.
-12123	Binary value is not a valid decimal number.

String to floating point conversion errors

-12134	Floating point value is not a number (NAN).
-12133	Floating point value is positive inifinity.
-12132	Floating point value is negitive inifinity.

Invalid conversions

-12131	Conversion between given types is not possible.
-12130	Illegal type in conversion (unknown or unsupported type in conversion). This includes: WORD_4, REAL_10/12 (sometimes), LONG...)

Error in calculations

-12129	Date/time calculation error.
-12128	Date calculation error
-12127	Time calculation error.
-12126	Size overflow.
-12125	Size underflow
-12124	Divide by zero.

TRAPABLE PROGRAMER ERRORS

These errors can occur due to a programmer error, but some programmer may be interrested in handling them

-12160	Cannot open file for execution.
-12154	File not found.
-12153	Bad file handle.

Symbol related errors

-10004	(CEUNDEF) Symbol was not previously declared.
-10016	(CENOVAL) Symbol used before given an initial value.
-12158	A system variable was assigned an illegal value.*/

Cursor related errors

-10020	CENOQRY) The specified cursor cannot be used, it is inactive.
-10202	(CEROWNR) Absolute or relative cursor move not in rowset.
-508	(CECURROW) Current row of cursor is invalid in CURRENT OF reference.
-10021	(CENOFTCH) No fetch done yet (no current row).
-10022	(CEMXFTCH) Fetch beyond last (no current row).
-12156	No such column alias exists for the given cursor
-12155	Cursor column ordinal number out of range.

Connection related errors

-12152	No connection has been opened.

-12151	Multiple connections specified in one statement.

APPENDIX C: GOLFERS DATABASE

In the reference sections of this manual, the golfers database is used to illustrate the various syntax possibilities in each command. A brief description of the tables in this database is given here, followed by the creation of thedatabase in-script form.

DATABASE DESCRIPTION

The golfers database consists of six tables storing information on golfers, golf clubs, golf courses, competitions, results of competitions and scores. The tables are as follows:

Golfers

The golfers table contains one entry for each golfer.

ID	An identity number for each golfer. This number is taken from the domain, GolferID, which has a serial default defined on it. The primary key for this table is defined on this column, as is an index.
SurName	The last name of the golfer.This value is taken from the domain, NameType, which is ordered using a case insensitive system collating sequence.

FirstNames	The first names of each golfer. This value is taken from the domain, NameType, whose values are defined to be case insensitive. This is done by specifying that the domain is to be ordered using the case insensitive system collating sequence.
Name	This is a composite column, which takes its values from the columns, SurName and FirstName. This column is a candidate key for the table, which means the composite values of the column are table-wide unique.
Title	This column stores the title of each golfer: for example, whether the golfer is a Mr., Mrs., Miss, etc., etc.
Gender	The sex of each golfer. A rule has been defined on this column, allowing only the values "M" and "F".
Nationality	The nationality of each golfer. All values in this column are taken from the domain, NameType.
DateOfBirth	The date of birth of each golfer.
Status	The status of each golfer; whether he/she is an amateur, or professional. All values in this column are taken from the domain, StatusType, which has a rule define on it, allowing only the following values: amateur, pro and pro/am. The value pro/a, can be applied to competitions only, and means that the competition includes both amateur and professional golfers.
Handicap	The handicap for each golfer is noted here. All values in this column are taken from the domain, HandicapType. A rule has been defined on this domain, allowing only those values in the range between 36 and -5 (inclusive).
MemberOfClub	This is an identity number for the club in which each golfer plays. A foreign key is defined on this column. All values are taken from the domain, ClubID. The value may be NULL if the player is not a member of any club.

Clubs

The Clubs table contains one entry for each club to which the golfers belong, or at which competitions take place.

ID	An identity number for each club. All values are taken from the domain, ClubID. The primary key for the table is defined on this column, as is an index.
Name	The name of the club. All values are taken from the domain, NameType. A candidate key is defined on this column.
Address	The address of the club. The data type of this column is VARCHAR, with a minimum size of 120 characters.
City	The city in which the club is situated.
Country	The country in which the club is situated.
Founded	The date when the club was founded.
Professional	The ID of the professional golfer who works at the club. A foreign key is defined on this column. The foreign key automatically references the Golfers table over the domain GolferID.
NoOfCourses	The number of golf courses at the club.
NoOfMembers	The total number of members that belong to the club.

Courses

Each club has a number of courses. The details of each hole of each course, are stored in the Courses table.

Club	The ID of the club, to which this course belongs. The foreign key for the table is defined on this column.

Course	The number of the course. All values are taken from the domain, CourseNO, which has a rule defined on it, ensuring that all values are greater than zero. Courses are numbered 1,2,... up to the total number of courses for that particular club.
Hole	The number of the hole on the course. All values are taken from the domain, HoleNO. A rule is defined on this domain, allowing only number in the range 1 to 18.
Key	A composite column, that takes its values from the simple columns, Club, Course, and Hole, in that order. The primary key for the table is defined on this composite column, as is an index.
Distance	The total distance of the hole in meters.
Par	The average number of strokes it takes a professional player to complete the hole. A rule is defined on this column, allowing only values 3, 4, and 5. A default has also been defined on this column, that automatically sets the par for a hole at 4.
Stroke	This is the difficulty rating for each hole. This rating is a value between 1 and 18, (inclusive). All values are taken from the domain, HoleNo, which has a rule enforcing this restriction.
Description	A description of the hole. The data type of this column is VARCHAR, and the description may not be more than 120 characters long.

Competitions

The Competitions table contains information on all competitions played by the golfers.

ID	An identity number for each competition. All values are taken from the domain, CompetitionID. The primary key for the table is defined on this column, as is the index.
Name	The name of the competition. All values are taken from the domain, NameType. A candidate key for the table is defined on this column.
Status	The status of the competition; whether it is amateur, professional, or whether amateurs and professionals play together. All values are taken from the domain, StatusType, which has a rule defined on it, allowing only the above values.
Month	The month in which the competition takes place. The data type of this column is SMINT. It is not DATE, as this is a record of an annual competition, therefore there is no need to state a year.
Day	The day on which the competition takes place. The data type of this column is SMINT.
Club	The ID of the club where the competition takes place. All values are taken from the domain, ClubID. A foreign key is defined on this column, that automatically references the Clubs table of the domain, ClubID.
Course	The number of the course where the competition takes place.All values are taken from the domain, CourseNO.
NoOfRounds	The number of rounds in the competition. This column has a default defined on it, setting the value automatically at 4. All values are taken from the domain, RoundNO.
Scoring	This column records the method of scoring at each competition. A rule is defined on this column that allows the scoring values, "Medal" (an absolute score count), "Stableford" (a point system), "Skins" (a game in which golfers play for money on each hole), and "Plus-Minus" (another type of point scoring system)

Results

Each competition has a number of results. The details are stored in the Results table.

Year	The year the competition took place. The data type of this column is SMINT.
Competition	The ID of the competition in question. A foreign key is defined on this column referring to the Competitions table.
Placing	The placing (1st, 2nd, 3rd...) obtained by a golfer who played in the competition.
Key	This is a composite column, which takes its values from the simple column, Year, Competition and Place. The primary key for the table is defined on this column, as is the index.
Golfer	The ID of the golfer who played in the competition. A foreign key is defined on this column.
TotalScore	The sum of the golfer´s score for each round of the competition.
Points	The value here depends on the type of competition. For example, in a Medal competition the value is the number of strokes above (a positive value) or below (a negative value) par at the ned of the competition. In a Stableford competition, this field contains the total number of Stableford points obtained.
Winnings	How much the golfer won in the competition. The data type of this column is MONEY[12,2].

Scores

The Scores table contains the score for each hole obtained by each player in each competition.

Year	The year in which the player took part in the competition.
Competition	The ID for the competition. All values are taken from the domain, CompetitionID. A foreign key is defined on this column, that references the Competitions table.
Golfer	The ID for the golfer who took part in the competition. All values are taken from the domain, GolfersID. A foreign key for the table is defined on this column.
Round	The number of the round, in which this hole was played. All values are taken from the domain, RoundNO.
Hole	The number of the hole played.
Key	This is a composite column, which takes its values from the simple columns, Year, Competition, Golfer, Round, and Hole. The primary key for the table is defined on this column, as is the index.
Score	The number of strokes it took the golfer to play the hole. There is a rule defined on this column, that states that all values must be greater than zero.
HoledOut	This is a column of BOOLEAN type. If FALSE, this means that the player did not complete the hole. In a medal competition this would disqualify the player, but in other types of Competitions, this would only mean the player obtains the worst possible score for the hole.
Points	The number of points achieved at the hole. The value here depends on the coring system used in the competition.

CREATE SCRIPT

This script creates the golfers database:

```
/* GOLFERS: */

CREATE DATABASE "Golfers";

OPEN   DATABASE "Golfers";

CREATE COUNTER INTEGER GolferCnt = 1;

CREATE COUNTER INTEGER ClubCnt = 1;

CREATE COUNTER INTEGER CompetitionCnt = 1;

CREATE DOMAIN GolferID INTEGER NOT NULL;

CREATE DEFAULT GolferDef ON DOMAIN GolferID AS SERIAL GolferCnt;

CREATE DOMAIN CompetitionID INTEGER NOT NULL;

CREATE DEFAULT CompetitionDef ON DOMAIN CompetitionID AS SERIAL
CompetitionCnt;

CREATE DOMAIN ClubID INTEGER;

CREATE DEFAULT ClubDef ON DOMAIN ClubID AS SERIAL ClubCnt;

CREATE DOMAIN NameType VARCHAR[55] AS CASE INSENSITIVE;

CREATE DOMAIN StatusType CHAR[8];

CREATE RULE StatusRule ON StatusType AS StatusType IN ( 'Ama-
teur', 'Pro', 'Pro/Am' );

CREATE DOMAIN CourseNO SMINT;

CREATE RULE CourseRule ON CourseNO AS CourseNO > 0;

/* Handicaps are stored as value that is subtracted from the
final score: */

CREATE DOMAIN HandicapType SMINT;
```

```
CREATE RULE HandicapRule ON HandicapType AS HandicapType BETWEEN
36 AND -5;

CREATE DOMAIN RoundNO SMINT;

CREATE RULE RoundRule ON RoundNO AS RoundNO BETWEEN 1 AND 4;

CREATE DOMAIN HoleNO SMINT;

CREATE RULE HoleRule ON HoleNO AS HoleNO BETWEEN 1 AND 18;

/* The golfers table contains one entry for each golfer.

CREATE TABLE Golfers
(
ID GolferID NOT NULL,
SurName NameType NOT NULL,
FirstNames NameType NOT NULL,
Name (SurName, FirstNames),
Title CHAR[10],
Gender CHAR[1] NOT NULL,
Nationality NameType,
DateOfBirth DATE,
Status StatusType,
Handicap HandicapType,
MemberOfClub ClubID,
Earnings MONEY[12,2]
);

CREATE PRIMARY KEY GolfersPk ON Golfers.ID;

CREATE CANDIDATE KEY GolferNameCk ON Golfers.Name;

CREATE FOREIGN KEY GolferClubFk ON Golfers.MemberOfClub;

CREATE INDEX GolfersIndex ON Golfers (ID);

CREATE RULE GenderRule ON Golfers AS Gender IN ( 'M', 'F' );

/* The Clubs table contains one entry for each club to which
golfers belong, or at which competitions take place.
```

```
CREATE TABLE Clubs
(
ID ClubID NOT NULL,
Name NameType NOT NULL,
Address VARCHAR[120],
City NameType,
Country NameType,
Founded DATE,
Professional GolferID,
NoOfCourses SMINT NOT NULL,
NoOfMembers INTEGER NOT NULL
);

CREATE PRIMARY KEY ClubsPk ON Clubs.ID;

CREATE CANDIDATE KEY ClubNameCk ON Clubs.Name;

CREATE FOREIGN KEY ClubProFk ON Clubs.Professional ON UPDATE
CASCADE ON DELETE CASCADE;

CREATE INDEX ClubsIndex ON Clubs (ID);

/* Each club has a number of courses. The details of each hole
of each course, are stored in the Courses table */

CREATE TABLE Courses
(
Club ClubID NOT NULL,
Course CourseNO NOT NULL,
Hole HoleNO NOT NULL,
Key ( Club, Course, Hole ),
Distance SMINT,
Par SMINT,
Stroke HoleNO,
Description VARCHAR[120]
);

CREATE PRIMARY KEY CoursesPk ON Courses.Key;

CREATE FOREIGN KEY CourseClubFk ON Courses.Club;
```

```
CREATE INDEX CoursesIndex ON Courses (Club, Course, Hole);

CREATE RULE ParRule Courses AS Par IN ( 3, 4, 5 );

CREATE DEFAULT ParDef ON Courses.Par AS 4;

/* The Competitions table contains information of all competi-
tions played by the golfers.*/

CREATE TABLE Competitions
(
ID CompetitionID NOT NULL,
Name NameType NOT NULL,
Status StatusType NOT NULL,
Month SMINT,
Day SMINT,
Club ClubID,
Course CourseNO,
NoOfRounds RoundNO,
Scoring CHAR[20]
);

CREATE PRIMARY KEY CompetitionsPk ON Competitions.ID;

CREATE CANDIDATE KEY CompetitionNameCk ON Competitions.Name;

CREATE FOREIGN KEY CompetitionClubFk ON Competitions.Club;

CREATE INDEX CompetitionsIndex ON Competitions (ID);

CREATE DEFAULT NoOfRoundsDef ON Competitions.NoOfRounds AS 4;

CREATE RULE ScoringRule ON Competitions AS Scoring IN

( 'Medal', 'Stableford', 'Skins', 'Plus-Minus' );

/* Each competition has a number of results. The details are
stored in the Results table. */

CREATE TABLE Results
(
Year SMINT NOT NULL,
```

```
Competition CompetitionID NOT NULL,
Placing SMINT NOT NULL,
Key ( Year, Competition, Placing ),
Golfer GolferID,
TotalScore SMINT,
Points SMINT,
Winnings MONEY[10,2]
);

CREATE PRIMARY KEY ResultsPk ON Results.Key;

CREATE FOREIGN KEY ResultGolferFk ON Results.Golfer;

CREATE INDEX ResultsIndex ON Results (Year, Competition, Plac-
ing);

/* The Scores table contains the score for each hole obtained by
each player in each competition. */

CREATE TABLE Scores
(
Year SMINT NOT NULL,
Competition CompetitionID NOT NULL,
Golfer GolferID NOT NULL,
Round RoundNO NOT NULL,
Hole HoleNO NOT NULL,
Key ( Year, Competition, Golfer, Round, Hole ),
Score SMINT,
HoledOut BOOLEAN,
Points SMINT
);

CREATE PRIMARY KEY ScoresPk ON Scores.Key;

CREATE FOREIGN KEY ScoreGolferFk ON Scores.Golfer;

CREATE INDEX ScoresIndex ON Scores (Year, Competition, Golfer,
Round, Hole);

CREATE RULE ScoreRule ON Scores AS Score > 0;
```

```
/*The View, GolfersAmateurs, includes all those golfers who have
the Status, "Amateur"*/

CREATE VIEW GolfersAmateurs
AS SELECT ID, Name, Title, Handicap, Status
FROM Golfers
WHERE Status = "Amateur" WITH CHECK OPTION;
```

INDEX

C

D

M

N

O

P

Q

R