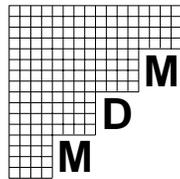


# dbView

*A Database Access Tool*

The Data Management Section  
Multimission Image Processing System  
of  
The Jet Propulsion Laboratory



John Rector, CDE  
Jeffery Jacobson  
Marc Sarrel  
Jimmie Young  
Leo Bynum

Version 1.4  
November 28, 1994



# *Table Of Contents*

## **Table Of Contents iii**

## **Licensing ix**

## **Preface 11**

1 **Where To Get Copies Of This Document 11**

2 **Typographic Conventions 11**

3 **Syntax Rules 12**

3.1 **Command Keys 12**

3.2 **Command Syntax 12**

4 **Default Values For Prompts 13**

## **Release Notes & Installation Guide 15**

5 **Release Notes 15**

5.1 **What's New In Version 2.0 15**

6 **Installing dbView 16**

6.1 **Setting Up A Sybase Environment 16**

## **Tutorial 19**

7 **Introduction 19**

7.1 **A Little More About dbView 20**

8 **Getting Started 21**

8.1 **Sybase System Stored Procedures 23**

8.2	Ending The dbView Session	25
8.3	What You've Learned	26
9	<b>Running And Exiting From dbView</b>	<b>27</b>
10	<b>Connecting To A Database Server</b>	<b>28</b>
10.1	The Connect Retry Prompt	28
10.1.1	Increasing The Time-out Period	29
10.1.2	The Show Server Command	29
10.1.3	Can't Find The Interfaces File	29
10.1.4	Getting The List Of Databases And Accessing A Database	30
10.1.5	Exiting After Connection Failure	31
10.1.6	The Connect command	31
10.1.7	Using A Default Database	32
11	<b>Connecting To Multiple DBMS's Simultaneously</b>	<b>33</b>
11.1	Defining Multiple Connections	33
11.2	Using Connection Handles	34
11.3	Handles And The Password Server	37
11.4	Handles And dbView's Batch Mode	38
12	<b>Executing A Command</b>	<b>39</b>
12.1	Cancelling A Command	40
12.1.1	Cancelling A Command Using <control-c>	40
12.1.2	Cancelling A Database Command Using <control-c>	40
12.1.3	The reset Command	40
12.2	Editing Commands	41
12.3	When dbView Encounters An Error In An SQL Command	42
12.4	The Set Command	42
12.4.1	Set Command Definitions	43
12.5	Saving dbView's Environment	46
12.6	Timing Commands	46
12.7	Altering The Database Table Display Format	48
12.7.1	Table Format	48
12.7.2	List Format	49
12.7.3	Export Format	50
12.7.4	Copy And Pasting Exported Data	52
12.8	Displaying Table Header And Footer Information	53
12.9	The Page command	53

---

12.10	Real Number Formats	<b>54</b>
12.10.1	The f Format	<b>54</b>
12.10.2	The E or e Format	<b>56</b>
12.10.3	The g or G Format	<b>57</b>
12.10.4	The Feedback Command	<b>58</b>
12.10.5	Using Sybase's "Set" Command In dbView	<b>58</b>
13	<b>Escaping To The Operating System</b>	<b>60</b>
14	<b>The Help Command</b>	<b>61</b>
14.1	The History List	<b>63</b>
14.1.1	Setting The Length Of The History List	<b>64</b>
14.2	The last Command	<b>64</b>
15	<b>Saving Commands And Data</b>	<b>66</b>
15.1	The Show File Command	<b>67</b>
15.2	The Directory Command	<b>67</b>
16	<b>Macro Commands</b>	<b>72</b>
16.1	Defining And Running A Macro	<b>72</b>
16.1.1	The Show Macro Command	<b>74</b>
16.1.2	Editing A Macro	<b>75</b>
16.1.3	Using Edit Macro To Create Another Macro	<b>76</b>
16.2	Using The History List In A Macro Command	<b>77</b>
16.3	Using Macros To Redefine dbView Commands	<b>78</b>
16.4	Using Variables In Macro Definitions	<b>81</b>
16.4.1	Local Variables And Default Values	<b>82</b>
16.4.2	The Special Local Variable \$password	<b>84</b>
16.4.3	Summarizing Macro Local Variable Rules	<b>85</b>
16.5	Repeated Execution Of Macros	<b>86</b>
16.6	Including, Saving, And Replacing Macro Definitions	<b>87</b>
16.6.1	Saving Macros To A File	<b>87</b>
16.6.2	Removing A Macro Command	<b>87</b>
16.6.3	Exiting From dbView Once You've Made Changes To Macros	<b>88</b>
16.6.4	Including A Macro File	<b>89</b>
16.6.5	The Default Macro File	<b>89</b>
16.6.6	Replacing A Set Of Macros	<b>90</b>
16.6.7	Sharing Macro Files	<b>91</b>

17	<b>Global Variables 92</b>
17.1	The global Command 92
17.1.1	Seeing Global Variable Assignments 93
17.1.2	The Expand Global Command 93
17.2	Referencing Global Variables In Macros 93
17.2.1	The Expand Macro Command 94
17.3	The Remove Global Command 95
17.4	Undefined Global Variables In Macros 95
17.5	Global Variables And Macro Files 96
18	<b>Finding Out About Database Objects 97</b>
18.1	Sybase Database Objects 97
18.1.1	Table Information 98
18.1.2	View Information 100
18.1.3	Stored Procedure Information 102
18.1.4	Trigger, Default And Rule Information 104
18.2	Illustra Database Objects 104
19	<b>Defining and Running Reports 105</b>
19.1	Sample Reports 105
19.1.1	The Report mission.rpt 105
19.1.2	The Report planets.rpt 110
19.2	Summarizing dbView's Report Writing Capabilities 115
19.2.1	Report Functions 115
19.2.2	Formatting Character Strings 116
19.2.3	Formatting Numbers 116
19.2.4	Special Variables Used In Report Footers 117
19.2.5	Options For Report Printing and Mailing 117
19.2.6	More About The Print Section 118
19.2.7	Cancelling A Report Specification Command 118
19.2.8	Error You May Encounter When Using Report Commands 118
19.3	Using The History List For Report Generation 120
19.4	Hints For Creating Reports 120
20	<b>The Script Command 122</b>
20.1	Some Characteristics Of Scripts 123
20.1.1	Scripts Can Be Nested 123
20.1.2	Putting Comments In A Script File 123

20.1.3	Pausing In A Running Script	123
20.1.4	Rules To Remember When Running Scripts	124
20.2	Example Script Files	125
20.2.1	Loading SQL commands	125
20.2.2	Generating And Printing A Report	126
20.2.3	Copying A Database Tables Contents	127
21	<b>dbView's Batch Mode</b>	<b>135</b>
22	<b>Error Messages</b>	<b>136</b>
22.1	What's In An Error Message?	136
22.1.1	The Banner Line	136
22.1.2	The Sybase Error Number	137
22.1.3	The Message	137
22.2	Some Common Login Errors	138
22.2.1	Incorrect User Name Or Password	138
22.2.2	Incorrect Server Name Or Server Name Not In Interfaces File	138
22.2.3	Incorrect Database Name	139
22.2.4	Server Is Not Running	139
22.2.5	Can't Reach Machine Named In Interfaces File	140
	<b>Database Bibliography</b>	<b>141</b>
23	<b>General Relational Database References</b>	<b>141</b>
24	<b>The SQL Language</b>	<b>141</b>
25	<b>Books About Sybase</b>	<b>142</b>
26	<b>Sybase Manuals</b>	<b>143</b>
	<b>Index</b>	<b>155</b>



# Licensing

## END-USER LICENSE AGREEMENT

This End-User License Agreement (this “Agreement”) is by and between The Jet Propulsion Laboratory, a part of the California Institute Of Technology (“The Jet Propulsion Laboratory”) and the end-user of the Software contained in the accompanying package (“you” or “your”).

### RECITALS

- A. The Jet Propulsion Laboratory is the developer of and the owner of world-wide rights to the “Software” as defined hereinbelow.
- B. “Software” shall mean the machine readable software program *dbView* and associated files in the accompanying package, and any modified version, upgrades and other copies of such programs and files.
- C. You desire to obtain a non-exclusive license to use the Software and The Jet Propulsion Laboratory desires to grant you said license, subject to the terms and conditions set forth in the Agreement.
- D. The Software includes the Open Client/C library from Sybase Corporation. The Software can not be run on any machine without first obtaining a valid license for the Open Client/C library. The number of users who are allowed to use the Software is limited by the number of users covered by the Open Client/C license.

NOW, THEREFORE, in consideration of the premises and the mutual covenants herein contained, the parties hereto agree as follows:

#### 1. Grant of Non-Exclusive License.

Subject to the terms and conditions of this Agreement, The Jet Propulsion Laboratory hereby grants to you a non-exclusive license to use the Software and you hereby accept said non-exclusive license. You hereby acknowledge that the licensor and the party-in-interest in this Agreement is The Jet Propulsion Laboratory.

#### 2. Scope of Use.

The Software may be operated on multi-user or networked systems, subject to any restrictions that result from Sybase Corporation licensing agreements.

#### 3. Proprietary Rights.

You acknowledge that the program code, structure and organization of the Software is the confidential copyrighted property of The Jet Propulsion Laboratory and is a valuable trade secret of The Jet Propulsion Laboratory and is licensed to you on a non-exclusive basis. You agree to hold such trade secrets in

confidence. You further agree not to translate, disassemble or reverse engineer the Software, in whole or in part, or to use the documentation ("Documentation") for any purpose other than to support your use of the Software.

**4. No Other Rights.**

The Jet Propulsion Laboratory retains title and ownership of the Software and the Documentation on all diskette copies and all subsequent copies of the Software, regardless of the form or media in or on which the original and other copies may exist. Except as stated above, this Agreement does not grant you any rights to patents, copyrights, trade secrets, trademarks or any other right in respect of the Software or the Documentation.

**5. Term.**

The license is effective until terminated. The Jet Propulsion Laboratory has the right to terminate your license immediately if you fail to comply with any term or condition of this Agreement. Upon any such termination, you must destroy the original and any copies of the Software and its documentation.

**6. Limitation of Liability.**

IN NO EVENT WILL The Jet Propulsion Laboratory BE LIABLE TO YOU FOR ANY CONSEQUENTIAL OR INCIDENTAL DAMAGES, INCLUDING BUT NOT LIMITED TO ANY LOST PROFITS, LOST DATA, LOST TIME, OR OTHER LOSSES, EVEN IF AN The Jet Propulsion Laboratory REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY PARTY.

**7. Choice of Law.**

This Agreement will be governed by and construed and enforced in accordance with the laws in force in the State of California.

**8. Integration.**

You acknowledge that you have read this Agreement, understand it and that it is the complete and exclusive statement of your agreement with The Jet Propulsion Laboratory which supersedes any prior agreement, oral or written, between The Jet Propulsion Laboratory and you for this product. No variation of the terms of this Agreement will be enforceable against The Jet Propulsion Laboratory unless The Jet Propulsion Laboratory gives its express consent, in writing signed by an officer of The Jet Propulsion Laboratory.

**9. Consent to Terms of Agreement.**

You agree that any use by you of the Software constitutes your consent to be bound by the terms and conditions of this Agreement.

The Jet Propulsion Laboratory  
California Institute Of Technology  
4800 Oak Grove Drive  
Pasadena, California 91109-8099  
USA

# Preface

## 1 Where To Get Copies Of This Document

A Postscript version of this document, named `dbView.ps`, is located on the MDMS WWW server. Use the following URL to locate it:

`http://www-mipl/mdms/MDMS.html`

## 2 Typographic Conventions

In this document, we use different typographic styles to signify how a word is being used.

Database and file names appear in normal Courier:

The `catalog` database...

The file `/usr/franklin/dbView...`

Database object names, tables, views, stored procedures, etc., as well as the names of fields within tables, are in italic type:

The examples in this document use the tables *missions* and *planets*.

The *missions* table contains the fields, *mission*, *scId*, *objective*,...

The names of program commands and their formal syntax specification are italicized:

To display the time of execution of an SQL statement, use the *set timer* command. The syntax for the command is:

*set timer { on | off }*

Values supplied to a program are placed between quotation marks when they appear in the body of the document:

The database server user name in our example is "franklin"...

Terminal, or screen, I/O examples appear in Courier type. Responses to prompts that are supplied by a user appear in Courier bold type:

To login to a database, use the *connect* command. At each prompt, you must supply a connection parameter, like so:

```
1> connect
  userName []: franklin
  password:
  server []: CATALOGDBS
  database []: catalog
```

## 3 Syntax Rules

### 3.1 Command Keys

Command keys appear in angle brackets. For example, when you are asked to press the return key on your keyboard, the text looks like this:

Press the <return> key.

Also, combinations of command keys appear in angle brackets and are separated by a dash. For example:

To cancel a command, press <control-c>

This command asks you to press the *control* and the *c* key simultaneously.

### 3.2 Command Syntax

- An options list, where one option must be selected appear between curly braces. A vertical bar separates options and stands for the word “or”. For example, the command:

```
set timer {on | off}
```

means that the *set timer* command takes one of two options: either “on” or “off”, so the valid commands are:

```
set timer on
set timer off
```

- Names that signify a value you must supply appear in angle brackets:

```
set page <number of lines>
```

In this example, you are expected to supply a number representing the number of lines to appear on a viewing page; for example:

```
set page 10
```

- Optional parameters appear in square brackets, “[ ]”. For example:

```
show set [<command name>]
```

signifies that the *show set* command can be used with, or without, a parameter. For example:

```
show set
```

```
show set timer
```

## 4 Default Values For Prompts

If a command issues a prompt, it will very often have a default value associated with it. You accept the default value by pressing the <return> key. Default values appear in square brackets following the prompt. The *connect* command that prompts you for database login information is an example of a command that uses default values:

```
1> connect  
userName [franklin]: <return>  
password:  
server [PDSSERVER]: CATALOGDBS  
database [master]: catalog
```

The command has four prompts. In the example, we accepted the *userName* default, “franklin”, by pressing the <return> key. There is no default—no values in square brackets—for the password. The default values for the last two prompts were not used; new values were supplied instead.



# Release Notes & Installation Guide

## 5 Release Notes

dbView is a command line interface to Sybase database servers. If you have bugs to report or suggestions to make that will improve the product or its documentation, please send an electronic mail message to:

`mdms@candide.jpl.nasa.gov`

Bug reports should include a description of the bug, including a simple example that illustrates the bug. Also include any error messages that were returned by the program at the time that the bug occurred.

All correspondence — bug reports or suggestions — should include information that will allow us to contact you. Include your name, organization and either your electronic mail address or telephone number.

For further information about this product, contact the MIPS Data Management Cognitive Design Engineer:

John Rector  
Jet Propulsion Laboratory  
MS 168-522  
4800 Oak Grove Driver  
Pasadena, CA 91109-8099

Email: `jar@next1.jpl.nasa.gov`

### 5.1 What's New In Version 2.0

- dbView macros have been changed so that you can include more than one SQL statement.
- Comments now appear in the body of the macro at any point you want to see them. Blank lines are also supported. These additional features caused us to change the syntax of macros. Version 1.n macros won't run with dbView version 2.
- Macro commands have a print statement that displays strings when a macro is run.
- The way in which individual macros can be saved to a file has been improved and

extended.

- A new set command, set feedback on or off has been added. When turned on, dbView will issue a string of dots on the command line while waiting for SQL statement to execute in the server.
- The server command will list the Sybase servers known to dbView.
- The few known bugs in dbView have been fixed.
- Sections of this document have been reordered to make the presentation of material more logically.

## 6 Installing dbView

dbView is a program written in ANSI Standard C and is in compliance with the POSIX.1 library interface protocol. Normally, you will receive an executable version of the program built to run on your particular machine. In this case, you don't need to build the program, so installation is just a matter of copying the program to a directory from which it can be run. Ask your system administrator for directions about where executables are located on your machines or network.

dbView connects with Sybase servers using Sybase's Open Client/C Library. Before you run dbView, you must have a run-time license from Sybase for this library. *It is illegal to run dbView without this license.* If you are using a terminal window to access another machine on which dbView is located, you don't have to have a license on your machine, but the machine you're accessing does need a license.

Before you can use dbView to connect to a Sybase server, you'll need to configure your machine to include a Sybase client environment. dbView is not unique in this requirement; you can't run any client process connecting to a Sybase server without a Sybase environment.

### 6.1 Setting Up A Sybase Environment

When you run dbView, it first attempts to connect to a Sybase database server. For this to be successful, you must set-up the environment necessary to make the connection. If you're lucky, somebody has already done this for you, but if that's not the case, here is what you must do.

The environment variable (Unix) or logical name (VMS) SYBASE must be defined as the directory where the Sybase `interfaces` file is located, and this file must be readable by you. The `interfaces` file contains network information that dbView needs to make a connection to a database server. The file is like a telephone book. It contains server names and connection information just as a telephone book contains names and telephone numbers.

You can check to see if everything is in order with one simple operation: try to read the `interfaces` file in an editor using the environmental variable (logical name) \$SYBASE. In

our example, we use the Unix editor *vi* and include `$$SYBASE` as part of the full file name:

```
% vi $$SYBASE/interfaces
```

If the file does not appear in the editor, you have a problem that must be fixed. The problem could be:

1. The `interfaces` file does not exist.
2. The variable `SYBASE` is not defined or does not point to the directory where the `interfaces` file can be found.
3. You don't have read access to the `interface` file or its directory.

Once you can view the `interfaces` file, you should take some time to examine its contents because it contains the names of all of the servers you can potentially connect to. In the following example `interfaces` file, the server names have been marked in bold type; they are: `MIPSDB1`, `CATALOGDBS` and `PDSYBASECN`. If you were using this `interfaces` file, you would use one of these names when `dbView` prompted you for a server name.

```
#
# File: $$SYBASE/interfaces
#
# Function: The interfaces file used by Data Management
# Development.
#
# Date: June 19, 1992
#
MIPSDB1
    query tcp sun-ether milano 1040
#
CATALOGDBS
    query tcp sun-ether mantua 1040
    query tcp sun-ether venice 1040
#
# PDS Central Node
#
PDSYBASECN
    query tcp sun-ether thorndyke 2030
```

(Note: You should never edit the `interfaces` file; leave that task to a database administrator. The syntax of the file is quite particular; and, if you change it, it may not work properly.)

The indented lines following server name contain the connection information — this is the part of the `interfaces` file that is equivalent to a telephone number. The connection information for a server may change with time; but, as long as the server names remain the same, you should be able to make your connection. There is only one case when you'll need to look at the connection line. We'll cover that now.

Using the telephone example again, suppose that you know you've got the right name and number, but you still can't make a connection. It may be the lines are down or the telephone you're trying to reach is out of order. This type of error can happen with computers, too. To test this, you need the *node name* of the computer you're trying to reach. That information is in the `interfaces` file. Look at the name of the server you're trying to reach. Below that you find an indented line that begins with "query...". (If you don't find that line; that's the problem; and you'll have to speak to your data administrator to get it fixed.) Just before the number that ends the indented line, you'll see a name, in the example below, it "milano". That's the name of the computer where the server is located.

```
#
MIPSDB1
    query tcp sun-ether milano 1040
```

We want to use that name to see if we can connect to the other machine. This has nothing to do with database servers, we just want to see if our machine can reach the server machine. To check the connection, we'll use the network utility *telnet*. In your terminal window, type "telnet milano":

```
% telnet milano
Trying 128.147.24.63...
Connected to milano.
Escape character is '^]'.

SunOS UNIX (milano)

login:
```

You should see the type of response in the example above. (Type <control-d> at this point to exit from telnet.) If you don't get this sort of response, you're unable to reach the server's machine. At this point, you should contact your system administrator who will configure your machine properly. Once that is done, try to connect to the server again using dbView.

# Tutorial

## 7 Introduction

dbView is a command line utility that allows you connect to a database server—currently a Sybase database server—over a network and to execute database statements and retrieve data. It can be used from a terminal, like a DEC VT100, or from a terminal emulation window. If you're in a windowing environment, dbView can be integrated with other applications to some extent. For example, you can cut data from the dbView window and paste it into another application.

This tutorial introduces dbView's commands using simple examples. We begin with a quick introduction that covers the topics:

1. Starting dbView
2. Connecting to a database server
3. Executing SQL commands and retrieving data from a database
4. Editing a SQL command
5. Exiting from dbView.

The introduction should take you about 15 minutes to complete. Once you've finished it, you should be able to use dbView to accomplish useful working, providing you know the database access language, SQL. (If you don't know SQL, look at the bibliography at the end of this document for some suggested reading.)

Following the introduction, we describe dbView's command set. The commands are grouped into the major topics:

- Connecting to a database server.
- Executing commands.
- Getting help for dbView commands.
- Saving commands and data to files.
- Using macros to encapsulate commands

- Accessing data dictionary information.
- Generating reports.
- Running script files that contain dbView and SQL commands.
- Reading error messages returned by dbView.

Once you've completed the tutorial, you can use the index to quickly locate references to specific commands. They are all listed under the major topic "commands".

## 7.1 A Little More About dbView

Before we begin in earnest, there are just a few more things you need to know about dbView.

- dbView is a *client* process. It accesses a database *server* over a network. This means you can use dbView from anywhere on a network as long as you've got the ability to make a connection to the server in which you're interested. (Of course, you must also have access privileges to the database. If you don't, contact your Database Administrator.)
- Currently, dbView supports Sybase and Illustra database servers.
- dbView currently runs in SUN (SUN OS and Solaris), HP, SGI, NeXT and DEC VMS environments.
- Currently, dbView only supports English language command input.

## 8 Getting Started

In this section we'll show you how to use dbView to execute and edit a database statement. In later sections we'll discuss the additional features of dbView; but for now we just want to access a database, do some useful work and exit the program. When you've completed this section, you'll know the basics of database retrieval using dbView.

To run dbView, type *dbView* at your system prompt and hit the <return> key. (In this guide we'll use a percent sign “%” as the system prompt. The commands you're required to type are displayed in **bold** type.)

```
% dbView
```

When you start dbView, it responds by displaying version and copyright information. Following immediately, it prompts you for four items: your database server login name, password, the name of the server you want to connect to and the name of the database you want to enter once the connection is made. Your screen will look something like this—the items in bold type are your responses:

```
dbView, version 1.4, (dblib, milib), 28 Nov, 1994
Copyright 1993, The Jet Propulsion Laboratory. All rights
reserved.
```

```
userName []: franklin
password:
server []: CATALOGDBS
database []: catalog
DBMS Type []: Sybase
```

```
1> ← this is the command prompt
```

If you're using Illustra, the last line should be:

```
DBMS Type []: Illustra
```

We've used the login name of “franklin”. The password is not displayed for security reasons. The name of the database server is “CATALOGDBS” and the database is “catalog”. Since dbView supports multiple DBMS', you also need to include the type of DBMS. Once we've given dbView this information, it connects us to the database and displays its command prompt—the line with the arrow pointing to it in the example above. If you see something different than this, like one or more error messages, refer to section 22 where frequently encountered errors are discussed.

Notice the square brackets, [], that appear at the end of each prompt except for “password” in the example login. dbView saves your current input values and displays them in these square brackets the next time you login. These are your default values for the prompts. You can accept a default value by simply typing <return> following the prompt. When you type in a new value, it becomes the default for the next session.

Once you're in dbView, type in a SQL command or the name of a Sybase stored procedure. To get another line to continue a command, press the <return> key. Once you've entered your command, press the <return> key once more and type the word "go" on a line by itself. "go" is dbView terminator command. It signifies that your SQL command is complete and should be executed. To execute the command, type <return> following "go"<sup>1</sup>.

For our first example, we'll type in a SQL command that returns the names of all the planets in the Solar System.<sup>2</sup>

Here's the Sybase version:

```
1> select name from planets
```

```
2> go
```

```
name
```

```
-----
```

```
Earth
```

```
Jupiter
```

```
Mars
```

```
Mercury
```

```
Neptune
```

```
Pluto
```

```
Saturn
```

```
Uranus
```

```
Venus
```

```
(9 row(s) affected)
```

For Illustra you need to end the SQL command with a semicolon, like this:

```
1> select name from planets; ← semicolon goes here
```

```
2> go
```

```
name
```

```
----
```

```
Earth
```

```
Jupiter
```

```
Mars
```

```
Mercury
```

```
Neptune
```

```
Pluto
```

```
Saturn
```

- 
- 1 Some DBMS's, Illustra for example, require that you terminate an SQL statement with a semicolon. In that case include the semicolon as part of the SQL statement and then type "go". The examples don't use the semicolon convention, so you should remember to add it if your DBMS requires it.
  - 2 The example database objects used in the document can be found in the script file `examples.sybase` or `examples.illustra`. To load them, connect to the target database and issue the command `script <path>/examples.sybase` or `<path>examples.illustra`.

```
Uranus
```

```
Venus
```

```
(9 row(s) affected)
```

So, what happened? First we typed in four lines that make up the SQL command. Then we typed in “go” on a line of its own and pressed <return> one more time. dbView then executed the command. The field *names* in the SELECT statement appears as the title of a column. The values for the *name* field follow, separated from the field name by a row of dashes. Following the data, dbView displays status information. It says that 2 rows were affected, which means that 2 rows were returned. The word “affected” is used because sometimes we may be altering the database. In that case we don’t get any rows returned, but we still “affect” some of them.

Once dbView displays the status information, it returns us to the command line. It prompts us with a “1>” again, indicating that it’s ready to receive a new command.

If you make a mistake while typing in the command, try one of these options:

1. Cancel the command and start over by typing <control c>. Then type in the command again.

```
1> select anme
2> from ^C ← typed <control c> here
1> select name
2> from planets
5> go
```

2. Edit the command by typing the word *edit* on a command line of its own. Edit the command and return to dbView. Type “go” to execute the command once you’ve returned from the editor. (Note: By default, the *vi* editor is used on Unix systems and *edt* on VMS systems.)

```
1> select naem
2> from
3> edit
```

...correct and complete the command in your editor and then return to dbView’s command line.

```
1> select name
2> from planets
5>
```

...execute the command by typing “go” on line 5

## 8.1 Sybase System Stored Procedures

Next, we’ll execute the system stored procedure, *sp\_who*. This procedure gives us the information the database server has about our connection. (If you’re not familiar with stored proce-

dures, refer to one of the Sybase references in the bibliography.) For our example, we follow the stored procedure name with our login name, “franklin”. You should use your own login name if you repeat the example:

```
1> sp_who franklin
2> go

 spid  status    loginame  hostname  blk  dbname  cmd
-----  -
 5     running   franklin  friuli    0    catalog SELECT

(1 row(s) affected, return status = 0)
```

dbView has responded just as it did in the last example, only this time we used a stored procedure instead of an SQL statement. Notice the final status line. It not only gives the rows affected, it also includes status information. This is the value returned by the stored procedure. If the value isn’t “0”, the stored procedure is signaling something other than normal termination. In those cases you should also receive a message along with the status information.

*sp\_who* is just one of a set of system stored procedures supplied with every Sybase database server<sup>1</sup>. The names of system stored procedures are always prefixed with “sp\_”. Next, we’ll execute the procedure *sp\_helpdb* that returns a list of all database on the server you’re currently connected to; but before we do that, we’ll change the format of the table so that the rows returned are in list format instead of table format. (*Set* is a dbView command, and is unrelated to Sybase. All dbView command execute immediately. You don’t need to—an shouldn’t—type “go” following a dbView command.)

```
1> set format list
1> sp_helpdb
2> go

Row 1>
      name = catalog ← the name of a database
      db_size = 236 MB
      owner = sa
      dbid =      4
      created = Mar 11, 1993
      status = no options set

Row 2>
      name = master
      db_size = 5 MB
      owner = sa
      dbid =      1
```

---

1 For more information about Sybase system stored procedures, look at the references in the bibliography that discuss Sybase.

```

created = Jan 01, 1900
status = no options set

```

Row 3>

```

name = model
db_size = 2 MB
owner = sa
dbid =      3
created = Jan 01, 1900
status = no options set

```

Row 4>

```

name = tempdb
db_size = 40 MB
owner = sa
dbid =      2
created = Mar 30, 1993
status = select into/bulkcopy

```

```
(return status = 0)
```

The procedure returned the names of four databases. The databases, `master`, `model` and `tempdb` are present on every Sybase database server, and are used by the server for administration purposes. The other database, `catalog`, is the one we're connected to, as we saw when we executed the system stored procedure, `sp_who`.

To change the format so that we get results in tabular format once again, execute the `dbView` command:

```
1> set format table
```

We could have moved to another database by executing the Sybase Transact-SQL<sup>1</sup> command:

```
user <database>
```

Since this is a SQL command, we need to type “go” following it:

```

1> use master
2> go
Changed database context to 'master'.

```

## 8.2 Ending The dbView Session

You now know the basics of using `dbView` with one exception—you don't know how to end a

---

<sup>1</sup> Sybase's implementation of SQL, which contains many extensions to the language, is called Transact-SQL.

session; but that's easy. Just type *exit* as a new command.

```
1> exit
```

At this point, dbView exits and returns us to the system prompt.

## 8.3 What You've Learned

In this section we:

- Started dbView
- Connected to a database
- Executed SQL statements and stored procedures.
- Used the dbView set command.
- Exited the program.

There is one rule to remember from this section:

SQL commands—and stored procedures are SQL commands—are executed by following the command with the word “go” on a line by itself. dbView command—like *set*—execute immediately once you press the <return> key.

So you're 15 minutes are up and you now can use dbView to retrieve data from a database. Congratulations! If you want to learn more about dbView's capabilities, read on.

## 9 Running And Exiting From dbView

Before running dbView, you must have a valid Sybase environment on your machine and you must be able to access one or more Sybase database servers. For more information on the Sybase environment, refer to the *Installation Guide*.

To run dbView, type the name of the program at your system prompt:

```
% dbView
```

dbView will first prompt you for database server connection information, which we cover in the next section. If dbView encounters an error condition, you'll receive one or more messages. For more information on error messages see "*Some Common Login Errors*" on page 138 and see "*Error Messages*" on page 136.

When you finished with dbView and want to exit, type *exit* as a new command (New commands always begin with a prompt of "1>". To reset the command line to "1>", type *reset*.) In the following example, we'll reset the command line and exit.

```
3> reset
```

```
1> exit
```

The full syntax for the exit command is

```
exit [noSave]
```

If you include the "noSave" option, dbView will not save your current configuration. Use this options when you've made changes to the configuration that you don't want to take effect the next time you run dbView; for example:

```
1> exit noSave
```

(The significance of this form of the *exit* command will be clearer once we have introduced more of dbView commands.)

## 10 Connecting To A Database Server

In the previous section, “Getting Started” we made an initial connection to a database server. Once we’ve initially supplied dbView with login information, these values are saved and used as the default connection values the next time we connect to a server. Default values appear in square brackets [ ] following the prompt. To accept a default value, just type <return> at the prompt. Otherwise, enter the new value. Any new values supplied immediately become the new default.

Suppose we are now running dbView for the second time and we want to use the same connection information. The only thing we need supply is the password value:

```
% dbView
```

```
dbView, version 1.0, May 15, 1993  
Copyright 1993, The Jet Propulsion Laboratory. All rights  
reserved.
```

```
userName [franklin]: <return>  
password: <password><return>  
server [CATALOGDBS]: <return>  
database [catalog]: <return>  
DBMS Type [Sybase]: <return>
```

```
1>
```

### 10.1 The Connect Retry Prompt

If an error occurs while dbView attempts to make a connection, you will receive an error message. Following the error message you will see the line:

```
Try again? { y | n } [y]:
```

dbView is asking if you want to try to make the connection again. If you press the <return> key, you accept the default value, which is “y” (yes). (A default value in dbView always appears in square brackets. The values within curly braces show you the possible response values.) Accepting the default or typing “y” will get the connection prompt sequence again:

```
Try again? { y | n } [y]: <return>
```

What happens if you type “n” (no)? dbView does not exit; rather it leaves you at its command prompt because there are a couple of things you may want to do within dbView even though you’re not connected to a database server.

### 10.1.1 Increasing The Time-out Period

You may have “timed-out”. That is, you may be on a busy network. By default, dbView waits for 60 seconds for a response from the server. If it doesn’t receive a response, it assumes it can not make a connection. To make dbView wait for a longer period, you’ll have to increase the time-out value, but you can only do that from within dbView, so you’re left at the command prompt where this can be done.

To increase the time-out period, use the command:

```
set timeout <number of seconds>
```

For example, let’s increase the time-out period to 180 seconds:

```
1> set timeout 180
```

The command takes effect as soon as you press the <return> key. The next time you connect, dbView will wait 180 seconds. (As we’ll see in a moment, you don’t have to exit from dbView to re-connect.)

### 10.1.2 The Show Server Command<sup>1</sup>

If you were unable to connect to a server, it may be that you have used the wrong database server. Before attempting to connect again, use the *show server* command to get the list of servers that your copy of dbView can recognizes. For example:

```
1> show server
    - CATALOGDBS
    - CDB
    - PDSYBASECN
    - SYSTEM10
```

You should used one of the names in the list as the server name when you make a dbView connection to a database server.

You don’t have to be connected to a server to execute the *show server* command; the information is derived from your local copy of the Sybase *interfaces* file which was mentioned in the introduction.

### 10.1.3 Can’t Find The Interfaces File

Sometimes dbView can not connect to a server because it can’t find a copy of the *interfaces* file. In this case, you will receive an error message at connection time:

```
userName [franklin]:
password:
server [CATALOGDBS]:
```

---

<sup>1</sup> This command will support Illustra in the next version of dbView.

```
database [catalog]:
DBMS Type [Sybase]:

MDMS DBLIB MSGFAILED milano::General Delivery Fri Apr 23
11:35:39 1993
MsgNo: 20015, Svr: 3
Could not open interface file.

MDMS PROGRAM ERROR milano::dbView Fri Apr 23 11:35:39 1993
dbopen error for SQL command: CATALOGDBS server connection

Try again? { y | n } [y]: n
```

Executing the *show server* will also tell you if you can access the interfaces file. If dbView can't find it, you'll see an error message:

```
1> show server

MDMS PROGRAM ERROR milano::dbView Fri Apr 23 11:35:51 1993
Could not open interfaces file - /usr/sybase/interfaces.
```

If you see this sort of messages, then your Sybase environment is not set-up correctly—see the *Installation Guide* for help.

#### 10.1.4 Getting The List Of Databases And Accessing A Database

At this point, you might wonder if dbView has a command to show database. The answer is no, because there is a Sybase stored procedure that does that. Just type the command

```
1> sp_helpdb
2> go
```

and you'll retrieve information on all the databases on the server to which you're connected.

There are other useful Sybase stored procedures. For a complete description of these, see the *Sybase Command Reference* manual.

Once we're connected to a server, we can access a particular database using the Sybase Transact-SQL command

```
user <database name>
```

For example, to access the catalog database, we would execute the command:

```
1> use catalog
2> go
Changed database context to 'catalog'.
```

(Note: Your Database Administrator must grant you the privilege to access a database before this command will actually allow you to enter the database.)

If you don't have the privilege to access a database, you'll get an error message that looks like this:

```
1> use payroll
2> go

MDMS DBS WARNING milano::dbView Thu Mar 17 07:32:11 1994
(Db: jar, MsgNo: 916, Svr: 14, St: 1)
Server user id 3 is not a valid user in database 'payroll'
Changed database context to 'catalog'.
```

### 10.1.5 Exiting After Connection Failure

Since dbView always leaves you at the command prompt following a connection attempt, you must use the *exit* command to terminate the session:

```
1> exit
```

The *exit* command is only recognized as a new command, i.e., the command line number is 1. If you're not at command line 1, type <control-c>—the “control” and “c” keys pressed simultaneously. That will cancel any current command and return you to the point at which you can exit.

### 10.1.6 The Connect command

You don't have to exit dbView to connect to another server or to reconnect to one after a connection has been dropped for some reason. While in dbView, you make a database connection with the *connect* command. While your in dbView you can make changes to its environment as we'll see in subsequent sections. When you use the *connect* command, you maintain your environment across database servers. For example, suppose we are connected to the server CATALOGDBS, and we now want to get some data from the server PSYBASECN as the user “anonymous”. (If the DBMS type were different, we would enter its name well.) We execute the connect command and change the necessary default values:

```
1> connect
userName [franklin]: anonymous<return>
password: <password><return>
server [CATALOGDBS]: PSYBASECN<return>
database [catalog]: <return>
DBMS Type [Sybase]: <return>

1>
```

We're now connected to the database server PSYBASECN as "anonymous". Notice that we also kept the default database name, "catalog".

### 10.1.7 Using A Default Database

dbView always prompts you for a database name. Since Sybase always places you in a default database, you may want to skip this and just let the database server use the default it has for you. To use the default database defined by the *database server*, supply a question mark as your database name in dbView, like so:

```
1> connect
userName [franklin]: <return>
password: <password><return>
server [CATALOGDBS]: <return>
database [catalog]: ?<return>
DBMS Type [Sybase]: <return>
```

The question mark indicated that you're not specifying the database. Instead, the database server should select your default.

## 11 Connecting To Multiple DBMS's Simultaneously

If you only use one type of DBMS and you only plan to make one connection to a database at a time, you can skip this section.

### 11.1 Defining Multiple Connections

dbView can maintain multiple simultaneous connections, and different connections can be made to different types of DBMS's.<sup>1</sup> To do this we need a way to refer to different connections. dbView used *handles* to do this. A handle is a name you supply that is associated with the set of information you supply when you make a connection. The full syntax for the connect command is:

```
connect [<user supplied handle> | default]
```

An example will explain how the command is used. Lets make an initial connection to dbView; the one you make when you first invoke the program. Then lets make two more connections, one to mapping data another to telemetry data.

First the initial connection.

```
dbView, version 1.4, (dblib, milib), 28 Nov 1994
Copyright 1993, The Jet Propulsion Laboratory. All rights
reserved.
```

```
User name []: franklin
password:
server []: CATALOGDBS
database []: catalog
DBMS Type []: Sybase
```

Now the two additional connections. The initial connection is still maintained because we supply handles with the two new connections so dbView can keep track of all three.

```
1> connect mapping
User name [franklin]:
password:
server [CATALOGDBS]: Illustra1
database [catalog]: planetMaps
DBMS Type [Sybase]: Illustra

1> connect telemetry
User name [franklin]: madison
password:
```

---

<sup>1</sup> This capability is Operating System dependent. Currently dbView supports Sybase connections on Sun OS, Sun Solaris 2, HP-UX, SGI, DEC VAX VMS and Open VMS, and NeXTStep operating systems. dbView also supports Illustra on Sun Solaris 2 and NeXTStep operating systems.

```
server [CATALOGDBS]: TELEM
database [catalog]: telem
DBMS Type [Sybase]:
```

Let's make some observations on what we've done so far.

1. Whenever we make a new connection, that becomes the current connection. (We'll show you how to change connections in a moment.)
2. The default parameters for a new *connection* command always come from your initial connection. If you supply a new parameter for the handle, the parameter becomes the default for the new handle.
3. You only have to change parameters that will be new for the handle you're defining. Notice in the third connection how we changed the user name but did not supply a DBMS Type.
4. If you look at the syntax for the *connect* command again, you see that it includes a special handle named *default*. That's the handle associated with your initial connection. We'll see how it's used in the next section.
5. There's an advantage to using handles if you have many DBMS's to connect to, or even if you have to use different names for different privileges within a single database server. And that advantage is: there's less for you to remember. Instead of knowing five parameters for a connection, you only need to remember one—the handle's name, which you define. And, once defined, dbView remembers your handles. They're available in subsequent sessions, so the advantage remains.

## 11.2 Using Connection Handles

Now that we've got three connections—see the example in the last section—we can change connection with the handles. To use a connection you've defined, just include the handle's name with the *connect* command. For example:

```
1> connect mapping
```

Since we've got this connection defined, dbView just changes the connection; it doesn't re-prompt you for new connection parameters. We're now connection to "mapping". Any commands we send to a database server go to the one represented by that handle.

Let's change connections a couple of more times.

```
1> connect telemetry
```

```
1> connect default
```

The last connection takes us to our initial connection—the one we made when we first entered dbView. Why do we need the special handle "default" in this case; why can't we just say *connect*? For people using sequential connections, dbView must provide a way to disconnect and

connect anew to another database server. If you just typed *connect*, that's what will happen. Your connection is dropped and you're prompted for new connection information. In this way, if you're not using handles, dbView hides the entire issue from you. If you are using handles, then you always provide a handle, even in the default case. Both methods are consistent.

With multiple connections, you can forget where you are, so dbView provides a command for finding out that information. The command is:

```
show handle
```

Let's connect to "mapping" and try it.

```
1> connect mapping
```

```
1> show handle
```

```
default      connected
mapping      connected    using
telemetry    connected
```

We've got three handles defined and we're connected to all of them. We're currently using the "mapping" handle. If we now connect to the "telemetry" handle, we see this:

```
1> connect telemetry
```

```
1> show handle
```

```
default      connected
mapping      connected
telemetry    connected    using
```

We can drop a connection with the command:

```
disconnect <handle name>
```

We'll drop the connection associated with the "telemetry" handle and then use *show handle* again.

```
1> disconnect telemetry
```

```
1> show handle
```

```
default      connected    using
mapping      connected
telemetry
```

We're no longer connected to the telemetry handle. When we disconnect from a handle we're using, dbView will return the default connection whether or not it's connected.

Notice that the "telemetry" handle is still defined, it's just not connected any more, so we can still use it in the future to make a new connection. When we reconnect to a defined handle, we'll be prompted by the *connect* command again:

```
1> connect telemetry
User name [madison]:
password:
server [TELEMCAT]:
database [telem]:
DBMS Type [Sybase]:
```

Notice that the handle maintains its default values. You could change them when prompted, but all that's necessary is that you enter your password again and press <return> for any default parameters you want to accept.

When you exit dbView, all of your defined handles are written into the `.dbView` file. The next time you run dbView it will read the contents the `.dbView` file, so you can connect with your defined handles immediately—there's no need to define them in each dbView session.

When you first use a handle in a new dbView session, you must enter the password, so dbView prompts you for the handle's parameters. If you want to redefine one or more parameters for a handle that's connected, just disconnect and the reconnect:

```
1> connect telemetry ← connected, so no prompt

1> disconnect telemetry

1> connect telemetry ← not connected, so we get prompted
User name [madison]: hamilton
password:
server [TELEMCAT]:
database [telem]:
DBMS Type [Sybase]:
```

Now we've change the handle to connect as "hamilton" instead of "madison".

If you just enter the command "connect", you're always prompted for parameters which are applied to the default handle. This is done for the benefit of people not using handles. In their case, they have no need to know anything about handles, so dbView gives them sequential access to connections. Issuing the *connect* command without a handle name, drops the current default connection, prompts you for new connection parameters and then makes a new connection.

You can also remove a handle completely so it no longer appears in your dbView session. You do this using the command:

```
remove handle <user supplied handle>
```

Notice that it only says *user supplied handle*; you can't remove the "default" handle. But, you can redefine it by simply issuing the *connect* command without a handle name as explained above.

dbView won't let you remove a handle that it's using for a connection—it would lose track of

the connection in that case, so you must first disconnect. Here's an example:

```

1> show handle
      default      connected      using
      mapping      connected
      telemetry    connected

1> remove handle telemetry ← won't work, we're connected
      Cannot remove connected handle. Disconnect and try
      again.

1> disconnect telemetry

1> remove handle telemetry

1> show handle
      default      connected      using
      mapping      connected

```

The first time we tried to remove the “telemetry” handle we were using it, so dbView issued an error message and kept the connection. Once we disconnected, we were able to remove the handle.

## 11.3 Handles And The Password Server<sup>1</sup>

The password server stores database passwords. To use it you must be registered with a Kerberos and a password server that work together to supply passwords and you must currently have a valid Kerberos ticket—which you get using kinit. (If you're not sure about whether or not you have a password server, ask your DBA. If you're not using one, skip this section.)

If you are using a password server and you're a valid user in that server, you can connect immediately without supplying a password with handles. When you define a handle, don't supply a password; you'll be connected using the password from the password server. If you enter a new dbView session and your handles are already defined, you can use them immediately, dbView won't prompt for any information.

Before you try this, you'll have to make one addition setting. By default, dbView always prompts, so you have to tell it not to. To do this you use the following command:<sup>2</sup>

```
set promptOnConnect {on | off}
```

By default it is set to “on”. To work with the password server, you want to set it off, so dbView suppresses prompts:

<sup>1</sup> Not currently supported on VAX and NeXTStep.

<sup>2</sup> This is just one of dbView's *set* commands. The *set* commands are introduced later on, but this is the logical place to describe the *promptOnConnect* setting, so we introduce it early.

```
1> set promptOnConnect off
```

Now you won't be prompted. If you did want to be prompted, turn on the setting.

## 11.4 Handles And dbView's Batch Mode

Handles and the password server can be used with a special dbView *batch mode*. Batch mode is discussed in its own section, so, for now, we'll just tell you what it does. In batch mode, you supply the name of a *script file* on dbView's command line. dbView executes the commands in the script file and never enters its *interactive mode*—the mode we've been in all along. For this to be possible, dbView needs a way to get passwords associated with the connections made in the script file. It goes to the password server to get them.

Later you'll see that batch mode has many uses. Suppose, for example, that your script file contains a set of *report* commands that generate reports from a set of database servers that are place throughout your enterprise and that you want to run these reports nightly at three in the morning. The results of the reports should be emailed to a group of people once the reports are generated. Having handles and the password server makes this possible as you'll discover later on.

## 12 Executing A Command

Once connected to a database server, dbView waits at its command prompt for input from you. You can execute two types of commands within dbView: dbView commands and SQL commands. All dbView commands are entered on a single line and are executed as soon as you press the return key. SQL commands can extend over several lines, so dbView needs a way to know that you've completed the command. The special terminator "go", placed on a line by itself, signals the end of a SQL command. (Remember that Sybase stored procedures are the same as SQL commands from dbView's point of view.)

Show server is an example of a dbView command. As soon as you type it in and press the return key, the list of servers is displayed.

```
1> show server
    - CATALOGDBS
    - CDB
    - PDSYBASECN
    - SYSTEM10
```

The next example shows an SQL command entered on 3 lines. The fourth line has the "go" terminator. Notice the results are retrieved by dbView as soon as it recognizes the "go" terminator.

```
1> select mission, scId
2> from missions
3> order by mission
4> go ← this is command terminator
```

mission	scId
Cassini	72
GLL	35
MO	91
VGR	0

(4 row(s) affected)

In the next example, we show a stored procedure command. Even though it can be entered on a single line, it must be followed by the "go" terminator because it's an SQL command.

```
1> showMissions
2> go

                MISSION INFORMATION

mission          scId          objective
-----
Cassini          72           Saturn
GLL              35           Jupiter
```

```
MO          91  Mars
VGR         0   NULL

(4 row(s) affected, return status = 0)
```

```
1> ← dbView's signal for the next command
```

Here's the underlying rule: Any command string dbView does not recognize as one of its own, is assumed to be the beginning of a database command, so dbView keeps accepting input lines until you terminate the command.

## 12.1 Cancelling A Command

If you make a mistake while entering a command, you can cancel it and start over.

### 12.1.1 Cancelling A Command Using <control-c>

When you are in the middle of a command, typing <control-c> (the *control* key and the *c* key pressed simultaneously) cancels the command. dbView then prompts with command line 1 again, signalling that you can enter a new command. In the following example, the word "select" is miss-spelled as "sedect". When we notice this, we type <control-c> and the command is cancelled. We then type and retype the command correctly. (Use of the edit command is another option for correcting a command, see "*Editing Commands*" on page 41.)

```
1> sedect mission
2> from^C

1> select mission
2> from missions
3> go
```

### 12.1.2 Cancelling A Database Command Using <control-c>

You can also use <control-c> to interrupt a database command that is in the process of executing. This is most often used to terminate query results before all of the rows are returned. As dbView displays the rows for a query, type <control-c>. As soon as the current row buffer is emptied, the query is terminated and you're returned to the command line prompt.<sup>1</sup>

### 12.1.3 The reset Command

You can also cancel a dbView command by typing *reset* on the next command line, for example:

```
1> select mission, scId
```

---

<sup>1</sup> For some operating systems, most notably DEC's VMS, you will have to press the <return> key following <control-c> before the cancel will take effect. This is the nature of the operating system, not dbView.

```
2> reset
```

```
1>
```

*Reset* only cancels a command in dbView's command buffer. It can not be used to terminate a database command that is in the process of executing.

Why does dbView have the *reset* command when <control-c> works just fine to cancel a command? In *scripts*—sets of dbview command that can be executed from a file—the *reset* command is used to end a comment. Text entered in the command buffer is normally a command, but in a script file it can be treated as a comment. Just type in whatever you want and then on the next line use the *reset* command. Follow that with the actual command. Here's a short example, see "*The Script Command*" on page 122, for more details.

```
1> The following SQL statement will bring back information
2> from the "planets" table which contains the names and
3> other information about the planets in the Solar System.
4> reset ← we don't want the comment executed

1> select * from planets
2> go
```

## 12.2 Editing Commands

In the previous section, we saw how to cancel a command to correct a mistake in an SQL statement. There is a better way to do this—use your editor to make the correction and then return to dbView. That way you don't have to type in the entire command again.

To use the editor, type in the *edit* command on a new command line, like so:

```
1> sedect mission
2> edit

... command corrected and completed in the editor

1> select mission, scId
2> from missions
3> order by mission
4>
```

Notice that when dbView returns from the editor it leaves you on a new command line. You can type "go" immediately to execute the command, or you can add more lines to the command.

dbView defines default editors: *vi* for Unix systems and *edt* for VMS, but you can change the default using the *set editor* command. The syntax is:

```
set editor <editor name>
```

For example, if you prefer to use *emacs*, type:

```
1> set editor emacs
```

dbView saves this setting, and will continue to use it in the current session and all subsequent sessions until you set it again.

## 12.3 When dbView Encounters An Error In An SQL Command

If you execute a command that contains errors, dbView returns one or more messages. For example, if you type a query with the word *select* miss-spelled:

```
1> sedect mission
2> from missions
3> order by mission
2> go
```

you get the following error messages back, (see "*What's In An Error Message?*" on page 136, for a discussion on how to interpret error messages):

```
MDM DBS WARNING milano::dbView Fri Nov 13 12:54:20 1992
(Db: catalog, MsgNo: 156, Svr: 15, St: 1)
Incorrect syntax near the keyword 'from'.
```

If you receive one or more error messages, correct the error in the editor and execute the command again.

## 12.4 The Set Command

We've mentioned the dbView *set* in passing several times. The *set* command sets a dbView parameter that effects dbView's environment in some way. The syntax for the *set* command is:

```
set <parameter> <value>
```

We can see all of the *set* commands using the *show set* command.

```
1> show set
- defaultMacroFile = <not defined>
- displayRows = on
- displayScriptCommands = on
- doublePrecision = 12
- editor = vi
- endField = \t
- endRow = \n
- feedback = off
- format = table
- header = on
- history = 20
```

```

- mailReport = off
- page = 0
- printReport = off
- promptOnConnect = on
- reals = f
- singlePrecision = 6
- spaces = 2
- timer = off
- timeout = 60
- verbose = off

```

The syntax for the command is:

```
show set [<set command parameter>]
```

As we've just seen, if you don't specify a *set* command parameter, dbView lists the values of all of the *set* commands.

When you supply a *set* parameter name with the *show set* command, dbView returns the values for that parameter alone.

```

1> show set timer

- time = on

1> show set format

- format = table

```

The *show set* command doesn't tell you the significance of a particular command nor does it tell you what values a command will accept. To get that information, use the *help* command, which we describe later on.

### 12.4.1 Set Command Definitions

We'll discuss how and where you'd use set commands further along in this document. In this section we just give their definitions so you can get an idea of what they do. We used dbView's *help* command to get this information. You could do the same on-line by typing *help set <command>*.

```
set defaultMacroFile <file name> [<file name> ...]
```

The full name of the file - including any directory specification - that contains macro definitions that should be read in when dbView is run. More than one file may be placed on this list.

Default: none

```
set displayRows { on | off }
```

If set to "off", rows are not returned when a query is executed. Most often used to determine the execution time of a query minus the time

for screen I/O.

Default: on

*set displayScriptCommands { on | off }*

If set to "off", commands executed from script files are not written to the screen. Useful for applications where the user must respond to prompts, or needs a clear view of output.

Default: on

*set doublePrecision <integer>*

The number of digits displayed for a double precision floating point number - 8 bytes in length on most machines.

Default: 12

Range: 1,...,18

*set editor <editor name>*

The name of the editor dbView invokes when you type the edit command.

Default: vi (Unix), edt (VMS)

*set endField <string>*

The set of symbols that separate fields in rows of data returned by the export table format.

Default: tab <\t>

*set endRow <string>*

The set of symbols that terminate a row, or record, when data is returned by the export table format.

Default: newline <\n>

*set feedback { on | off }*

If set to "on", dots will be displayed on the screen while waiting for the results of each query.

Default: off

*set format { table | list | export }*

The display format for data returned by a database query.

Default: table

*set header { on | off }*

If set to "off", the columns names and status line that normally appear as part of a information returned by a query are suppressed.

Default: on

*set history <list size>*

The number of commands kept in the history list.

Default: 10

Range: 1,...,100

*set mailReport { on | off }*

If a report has a list of eMail addresses associated with it, mail is not sent if the value of mailReport is "off".

Default: on

*set page <number of display lines>*

The number of lines in a page full of display data before dbView will stop and wait for you to signal for more. The signal for another page of data is <return>. If the number of display lines is set to 0, dbView understands this to mean an unlimited number of rows should be returned on a page.

Default: 10000

Range: 0,,10000

*set printReport { on | off }*

If a report contains a command to send the report to a printer, you can suppress that command by setting printReport of a values of "off".

Default: on

*set promptOnConnect { on | off }*

If set to "on", the user is prompted for connection information when using pre-defined handles. If set to "off" no prompt is made.

Default: on

*set reals { f | e | E | g | G }*

The format for real and floating point numbers. "f" is decimal format. "e" and "E" are scientific notation format. And "g" and "G" are mixed, depending on the magnitude of the value. The case of the letters "e" and "g" will be reflected in the scientific notation display. For example:

set reals e

real      float

-----

1.234456e+03    1.234567890000e+07

set reals E

real      float

-----

1.234456E+03    1.234567890000E+07

The number of digits displayed is controlled by the parameters singlePrecision and doublePrecision.

Default: f

*set singlePrecision <integer>*

The number of digits displayed for a single precision floating point numbers - 4 bytes in length on most machines.

Default: 6  
Range: 1,...,10

*set spaces <integer>*

The number of spaces between columns of data returned by a database query.

Default: 2  
Range: 1,...,80

*set timer { on | off }*

When set "on", database commands return the time it took to execute the command and to display any data returned.

Default: off

*set timeout <integer>*

The number of seconds dbView waits while attempting to connect to a database server.

Default: 60  
Range: 0,...,32767

*set verbose { on | off }*

When set to "on", causes the commands show db and show macro to display additional information. Also, if set to "on", causes any comment associated with a macro to be displayed before the macro is executed.

## 12.5 Saving dbView's Environment

When you exit dbView, your current environment is saved to a file. For Unix systems the file is named `.dbview`, and for VMS systems `.dbview.<version number>`. (dbView purges the `.dbview` file on VMS after writing a new one.) For both types of systems, the file is located in your logon—or home— directory.

The file contains all of the set command values plus your database server connection parameters. The next time you run dbView, the values in the `.dbview` file are read and used as your current environment.

If you've made changes to the set commands in a dbView session; and you don't want those changes saved, you can use a special form of the exit command:

```
1> exit nosave
```

dbView then exits without writing-out a new `.dbview` file.

## 12.6 Timing Commands

You can measure the length of time it takes to completely execute an SQL command, including the amount of time to display any rows returned, using the *set timer* command. The following example shows how the timer is set. Following the execution of each command sent to the

database server, the time is reported along with the number of rows returned, if there were any:

```
1> set timer on
```

```
1> select mission, scId
2> from missions
3> order by mission
4> go
```

mission	scId
Cassini	72
GLL	35
MO	91
VGR	0

```
(4 row(s) affected)
```

```
SQL statement took 0.04 seconds to execute.
```

```
1> set timer off
```

```
1>
```

Times are recorded to a hundredth of a second.

Reported times include the amount of time it took to return the data to your screen. If you want the time minus the screen I/O, use the *set displayRows* command to turn off screen I/O. (Later we'll see other instances where we might not want query results displayed.)

```
1> set displayRows off
```

```
1> select mission, scId
2> from missions
3> order by mission
4> go
```

```
(4 row(s) affected)
```

```
SQL statement took 0.02 seconds to execute.
```

Without the screen I/O, the query took half as long to complete. Turn the display back on with the command:

```
1> set displayRows on
```

## 12.7 Altering The Database Table Display Format

Data returned from a database server to dbView can be formatted in one of three ways: table, list or export. The syntax for the command that sets the current format is:

```
set format { table | list | export }
```

The curly braces enclose a list of options, one of which must be chosen. The vertical line means “or”, i.e., “table” or “list” or “export”. When you first begin using dbView, the default format is “table”.

### 12.7.1 Table Format

Here’s an example of table formatted data:

```
1> select mission, scId
2> from missions
3> order by mission
4> go

mission          scId
-----
Cassini           72
GLL               35
MO                91
VGR               0

(4 row(s) affected)
```

In table format you can adjust the number of spaces between each column with the *set spaces* command; for example:

```
1> set spaces 6

1> select mission, scId
2> from missions
3> order by mission
4> go

mission          scId
-----
Cassini           72
GLL               35
MO                91
VGR               0

(4 row(s) affected)
```

## 12.7.2 List Format

Sometimes the rows in a table are so long that the data wraps one or more lines on your screen. In the next example, we add some fields to our last query that cause this to occur.

```
1> select mission, scId, objective, description
2> from missions
3> go
```

mission	scId	objective	description
Cassini Saturn	72	Saturn	The Cassini Mission to Saturn
GLL to Jupiter	35	Jupiter	The Galileo Mission to Jupiter
MO sion	91	Mars	The Mars Observer Mis- sion
VGR the outer solar system	0	NULL	The Voyager Mission to the outer solar system

(4 row(s) affected)

If you find this difficult to read, you may want to try the *list* format:

```
1> set format list

1> last
1> select mission, scId, objective, description
```

```
2> from missions
3> go

Row 1>
      mission = Cassini
      scId =          72
      objective = Saturn
      description = The Cassini Mission to Saturn

Row 2>
      mission = GLL
      scId =          35
      objective = Jupiter
      description = The Galileo Mission to Jupiter

Row 3>
      mission = MO
      scId =          91
      objective = Mars
      description = The Mars Observer Mission

Row 4>
      mission = VGR
      scId =           0
      objective = NULL
      description = The Voyager Mission to the outer solar
system

(4 row(s) affected)
```

For this query, list format is a more readable format. Notice that each row is numbered and each column value is preceded by its name. (Also, notice the use of the *last* command to recover the database command we had previously typed in.)

*List* format works best when a few rows are returned. If many rows are returned, the output will take up many of lines on your screen.

### 12.7.3 Export Format

Export format is used to create a data set that will be exported to another program, like a spread sheet, word processor or database import program.

In *export* format, data comes back in rows like it does in *table* format, but the fields are not aligned. Instead a field delimiter is placed between each field's value. Also, lines are terminated with a row delimiter character. Programs that import data often require this format. For example, Sybase's data import program, *bcp*—bulk copy—imports data separated by delimiters. The *bcp* defaults are tabs between fields and newlines between rows. These are the default

values used by `dbView`.

In the next example we return data in export format. We use a comma—defined with the `set endField` command—as the field delimiter so that the delimiter can be more easily seen:

```
1> set format export
1> set endField ,
1> select mission, scId
2> from missions
3> order by mission
4> go
```

```
mission,scId
Cassini,71
GLL,31
MO,95
VGR,0
```

```
(4 row(s) affected)
```

Later we'll describe how this data set can be saved to a file where it can be read by another program.

As we've mentioned, the default field delimiter is the tab character and the default row delimiter is the newline character. You redefine these values using the commands:

```
set endField <character string>
```

```
set endRow <character string>
```

For example:

```
1> set endField ,
1> set endRow \r
```

The “\r” is a special representation of the ASCII “carriage return” character. Both `set endField` and `set endRow` commands accept the special characters listed below:

```
\b    back space
\f    form feed
\n    new line
\r    carriage return
\s    white space
\t    horizontal tab
\v    vertical tab
```

```
\\      back slash
\000   octal number
```

If you want several blanks between each field, use the “white space” character. The following command would put four blanks between each field. (Of course, for spaces, you would be better off just using the set spaces command.) Since a row delimiter follows the last field, there would be no blanks following the last field:

```
1> set fieldDelimiter \s\s\s\s
```

#### 12.7.4 Copy And Pasting Exported Data

If you’re using dbView in a windowing environment, you can copy and paste results between windows. In fact, that’s how most of the examples in this document were created. In the next example, we produce some output that is pasted into a table in a page layout program—we’re using Frame Maker for the example. The steps used are:

1. Set the *format* to “export”, the *endField* to “,” and the *endRow* to “\n”.

```
1> set format export
```

```
1> set endField ,
```

```
1> set endRow \n
```

2. Execute the SQL statement.

```
1> select mission, scId, objective, description
2> from missions
3> order by mission
4> go
```

```
mission,scId,objective,description
Cassini,72,Saturn,The Cassini Mission to Saturn
GLL,35,Jupiter,The Galileo Mission to Jupiter
MO,91,Mars,The Mars Observer Mission
VGR,0,NULL,The Voyager Mission to the outer solar system
```

```
(4 row(s) affected)
```

3. Select and copy the data to the paste buffer from dbView’s terminal window and then paste it into the Frame Maker document—or some other word processing program. In Frame Maker, we then use the command “Convert to Table...” to produce the following

table from the export format data:

mission	scId	objective	description
Cassini	72	Saturn	The Cassini Mission to Saturn
GLL	35	Jupiter	The Galileo Mission to Jupiter
MO	91	Mars	The Mars Observer Mission
VGR	0	NULL	The Voyager Mission to the outer solar system

*Table 1: Mission Data*

You should begin to see the possibilities of export format.

## 12.8 Displaying Table Header And Footer Information

When using table or export *format*, header and footer information is normally included. The header consists of a row of field names above the data, separated from the data by dashed lines. The footer contains the status line returned following the completion of an SQL statement. You can turn off the display of header and footer information using the command:

```
set header {on | off}
```

This command is particularly useful when using the *export format* to produce a data file that will be imported by a spreadsheet or database loading program because they normally only want to accept the data to be loaded. We can repeat one of our previous examples with the headers and footers suppressed:

```
1> set header off

1> set format export

1> select mission, scId
2> from missions
3> go

Cassini,72
GLL,35
MO,91
VGR,0
```

## 12.9 The Page command

When data is returned to you it will scroll down the screen until a page is full. A page is defined as the number of lines of text displayed, or the number of rows returned from a database query. By default the number of lines per page is 10,000. If you find lines are scrolling

past your screen before you can view them, set the page size to something smaller, 24 for example.

```
1> set page 24
```

After this setting, 24 lines are displayed and then dbView stops writing output to your screen. To get the next page of data, press the <return> key.

Note: If the database is returning a large number of rows and you don't want to page through the rest of data, cancel the query using <control-c>; you'll be returned to the command line prompt once the query is cancelled. (When cancelling on a page break, you'll have to press the <return> key following <control-c> to get back to the prompt.)

The syntax for the set page command is:

```
set page [number of lines]
```

where the number of line or rows displayed per page can be between 1 and 10,000. The *set page* command also takes the special value of 0 which set the page size to its maximum value.

```
1> set page 0
```

## 12.10 Real Number Formats

Sybase has two types of floating point numbers. One type is called *real* and uses 4 bytes to store the number internally. The second type is called *float* and uses 8 bytes to store the number internally. In dbView these numbers are referred to as single and double precision floating point numbers.

You can set the precision of single and double precision numbers independently using the commands:

```
set singlePrecision <digits>
```

```
set doublePrecision <digits>
```

By default, single precision numbers have a precision of 6 digits and double precision numbers 12 digits. The interpretation of precision depends on the real number format chosen. The command to set this real number format is:

```
set reals { f | e | E | g | G }
```

### 12.10.1 The f Format

The single and double precision floating point numbers are converted to decimal notation in the style “[-]ddd.ddd” where the number of digits after the decimal point is equal to the precision specification.

In the following example the field *IgtYrsFromSun* is a double precision number and the field

*hrsPerRotation* is a single precision number.

```
1> select number, name, lgtYrsFromSun, hrsPerRotation
2> from planets
3> order by number
4> go
```

number	name	lgtYrsFromSun	hrsPerRotation
1	Mercury	0.000006307862	211.283997
2	Venus	0.000011422344	NULL
3	Earth	0.000015854896	24.000000
4	Mars	0.000024208551	24.664000
5	Jupiter	0.000082343170	10.021000
6	Saturn	0.000151047719	10.273000
7	Uranus	0.000303459300	10.801000
8	Neptune	0.000475646880	15.898000
9	Pluto	0.000662567170	NULL

Now we change the precision and run the command again.

```
1> set singlePrecision 3
1> set doublePrecision 5
1> select number, name, lgtYrsFromSun, hrsPerRotation
2> from planets
3> order by number
4> go
```

number	name	lgtYrsFromSun	hrsPerRotation
1	Mercury	0.00001	211.284
2	Venus	0.00001	NULL
3	Earth	0.00002	24.000
4	Mars	0.00002	24.664
5	Jupiter	0.00008	10.021
6	Saturn	0.00015	10.273
7	Uranus	0.00030	10.801

number	name	lgtYrsFromSun	hrsPerRotation
8	Neptune	0.00048	15.898
9	Pluto	0.00066	NULL

### 12.10.2 The E or e Format

The single and double precision floating point numbers are converted in the style “[-]d.ddde±ddd”, where there is one digit before the decimal point and the number of digits after it is equal to the precision. The “E” format code will produce a number with “E” instead of “e” introducing the exponent. The exponent always contains at least two digits and can be as great as ±999.

In this example we use the default precision values with a real number format of “E”.

```
1> set reals E

1> select number, name, lgtYrsFromSun, hrsPerRotation
2> from planets
3> order by number
4> go
```

number	name	lgtYrsFromSun	hrsPerRotation
1	Mercury	6.307861850000E-06	2.112840E+02
2	Venus	1.142234440000E-05	NULL
3	Earth	1.585489600000E-05	2.400000E+01
4	Mars	2.420855090000E-05	2.466400E+01
5	Jupiter	8.234316950000E-05	1.002100E+01
6	Saturn	1.510477190000E-04	1.027300E+01
7	Uranus	3.034593000000E-04	1.080100E+01
8	Neptune	4.756468800000E-04	1.589800E+01
9	Pluto	6.625671702000E-04	NULL

And now, keeping the “E” format we reset the precision.

```
1> set singlePrecision 3

1> set doublePrecision 5

1> select number, name, lgtYrsFromSun, hrsPerRotation
2> from planets
3> order by number
```

```
4> go
```

number	name	lgtYrsFromSun	hrsPerRotation
-----	-----	-----	-----
1	Mercury	6.30786E-06	2.113E+02
2	Venus	1.14223E-05	NULL
3	Earth	1.58549E-05	2.400E+01
4	Mars	2.42086E-05	2.466E+01
5	Jupiter	8.23432E-05	1.002E+01
6	Saturn	1.51048E-04	1.027E+01
7	Uranus	3.03459E-04	1.080E+01
8	Neptune	4.75647E-04	1.590E+01
9	Pluto	6.62567E-04	NULL

### 12.10.3 The g or G Format

The single and double precision floating point numbers are style “f” or “e” (or in style “E” for an upper case “G” format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style “e” or “E” will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

In this example we set the real number format to “G” and reset both precision values to “2”.

```
1> set doublePrecision 2
```

```
1> set singlePrecision 2
```

```
1> set reals G
```

```
1> last
```

```
1> select number, name, lgtYrsFromSun, hrsPerRotation
```

```
2> from planets
```

```
3> order by number
```

```
4> go
```

number	name	lgtYrsFromSun	hrsPerRotation
-----	-----	-----	-----
1	Mercury	6.3E-06	2.1E+02
2	Venus	1.1E-05	NULL

number	name	lgtYrsFromSun	hrsPerRotation
3	Earth	1.6E-05	24.
4	Mars	2.4E-05	25.
5	Jupiter	8.2E-05	10.
6	Saturn	0.00015	10.
7	Uranus	0.00030	11.
8	Neptune	0.00048	16.
9	Pluto	0.00066	NULL

#### 12.10.4 The Feedback Command

If feedback is set on, and a database transaction takes several seconds to complete, dbView will begin displaying a line of dots on your terminal screen until the transaction completes or until the first row of data from a query is returned. The dots appear at about one second intervals. To see this, we'll give the database server something hard to do in the next query.

```
1> set feedback on
1> select distinct (a.name)
2> from sysobjects a, sysobjects b
3> order by a.id
4> go
..... ← feedback dots
name
-----
sysobjects
sysindexes
more rows follow
```

Feedback may be a misnomer; heartbeat might be better, because, dbView is just waiting for the transaction to be executed by the database server. Unless there is an error, the server does not communicate with dbView during this time period, so what we're really seeing is dbView's heartbeat while it's waiting on the server.

#### 12.10.5 Using Sybase's "Set" Command In dbView

Sybase also has a *set* command, but if you try to execute it within dbView, you'll get an error message, because dbView regards any *set* command as one of its own. For example, since *set showplan* is a Sybase and not a dbView command, you get the error message:

```
1> set showplan on ← a Sybase command
Unknown set parameter "showplan".
```

To get around this predicament, create a Sybase *batch* command between the words *begin* and *end*, like so:

```
1> begin
2> set showplan on
3> end
4> go
STEP 1
The type of query is SETON.
```

In this case, since the command started with the word *begin* instead of *set*, dbView recognized it as a command to be sent to the database server<sup>1</sup>.

---

1 For more information on the Sybase *set* command, refer to the Sybase *Command Reference* manual.

## 13 Escaping To The Operating System

You can execute an operating system level command from within dbView by preceding the *operating system* command with the *dbView* command *escape*. The command's syntax is:

```
escape <operating system command>
```

For example, to list the names of all files with the extension “.sql” use the following command:

Unix

```
1> escape ls -l *.sql  
dbSize.sql  
dbView.sql  
domains.sql  
dropAll.sql  
showDb.sql
```

VMS

```
1> escape dir [*.sql;*]  
dbSize.sql;1  
dbView.sql;1  
domains.sql;1  
dropAll.sql;1  
showDb.sql;1
```

Once the shell command completes, you are returned to dbView.

As another example, we could use the *escape* command to view a file with an editor:

Unix

```
1> escape vi dbView.sql
```

VMS

```
1> escape edt dbView.sql
```

## 14 The Help Command

Once you read this tutorial, you'll want to refer to definitions of `dbView` command once in a while. Use the on-line help facility for that. You can get on-line help with the `help` command. *Help* will give you as much information as it can on a topic. If you just type `help`, you get the list of subtopics for which there is help:

```
1> help
Subtopics:
close          connect       directory
disconnect     edit          escape
exit           expand        global
go             help          history
include        info          last
leслиe         macro         open
print          remove        rename
repeat         replace       report
reset          run           save
script         set           show
```

If you include a topic from a subtopic list as part of the `help` command, `help` will:

1. Redisplay the topic.
2. Explain what the topic does.
3. If there are subtopics for the command, *help* will display the new list of subtopics.

For example, the `set` command takes many parameters as part of the command. If we type "help set", we'll get information about `set` and the list of subtopics that pertain to the `set` command:

```
1> help set
Topic:
set
```

The `set` command allows you to assign a value to a parameter within `dbView`.

```
Subtopics:
defaultMacroFile  displayRows    displayScriptCommands
doublePrecision   editor          endField
endRow            feedback       format
header            history        mailReport
page              printReport    promptOnConnect
reals             singlePrecision spaces
timer             timeout        verbose
```

If we now choose a subtopic, like “set page”, we’ll get further information. Since no subtopics are displayed for “set page”, we know that there is no further level of detail for this topic:

```
1> help set page
```

```
Topic:
```

```
set page
```

```
set page <number of display lines>
```

```
The number of lines in a page full of display data before dbView will
```

```
stop and wait for you to signal for more. The signal for another page
```

```
of data is <return>. If the number of display lines is set to 0, dbView understands this to mean an unlimited number of rows should be
```

```
returned on a page.
```

```
Default: 10000
```

```
Range: 0,,10000
```

If you enter a topic that help does not recognize, it will give you all the help it can, as the next couple of examples show.

“sat” is not a command, which *help* tells us. Then it shows its main list of topics.

```
1> help sat
```

```
WARNING: extra text ignored: sat
```

```
Subtopics:
```

close	connect	directory
disconnect	edit	escape
exit	expand	global
go	help	history
include	info	last
leslie	macro	open
print	remove	rename
repeat	replace	report
reset	run	save
script	set	show

Help knows about “set”, but not about “set nothing”, so it gives us a warning message to that effect; and then shows us what it can, in this case information about the *set* command.

```
1> help set nothing
```

```
WARNING: extra text ignored: nothing
```

```
Topic:
```

```
set
```

```
The set command allows you to assign a value to a parameter
```

within dbView.

Subtopics:

defaultMacroFile	displayRows	displayScriptCommands
doublePrecision	editor	endField
endRow	feedback	format
header	history	mailReport
page	printReport	promptOnConnect
reals	singlePrecision	spaces
timer	timeout	verbose

The syntax for the help command is:

```
help [<topic>]
```

## 14.1 The History List

As you enter and execute command, dbView saves them in a history list. You can view the list of previous commands using the *history* command:

```
1> history

---1
show set

---2
select mission, scId
from missions

---3
select objective, mission, scId
from missions
order by objective

---4
showMissions GLL

---5
set page 10

---6
showMissions GLL

1> history 2
1> select mission, scId
2> from missions
3>
```

You recall any command in the history list, making it the current command, by adding the command's history number to the *history* command. In the example above, we chose the second command, history 2. Notice that we are left with a new command line following the command's retrieval, so we can decide what to do from there. We can edit the command, execute it, add to it, etc. For example:

```
1> history 2
1> select mission, scId
2> from missions
3> edit
```

...enter the editor where the statement is modified; then return

```
1> select mission, scId, planet = name
2> from missions, planets
3> where missions.objective = planets.name
4> go
```

The syntax for the history command is:

```
history [<command number>]
```

The square brackets indicated that the number is optional.

### 14.1.1 Setting The Length Of The History List

By default, the history list contains the last 20 commands. You can change the number of commands saved using the *set history* command. For example, to save the last 30 commands in the history list:

```
1> set history 30
```

The syntax for this command is:

```
set history <number of commands in list>
```

## 14.2 The last Command

The *last* command returns the last *database* command you entered to the command buffer. Use this command to change settings with *dbView* commands and then re-execute the last SQL command. You could use the *history* command to do this, *last* is just faster.

For example, if you executed a command and then decided to time it, you could set the *timer* command to "on" and then recall the SQL command again using *last*:

```
1> select mission, scId
2> from missions
3> go
```

```
mission          scId
-----
Cassini          72
GLL              35
MO               91
VGR              0
```

(4 row(s) affected)

```
1> set timer on
```

```
1> last
```

```
1> select mission, scId
```

```
2> from missions
```

```
3> go
```

```
mission          scId
-----
Cassini          72
GLL              35
MO               91
VGR              0
```

(4 row(s) affected)

SQL statement took 0.03 seconds to execute.

Following the retrieval of the SQL command, you must type “go” to execute it. (Why doesn’t dbView execute the command immediately for you? Because, you might want to edit it first.)

## 15 Saving Commands And Data

As you work in dbView, you can save:

- Your session, that is, all of the commands, data and messages sent to the screen.
- Only the commands you enter.
- Only the data retrieved from a database.

Files are opened with one of three *open* commands. There is a corresponding *close* command for each *open* commands. If the file doesn't already exist, an *open* command creates a new file; otherwise it appends records to the end of the existing file. Files can be opened and closed at any time during a dbView session. Only one file of any type can be open at a time.

Start saving your dbView session to a file using the command:

```
open logFile <file name>
```

and to stop saving it, use:

```
close logFile
```

To save only your commands, use:

```
open commandFile <file name>
```

```
close commandFile
```

To save only data returned from the database, use:

```
open dataFile <file name>
```

```
close dataFile
```

In the following example, we'll open three files, one of each type. Then we'll execute some commands. And finally, we'll close the files.

The files are opened in the following sequence:

```
1> open logFile session.log
```

```
1> open dataFile session.dat
```

```
1> open commandFile session.sql
```

Once we're finished, we close the files with these command:

```
1> close commandFile
```

```
1> close dataFile
```

```
1> close logFile
```

The contents of the three files appear at the end of this section. The log file contains everything we typed after the file was opened, including any error messages. The command file contains the “go” terminator line for commands that require it. A command file can be reexecuted in dbView using the *script* command which is described in a later section. The data file contains just the results returned from the database.

## 15.1 The Show File Command

We use the *show file* command to see which files we have open. For example, if we open a command file and then issue the *show file* command, we see the following:

```
1> open commandFile example.cmd

1> show file
    commandFile: "example.cmd".
    dataFile not open.
    logFile not open.
```

## 15.2 The Directory Command

dbView works out of the directory in which it was invoked, so when you open a file, the file is opened in the working directory, unless you specify a path name with the file name. You can change the working directory with the command:

```
directory [<new path>]
```

If a new path is not supplied, the command shows you the current directory without changing it. Here's an example of how the command works:

```
1> directory
/usr/franklin

1> directory /usr/franklin/dbData

1> directory
/usr/franklin/dbData
```

Now, if we were to open a file without specifying a directory path, it would be opened in

```
/usr/franklin/dbData.
```

**Log File—session.log**

```
1> open dataFile session.dat
1> open commandFile session.sql
1> set format list
1> select mission, scId, description, objective, created
2> from missions
3> order by mission
4> go

Row 1>
      mission = Cassini
      scId =           72
      description = The Cassini Mission to Saturn
      objective = Saturn
      created = Mar 18 1993  8:51:56:850AM

Row 2>
      mission = GLL
      scId =           35
      description = The Galileo Mission to Jupiter
      objective = Jupiter
      created = Mar 18 1993  8:51:56:890AM

Row 3>
      mission = MO
      scId =           91
      description = The Mars Observer Mission
      objective = Mars
      created = Mar 18 1993  8:51:56:906AM

Row 4>
      mission = VGR
      scId =           0
      description = The Voyager Mission to the outer solar
system
      objective = NULL
      created = Mar 18 1993  8:51:56:940AM

(4 row(s) affected)
```

```
1> select mission
2> from mission
3> go
```

```
MDMS DBS WARNING milano::dbview Fri Mar 19 08:25:08 1993
(Db: jar, MsgNo: 208, Svr: 16, St: 1)
Invalid object name 'mission'.
```

```
1> set format table
1> set reals E
1> set singlePrecision 3
1> set doublePrecision 5
1> select number, name, lgtYrsFromSun, hrsPerRotation
2> from planets
3> where number >
4>     (select number
5>     from planets
6>     where name = 'Earth')
7> order by number
8> go
```

number	name	lgtYrsFromSun	hrsPerRotation
4	Mars	2.42086E-05	2.466E+01
5	Jupiter	8.23432E-05	1.002E+01
6	Saturn	1.51048E-04	1.027E+01
7	Uranus	3.03459E-04	1.080E+01
8	Neptune	4.75647E-04	1.590E+01
9	Pluto	6.62567E-04	NULL

```
(6 row(s) affected)
```

```
1> close commandFile
1> close dataFile
1> close logFile
```

### Command File—session.sql

```
set format list
select mission, scId, description, objective, created
from missions
order by mission
go
select mission
from mission
go
set format table
set reals E
set singlePrecision 3
set doublePrecision 5
select number, name, lgtYrsFromSun, hrsPerRotation
from planets
where number >
      (select number
       from planets
       where name = 'Earth')
order by number
go
close commandFile
```

**Data File—session.dat**

Row 1&gt;

```
mission = Cassini
scId =          72
description = The Cassini Mission to Saturn
objective = Saturn
created = Mar 18 1993  8:51:56:850AM
```

Row 2&gt;

```
mission = GLL
scId =          35
description = The Galileo Mission to Jupiter
objective = Jupiter
created = Mar 18 1993  8:51:56:890AM
```

Row 3&gt;

```
mission = MO
scId =          91
description = The Mars Observer Mission
objective = Mars
created = Mar 18 1993  8:51:56:906AM
```

Row 4&gt;

```
mission = VGR
scId =          0
description = The Voyager Mission to the outer solar
system
objective = NULL
created = Mar 18 1993  8:51:56:940AM
```

number	name	lgtYrsFromSun	hrsPerRotation
-----	-----	-----	-----
4	Mars	2.42086E-05	2.466E+01
5	Jupiter	8.23432E-05	1.002E+01
6	Saturn	1.51048E-04	1.027E+01
7	Uranus	3.03459E-04	1.080E+01
8	Neptune	4.75647E-04	1.590E+01
9	Pluto	6.62567E-04	NULL

## 16 Macro Commands

dbView has a macro command that enables you to assign a name to a set of commands to be executed. Macros are useful because:

1. They reduce the amount of typing necessary to run frequently executed commands.
2. They hide the complexity of the command sequence represented by the macro name.
3. Macros can have explanatory text associated with them; i.e., they carry their documentation.
4. Macros can have variables embedded in them; allowing you to modify a macro command at execution time.
5. Macros can contain more than one command, allowing a mini-script capability.
6. Macros may be executed repeatedly using the *repeat* command.

In the following sections we'll describe:

1. How to define, edit and execute a macro command.
2. How to document a macro command.
3. How to save a collection of macro commands in a file and include the macros in a later dbView session.
4. How to include variables in a macro command.

### 16.1 Defining And Running A Macro

Macros are defined using the macro command:

```
macro <name> [<history number>]
```

When you declare a macro, you give it a unique name. The name can include upper and lower case letters, numbers and the underscore character “\_”. Macro names can not be one of dbView's command words, like *set*, *show*, *open*, etc.; but they can be the same as objects defined in a database.

Here is a simple example definition of a macro named *GLL*:

```
1> macro GLL
---Command
1> # This command displays the mission acronym and
2> # spacecraft ID for the Galileo project
3>
4> set timer off
5> print
```

```

6> print "Galileo Mission Information"
7> select mission, scId, objective
8> from missions
9> where mission = "GLL"
10> go
11> done

```

Let's take a moment to look at the parts of the macro. The first line has the *macro* command followed by the name of the macro, in this case "GLL". Once we press return, dbView responds with the line

```
---Command
```

That's the signal that the macro specification should follow. A macro can contain a mixture of:

- dbView commands. Line 4
- Database commands. Lines 7–10
- References to previously defined macros.
- Blank lines to separate groups of commands within the macro. Line 3.
- Comment lines that begin with zero or more white spaces followed by the pound sign character "#". Lines 1–2.
- Print commands that display their contents when the macro is executed. Lines 5–6. Note that the string to be printed must be quoted with the exception of a line that contains no string. A *print* command alone just gives you a new line.

Once complete, we end a macro by typing the word "done" on a line by itself. dbView will set you back at the command line prompt at that point from which you can run the macro.

A few things to note:

- You can cancel a macro command like any other command using <control-c>.
- Database commands are terminated with "go" just as they are in dbView's command buffer. (There is an exception. If the macro ends with an SQL statement, we only need add "done". This will terminate both the query and the macro.)
- Comment and blank lines are for internal documentation. Macro print commands display their strings when the macro is executed. Use the print command without a following string to get a blank line.

To execute the macro we've just created, we type its name as a new command and press return:

```

1> GLL

Galileo Mission Information

```

```
mission          scId
-----
GLL              35
```

```
(1 row(s) affected)
```

Macros are `dbView` commands, so `dbView` executes them immediately. There is no need to follow the name with the “go” terminator when executing a macro command.

### 16.1.1 The Show Macro Command

Once you have defined some macros, you may want to know refer to them without executing them. You can use the `show macro` command for this purpose. The syntax for the command is:

```
show macro [macro name]
```

If you don’t include a macro name, the names of all your currently defined macros are displayed in alphabetical order. You can peruse the list to find the macro you’re looking for; or you may discover that you don’t have the exact macro you want; and you need to create a new one. For example:

```
1> show macro
    - GLL
    - planet
    - test
```

Note: If `verbose` is “on” and you issue the `show macro` command you’ll see the contents of all of your macros.

If you include a macro name with the command, the contents of the macro are displayed:

```
1> show macro GLL
---Command
# This command displays the mission acronym and
# spacecraft ID for the Galileo project

set timer off
print
print "Galileo Mission Information"
select mission, scId
from missions
where mission = "GLL"
go
```

Notice that `show macro` displayed the comment and blank lines we entered in the macro. This is for your internal documentation; these lines won’t appear when you execute the macro.

## 16.1.2 Editing A Macro

You can edit a macro while you're defining it or later on in a dbView session.

If, while you're defining a macro, you want to make changes on a line that's completed, just type *edit* as you would for any other command. dbView will display the text of the macro in your editor. The full text of the macro is displayed, including comments and blank lines.

```
1> macro mission
---Command
1> # This command displays the mission acronym and
2> # spacecraft ID for the Galileo project.
3> select mission, scId
4> form missions ← misspelled "from"
5> edit ← start editing the macro
```

Once you return from the editor, dbView redisplayes the command, leaving you with an entry point on a new command line.

Once you have created a macro you can still edit it using the *edit macro* command:

```
edit macro <macro name>
```

For example, to edit the macro we have just created, we would enter the command:

```
1> edit macro GLL
```

and the command would appear in the editor looking like this:

```
# This command displays the mission acronym and
# spacecraft ID for the Galileo project

set timer off
print
print "Galileo Mission Information"
select mission, scId
from missions
where mission = "GLL"
go
```

In the editor, we'll add a field to the select statement and return to dbView.

```
# This command displays the mission acronym and
# spacecraft ID for the Galileo project

set timer off
print
print "Galileo Mission Information"
select mission, scId, objective ← this was added
from missions
```

```
where mission = "GLL"  
go
```

Now when we execute the macro, we get this:

```
1> GLL  
  
Galileo Mission Information  
  
mission                scId      objective  
-----  
GLL                    35      Jupiter  
  
(1 row(s) affected)
```

### 16.1.3 Using Edit Macro To Create Another Macro

You can use an existing macro the basis for creating a new macro with the *edit macro* command. To do this, follow the name of the currently defined macro with a new macro name. The syntax is:

```
edit macro <macro name> [<new macro name>]
```

For example, if we wanted a macro that would write the results the GLL macro to a file, we could create a new macro GLLSave starting with the contents of GLL:

```
1> edit macro GLL GLLSave
```

The editor would show the contents of the GLL macro which we want to modify.

```
# This command displays the mission acronym and  
# spacecraft ID for the Galileo project  
  
set timer off  
print  
print "Galileo Mission Information"  
select mission, scId, objective  
from missions  
where mission = "GLL"  
go
```

We now make changes with the editor so the new macro looks like this:

```
# This command displays the mission acronym and  
# spacecraft ID for the Galileo project  
  
open dataFile gll.data
```

```

print
print "Galileo Mission Information"
select mission, scId, objective
from missions
where mission = "GLL"
go

```

```

close dataFile

```

Now when we issue the *show macro* command, we see that the new macro “GLLSave”. Our old macro, “GLL”, is still there, too.

```

1> show macro
    - GLL
    - GLLSave
    - planet
    - test

```

## 16.2 Using The History List In A Macro Command

You can begin a macro with a statement in the history list by placing the history number after the macro name in a macro definition. This is most useful for SQL commands. Create the command in dbView and run it. If it works, create your macro and reference the command with its history number. Then complete your macro.

```

1> history

---1
select mission, scId
from missions
where mission = "GLL"

---2
select mission, scId, objective
from missions
where mission = "GLL"

1> macro GLL 2 ← addition of the history number
1> select mission, scId, objective
2> from missions
3> where mission = "GLL"
4> go
5> done

```

In the example, we selected the second statement from the history list for inclusion in the

macro command. dbView sets us at the end of the command once it copied it in—we could continue to add to the command or edit it. Finally we complete the macro with the SQL command with the “go” terminator and the macro with the “done” terminator.

### 16.3 Using Macros To Redefine dbView Commands

All of dbView’s commands are spelled out completely. While this helps make them self-explanatory, it also increases the amount of typing needed to enter a command. You can use macros to redefine or shorten them.

Macros have an additional ability that we haven’t discussed yet. If you add text following a macro name, that text is concatenated to the end of the command. We’ll use this capability to design some generally useful macros that incorporated shortened dbView commands.

As an example, we’ll look at the *history* command. First we create a macro to redefine *history* to be just the letter h:

```
1> macro h
---Command
1> history
2> done
```

Now, when we type “h”, we get the history list:

```
1> h

---18
select mission, scId
from missions

---19
select name
from planets
order by number

---20
h

1>
```

Since macros concatenate any trailing characters on the command line to the macro, we can use the same macro to retrieve a command from the history list:

```
1> h 18
1> select mission, scId
2> from missions
3>
```

The macro expands to its definition plus the trailing characters to become:

```
history 18
```

Besides redefining `dbView` commands, we can use macros to create new ones. Putting the `escape` command in a macro is a good example.

Create a macro to list our current directory under Unix:

```
1> macro ls
---Command
1> escape ls -l
2> done
```

Now we can use the macro to list files:

```
1> ls *.macros
-rw-r--r-- 1 franklin          660 Mar 24 11:17 example.macros
```

(Notice that we are again using the feature that a macro will concatenate the string of characters following it on the command line.)

Now, create a macro to view a file under Unix:

```
1> macro vi
---Command
1> escape vi
2> done
```

We can use this macro to view the file we just found using the `ls` macro:

```
1> vi example.macros
```

Now let's incorporate an SQL statement. When we want to see all of the columns in a table we can use the command "Select \* from <some table>". We can capture this syntax in a macro called "all".

```
1> macro all
---Command
1> select * from
2> done
```

Now we can issue the command:

```
1> set format list
```

```
1> all missions
```

```
Row 1>
      id = 1
```

```
mission = Cassini
scId = 72
objective = Saturn
flying = 0
description = The Cassini Mission to Saturn
created = Jun 22 1993  5:58:10:403PM
```

Row 2>

```
id = 2
mission = GLL
scId = 35
objective = Jupiter
flying = 1
description = The Galileo Mission to Jupiter
created = Jun 22 1993  5:58:10:460PM
```

Row 3>

```
id = 3
mission = MO
scId = 91
objective = Mars
flying = 1
description = The Mars Observer Mission
created = Jun 22 1993  5:58:10:476PM
```

Row 4>

```
id = 4
mission = VGR
scId = 0
objective = NULL
flying = 1
description = The Voyager Mission to the outer solar
system
created = Jun 22 1993  5:58:10:493PM
```

(4 row(s) affected)

We could qualify our statement with an SQL WHERE clause also.

```
1> all missions where objective = 'Jupiter'
```

Row 1>

```
id = 2
mission = GLL
scId = 35
objective = Jupiter
flying = 1
```

```
description = The Galileo Mission to Jupiter
created = Jun 22 1993 5:58:10:460PM
```

```
(1 row(s) affected)
```

Notice that since “all” is a macro, we’ve found a way to execute the SQL statement immediately—we didn’t add a “go” terminator because this is a macro.

## 16.4 Using Variables In Macro Definitions

In the last section, we began to explore the idea of modifying a macro by concatenating text to the end of the macro command when we execute it. dbView has a more general mechanism for modifying macros at execution time—variable substitution.

In our “GLL” macro we got back information on the Galileo Mission. We can change that macro so we get back information on any specified mission. Let’s edit our previous macro so it includes a *local variable* in place of the value “GLL”. (A local variable applies to a single macro in which it’s defined. Later we’ll see that dbView also has global variables defined outside of macros, but usable by them.) But now, back to our example.

```
1> edit macro GLL
```

In the editor, we change “GLL” to “\$mission\_name”. Local variables always begin with a dollar sign followed by a string that becomes a prompt when you execute the macro. The prompt string can contain lower and upper case letters, numbers and the underscore character “\_”; and it can be of any length allowed on your system.

We also change the text we display when the macro is executed to explain to the user what is expected.

```
# This command displays the mission acronym and
# spacecraft ID for the Galileo project

set timer off
print
print "Display the mission name (acronym), spacecraft ID and"
print "primary objective (planet) for the mission name"
print "supplied at the macro's prompt."
print
select mission, scId, objective
from missions
where mission = "$mission_name" ← the local variable
go
```

Since we’ve now made a general macro that will return information for any mission, we should rename it. To do this we use the rename macro command. The syntax is:

```
rename macro <current name> <new name>
```

For our example, we'll do this:

```
1> rename macro GLL mission
```

Now when we execute the updated and renamed macro, we see:

```
1> mission
```

Display the mission name (acronym), spacecraft ID and primary objective (planet) for the mission name supplied at the macro's prompt.

```
mission_name []: Cassini
```

```
mission          scId          objective
-----
Cassini          72    Saturn
```

```
(1 row(s) affected)
```

The local variable we defined becomes a prompt that is displayed when we execute the macro. We responded to the prompt by entering the name "Cassini". dbView substitutes this value for the local variable so the SQL command sent to the database server becomes:

```
select mission, scId, objective
from missions
where mission = "Cassini"
```

Notice that we includes the local variable within the string quotation marks required by the SQL language. Then, at the prompt, you did not have to provide a quoted string. If we had not enclosed the variable with quotation marks, we would have need to supply them.

#### 16.4.1 Local Variables And Default Values

When the macro was executed, you may have noticed that the prompt was followed by a pair of square brackets, []. If you now executed the macro again, you last response to the prompt becomes the default value and appears in the square brackets.

```
1> mission
```

Display the mission name (acronym), spacecraft ID and primary objective (planet) for the mission name supplied at the macro's prompt.

```
select mission, scId, objective
```

```
mission_name [Cassini]: GLL
```

```

mission          scId          objective
-----
GLL              35  Jupiter

(1 row(s) affected)

```

We can accept a default value at a prompt by typing <return>, or we can enter a new value as we did in the above example.

There are two types of default values associated with local variables: *session* and *persistent*.

#### *Session Defaults*

Whenever you enter a new value for a local variable, it is saved; and the next time you execute the macro it is displayed as the default value. The default values last for the life of the dbView session only.

#### *Persistent Defaults*

A persistent default value acts like a session default, but it is defined in the macro along with the local variable; and it persists across dbView sessions, i.e., it appears as the default whenever the macro is first used in a session. (We'll discuss how macros are saved to files shortly.)

We can change the *mission* macro to include a persistent default value. The value is enclosed in square brackets—the syntax for a default value—immediately following the declaration of the local variable.

The persistent default value—we've made it "Cassini"—and its local variable are in bold type.

```

1> show macro mission
---Command
print
print "Display the mission name (acronym), spacecraft ID and"
print "primary objective (planet) for the mission name"
print "supplied at the macro's prompt."
print
select mission, scId, objective
from missions
where mission = "$mission_name[Cassini]"

```

Notice that the persistent default value is included with the local variable *within the quotation marks* used for string values in an SQL SELECT statement.

We've included this macro in our default macro file using the *save macro* command. Now when we execute the macro in a new dbView session, the default for *mission\_name* is already set to "Cassini".

```

dbView, version 1.4, (dblib, milib), 28 Nov 1994
Copyright 1993, The Jet Propulsion Laboratory. All rights
reserved.

```

```
userName [franklin]:  
password:  
server [CATALOGDBS]:  
database [catalog]:
```

```
1> mission
```

Display the mission name (acronym), spacecraft ID and primary objective (planet) for the mission name supplied at the macro's prompt.

```
mission_name [Cassini]: ← persistent default value
```

```
mission          scId          objective  
-----  
Cassini          72    Saturn
```

```
(1 row(s) affected)
```

If we had entered a different value at the prompt, “GLL” for example, the default would change to “GLL” for the rests of this session, or until we again changed the value. In subsequent sessions, however, the default would again be “Cassini”.

#### 16.4.2 The Special Local Variable \$password

If you include the local variable \$password in a macro, dbView handles it in a special way:

- When you type in a value at the prompt, the value is not echoed to your screen.
- There is no default value for the local variable, you must always type it in at the prompt.
- The variable is special in that dbView reserves the name for special handling.

\$password—all lower case letters—is used when you are executing a command from within dbView that requires a database server password. In the following example, we create and run a macro that will load the *missions* table data into a database using the Sybase *bcp* utility. Since *bcp* requires a password, we include the variable in the macro. If we were to “hard code” the password value in the macro and then save the macro in a file (see “*Including, Saving, And Replacing Macro Definitions*” on page 87), we would be violating security rules.

The macro definition is shown below. We’ve highlighted the \$password variable in bold type so you can see it in the command.

```
1> show macro bcpCommand  
---Command  
print  
print "Loading mission data into the 'example' database"
```

```

print "using bcp."
print "--Enter the password to be used by "bcp" at the prompt."
print
escape bcp example.dbo.example in mission.dat -c -U sa -P
$password -S CATALOGDBS

```

Now when we run the macro, we see this:

```

1> bcpCommand

Loading mission data into the 'example' database
using bcp.
  --Enter the password to be used by "bcp" at the prompt.

password: ← value not echoed

1>

```

Note the word 'example' is in single quotes in the *print* string. You can't use double quotes within a *print* string.

(We'll see this macro used in a broader context when we look at scripts—files that can contain dbView command to be executed—see "*Copying A Database Tables Contents*" on page 127.)

### 16.4.3 Summarizing Macro Local Variable Rules

1. Local variable names can only appear in macro commands.
2. Variable names begin with a dollar sign.
3. Variable names can be of any length. The names can include letters of the alphabet, numbers (integers), and the underscore character, "\_".
4. dbView performs string substitution on variable names, replacing the name with the string supplied by you. Therefore, variables can appear anywhere in a macro command.
5. When you execute a macro, dbView uses the macro name as a prompt. Following the prompt, dbView displays a pair of square brackets, [ ]. These brackets will contain the default value for the prompt. The default value is always the last value supplied for the variable. In the case of persistent default values, the default value always appears.
6. The value supplied at a local variable prompt can be any string of characters. dbView accepts everything you type on the remainder of the prompt line. That means that strings containing spaces are accepted; for example:

```

1> theDate
theDate []: April 14, 1993

```

The string, "April 14, 1993", is the value assigned to the local variable *\$theDate*.

## 16.5 Repeated Execution Of Macros

If you precede a macro with the command word *repeat*, the macro will be repeatedly executed until you type <control-c>. This command is particularly useful when you want to insert or update data in a table. For example, let's create a new table containing a user ID and a name, and then add some records to the table using a macro that contains an insert statement. First we create the table, which we'll call *example*:

```
1> create table example (  
2>     id      int,  
3>     name    varchar(15)  
4> )  
5> go
```

Next, we'll create the macro that inserts the data:

```
1> macro doExample  
---Command  
1> insert into example (id, name)  
2> values ($id, '$name')  
3> go  
4> done
```

Finally, we execute the macro, preceding the macro's name with the *repeat* command so it will execute until interrupted:

```
1> repeat doExample  
id []: 1  
  
name []: Franklin  
  
(1 row(s) affected)  
  
id [1]: 2  
  
name [Franklin]: Washington  
  
(1 row(s) affected)  
  
id [2]: 3  
  
name [Washington]: Jefferson  
  
(1 row(s) affected)  
  
id [3]: ^C ← this gets us out of the loop  
1>
```

To finish off the example, we retrieve the information we just stored in the *example* table:

```
1> select * from example
2> go
```

```
id          name
-----
          1 Franklin
          2 Washington
          3 Jefferson
```

```
(3 row(s) affected)
```

## 16.6 Including, Saving, And Replacing Macro Definitions

### 16.6.1 Saving Macros To A File

Once we've defined a macro, we'd like to save it so we can use it in a later dbView session. To do this we use the *save macro* command:

```
save macro <file name> <macro name>
```

If the file name is new, dbView creates it and writes the macro to the file. If the file exists, dbView writes the macro to the file. If the macro is already present in the file, the old definition is removed and the new one placed in the file.

For example, to save the mission macro to the file `example.macros`, we do this:

```
1> save macro example.macros mission
```

### 16.6.2 Removing A Macro Command

If you want to remove a macro definition, use the command:

```
remove macro <macro name> [<macro file name>]
```

This command removes the macro definition from the dbView session. If you include the name of a macro file in the command, the macro's definition is removed from the file also. For example:

```
1> show macro GLL
---Command
# This command displays the mission acronym and
# spacecraft ID for the Galileo project

set timer off
print
```

```
print "Galileo Mission Information"
select mission, scId, objective
from missions
where mission = "$mission_name"
go

1> remove macro GLL

1> show macro GLL
Unknown macro name "GLL".
```

After executing the *remove macro* command, you can no longer reference the macro. If you do, dbView will identify it as a command that should be sent to the database server, and you will get an error message.

If we include a file name, the macro is removed from the file as well. For example:

```
1> directory /usr/franklin
1> remove macro GLL example.macros
```

The remove macro command removes a macro from a file even if the macro is no longer defined within dbView.

We could have included a directory path in the last example if dbView was not currently “looking at” the directory where the macro file exists. This would change the example to look like this:

```
1> remove macro GLL /usr/franklin/example.macros
```

### 16.6.3 Exiting From dbView Once You’ve Made Changes To Macros

Once you’ve made changes to macros, you probably want to save them before exiting from dbView. If you forget, dbView will prompt you with the following message:

```
1> exit
You have made changes to macro definitions. They can be saved
using the "save macro" command. The following are the changed
macros:
```

```
GLL
```

```
exit anyway? { y | n } [n]: ← typing <return> accepts the default-“no”
```

```
1> save macro example.macros GLL
```

dbView will prompt you before exiting if you’ve used any of the following command to create or change a macro or global variable definition:

```
global
```

```
macro
edit macro
rename macro
```

To exit without saving macros, you could answer “y” (yes) to the prompt in the example above, or you could add a parameter to the exit command so it looks like this:

```
1> exit ignore set global macro
```

In this case, you force dbView to exit without checking to see if you’ve created macros. This form of the exit command could be useful in a script file that executes without your intervention within dbView.

#### 16.6.4 Including A Macro File

Once a set of macros is saved to a file, we can include the macros in our dbView session by using the command:

```
include macro <file name>
```

For example:

```
1> include macro example.macros
```

or if the macro is not in dbView’s current directory

```
1> include macro /usr/franklin/example.macros
```

You can use the *include macro* command to include macros from as many files as you like. Macro loading is cumulative, which means:

1. As new macros are loaded, the old macro definitions remain defined.
2. If a new macro has the same name as an old one, the new one replaces the old one unless the old macro has been edited and not yet saved in your current dbView session.

#### 16.6.5 The Default Macro File

You can also load one or more macro files when you start dbView by specifying the file paths as default macro files with the command:

```
set defaultMacroFile <file specification> ... [<file specification>]
```

The file specification includes the entire directory path along with the file name. If you include more than one default macro file, place one or more spaces between the file specifications. All of the files must appear on a single line, even if it means that the text wraps to the next line, because dbView executes the command as soon as you press <return>.

For example, if we wanted to include our standard set of macros plus those used just for this tutorial, we would include two files containing macro definitions:

```
1> set defaultMacroFile /usr/franklin/standard.macros /usr/
franklin/example.macros ← a single wrapped line
```

Now, the next time you start dbView, the macros in these files are loaded immediately, ready for use. You can change the default macro specification at any time, but it wouldn't take effect—the macros won't be loaded automatically—until your next dbView session starts.

When you set the path name of a default macro file, dbView checks to determine if it can find the file. If it can't, dbView displays an error message. For example:

```
1> set defaultMacroFile /usr/junk

WARNING:  Unable to open defaultMacroFile:
/usr/junk.
Your default macros cannot be loaded at startup.

MDMS SYSTEM WARNING candid::dbView Fri Apr 23 11:07:53 1994
(2) No such file or directory
```

Also, if some change was made to your configuration that prevents dbView from reading the default macro file at start-up time, you will receive error messages to that effect that dbView can't load the macros properly:

```
% dbView

dbView, version 1.4, (dblib, milib), 28 Nov 1994
Copyright 1993, The Jet Propulsion Laboratory. All rights
reserved.

WARNING: Could not open default macro file:
/usr/franklin/example.macros.

MDMS SYSTEM WARNING candid::dbView Fri Apr 23 11:03:54 1994
(2) No such file or directory
```

### 16.6.6 Replacing A Set Of Macros

If you are loading more than one macro file and you don't want the macro definitions to be cumulative, use the command:

```
replace macro [<file name>]1
```

In this case the macros from the new file are loaded once all of the old macros are removed from dbView. Here's an example in which the macros in the file `example.macros` are replaced by those in the file `new.macros`.

---

1 dbView will not execute this command if you have created new macros and not yet saved them. It protects you from destroying your own work.

```
1> include macro example.macros
```

```
1> replace macros new.macros
```

### **16.6.7      Sharing Macro Files**

dbView stores macros in ASCII formatted files so that you can use them on different types of machines. This also allows you to send the files over a network to other people who may have different types of hardware.

## 17 Global Variables

We have already introduced macro variables. However, we left something out at that time to simplify the presentation. It's also possible to define *global variables* in macros—the variables we described earlier are referred to as *local variables*.

Global variables are used to define a value that is frequently used across more than one macro. They differ from local variables in the following ways:

1. A global variable is defined within dbView using the *global* command and the global variables value is referenced within a macro or another global by preceding the variable name with two dollar signs, e.g., \$\$planet. (Local variables are preceded by a single dollar sign, e.g., \$planet.)
2. Global variables are independent of particular macros and can be reference by any macro during a dbView session.
3. Once defined and included in a macro, a global variable's value is automatically substituted in the macro at the time the macro is executed. dbView does not prompt for the value of a global variable.
4. Global variables are saved, loaded and replaced, along with macros, using the commands *save macro*, *include macro* and *replace macro*. This means that a set of global variables is saved to a file along with macros.

### 17.1 The global Command

A global variable is defined with the *global* command:

```
global <name> <global variable> [<value>] | <value>
```

The global command is a single line command, and therefore executed immediately when you press the <return> key.

For example:

```
1> global object mission
```

The global variable *object* is assigned the value “mission”.

In the next example,

```
1> global fileName $$object.tmp
```

the global variable *fileName* is assigned the value of the global variable *object* concatenated with the value “.tmp”. Since dbView won't be able to evaluate the global variable *object* used as part of the *fileName* global variable, we use the nomenclature \$\$*object* which “stands in” for the value until its actually used.

A global variable like *object* is the name of variable. When the dollar signs are placed in front

of the global variable `$$object`, it refers to the *value* of the variable. So, when you define a global variable you assign a value to the name:

```
1> global object missions
```

and when you use a global variable, you reference the value by preceding the variables name with the dollar signs:

```
1> global fileName $$object.tmp
```

### 17.1.1 Seeing Global Variable Assignments

You can see the definition of global variables using the command:

```
show global [<variable>]
```

For example:

```
1> show global
    - fileName = $$object.tmp
    - object = missions
```

Or, if you only want to see a specific one:

```
1> show global fileName
    - fileName = $$target.tmp
```

### 17.1.2 The Expand Global Command

Notice that the definition of `fileName` is “`$$target.tmp`”. If you want to fully expand the global variable so that you can see what it would evaluate to at run time, use either of the commands:

```
expand global <global variable>
```

For example:

```
1> expand global fileName
missions.tmp
```

This command is useful for debugging complicated global variables.

You can also expand a macro to see the run-time value of global variables referenced in the macro—see “*The Expand Macro Command*” on page 94.

## 17.2 Referencing Global Variables In Macros

Once defined, you can reference a global variable in any number of macros. In the following examples we define a select statement, and commands to open a file and to delete a file. All three macros can be changed simply by changing the value assigned to `object`. (The two macros

that manipulate files are defined indirectly through the *fileName* global variable which gets its value from *object*.)

The example assumes that we've already defined the global variables *object* and *fileName*.

1. Create an SQL SELECT statement that will retrieve all of the data from the table referenced by the global variable *object*.

```
1> macro showTable
---Command
1> print "Select all of the data from the table whose"
2> print "name is assigned to the global variable"
3> print "`object'."
4> select *
5> from $$object
6> go
7> done
```

2. Open a data file using the value of *fileName*, the global variable that includes a reference to the other global variable *object*.

```
1> macro openFile
---Command
1> open dataFile $$fileName
2> done
```

3. Define a macro to remove the file opened by the previous macro. (This macro uses the *dbView escape* command to execute a command at the operating system level. The example removes the file using the Unix *rm* command.)

```
1> macro removeFile
---Command
1> escape rm $$fileName
2> done
```

We'll show a practical application that moves a table from one database to another using these global variables and macros when we discuss *scripts*—files that contain a collection of *dbView* commands, see "*Copying A Database Tables Contents*" on page 127.

### 17.2.1 The Expand Macro Command

Once global variables are defined and reference in a macro, you can see the run-time value of the macro using the command:

```
expand macro <macro name>
```

As an example, we'll define our global variables again, then look at the macro "openFile" using both *show macro* and *expand macro* to see the difference.

```

1> global fileName $$object.tmp

1> global object missions

1> show macro openFile
---Command
open dataFile $$fileName

1> expand macro openFile
open dataFile missions.tmp ← the value at run-time

```

*Show macro* gives the definition of the macro, while *expand macro* shows its current evaluated state. *Expand macro* is most useful for debugging a macro that references global variables.

### 17.3 The Remove Global Command

Just as there is a *remove macro* command to remove a macro definition, so there is an *remove global* command to remove a global variable definition. The syntax is:

```
remove global <name> [<macro file name>]
```

Global variables are stored along with the macros that use them, so the file name used with the command is a *macro* file name.

For example:

```
1> remove global object example.macros
```

or

```
1> remove global object /usr/franklin/example.macros
```

Note: Once we've removed the global variable *object*, any macro that reference the global variable will not execute. See the next section.

### 17.4 Undefined Global Variables In Macros

If you execute a macro containing an undefined global variable, you'll get an error message like this:

```
1> set verbose on
```

```
1> showTable
```

```
Select all of the data from the table whose name is
assigned to the global variable "object".
```

```
Global variable 'object' not defined. Macro command
```

*aborted.*

## 17.5 Global Variables And Macro Files

Since global variables are associated with macros, when you have a macro, the global variables reference in the macro are also save to the same file. The global variables being saved replace any existing definitions in the macro file.

When you execute an *include macro* command, global variables in the file are also brought in and defined in your current session.

When you execute a *replace macro* command, it removes all of the current global definitions in your dbView session.

If you create a global variable that is not associated with a macro, there's no way to save it because global variables are saved when the macro in which they are referenced is saved. dbView will warn you about such variables when you exit. At that time, you should either associate the variable with a macro and save the macro, or you should exit anyway.

## 18 Finding Out About Database Objects<sup>1</sup>

### 18.1 Sybase Database Objects

The *show db* command is used to display information about objects in the current database including:

- tables—both user and system defined
- views
- stored procedures
- triggers
- defaults
- rules

The syntax for the command is:

```
show db [<database object name>]
```

If you don't include the name of an object in the database, dbView returns the list of all objects in the current database organized by type, i.e. default, stored procedures, rules, *etc.* For example:

```
1> show db

defaults :
    default_timeOfDay

stored procedures :
    showMissionObjective          showMissions
    showPlanets

rules :
    rule_zeroOne

system tables :
    sysalternates                syscolumns
    syscomments                  sysdepends
    sysgams                       sysindexes
    syskeys                       syslogs
    sysobjects                    sysprocedures
    sysprotects                   syssegments
    systypes                       sysusermessages
```

<sup>1</sup> This version does not support *show db* for Illustra databases.

```
sysusers

triggers :
  missionsDelTrig           missionsInsUpdTrig

user tables :
  missions                   planets

views :
  missionObjective
```

You can then use the list to get more specific information about a particular object, like the *missions* table. The information returned on a specific object depends on the object's type. In the following sections we'll illustrate how *show db* reacts for different types of database objects.

### 18.1.1 Table Information

Suppose we include the object name “missions” with the *show db* command. *dbView* would then display the following:

```
1> show db missions

Table: missions
Owner: dbo

  colid  column                data type      length  nulls?
-----  -
1       id                    id             4       No
2       mission              name           30      No
3       scId                 id             4       No
4       objective            name           30      Yes
5       flying               flag           1       No
6       description           description     255     Yes
7       created              timeOfDay      8       No
```

The first line tells us that “missions” is a table and that its owner is “dbo” (database owner). Next, *dbView* lists information about the columns—or fields—in the table. The first column is the order of the column in the table. The second column is the name of the column in the table. The last three columns give us information about the table column's data type, including the data type name, length in bytes and whether or not the column will accept NULL values.

If you include the *verbose* option, you see the same display just described. Following that, you'll see additional table information. This information is only displayed if it exists. In other

words, if there are no indexes on the table, you won't see the "Indexes" title. dbView returns the following additional information on tables after the command *set verbose on* has been executed.

1. **Indexes**—The name of the index is given followed by the names and order of the columns included in the index. A second line describes the characteristics of the index.
2. **Keys**—Foreign and common key definitions. Key definitions are important because they show you how one table is related to another.
3. **Capabilities**—This is the list of actions you can carry-out on the object. For example, if you have the "select" capability on the table, you can query it. If you are the owner of the table, Capabilities returns the word "all", meaning that you have all capabilities on the table. If you have no capabilities, the word "none" is returned.
4. **Defaults**—The columns in the table having default values associated with them are listed along with the name of the default. To see the definition of the default, use the *show db* command with the name of the default.
5. **Rules**—The columns in the table having rules associated with them are listed along with the name of the rule. To see the definition of the rule, use the *show db* command with the name of the rule.
6. **Triggers**—Tables can have insert, update and delete triggers associated with them. If any of these exist, their names are listed. To see the definition of a trigger, use the *show db* command with the name of the trigger.
7. **Related Stored Procedures**—Any stored procedures that reference the table are listed next. To get more information about a procedure, use the *show db* command with the name of the procedure.

Here we repeat the *show db* command for the *missions* table, but this time we'll execute the command in dbView's verbose mode.

```
1> set verbose on
```

```
1> show db missions
```

```
Table: missions
```

```
Owner: dbo
```

colid	column	data type	length	nulls?
1	id	id	4	No
2	mission	name	30	No
3	scId	id	4	No
4	objective	name	30	Yes

colid	column	data type	length	nulls?
-----	-----	-----	-----	-----
5	flying	flag	1	No
6	description	description	255	Yes
7	created	timeOfDay	8	No

**Indexes:**

- missionPK1 on column(s) mission  
clustered, unique located on default
- missionFK1 on column(s) objective  
nonclustered located on default

**Keys:**<sup>1</sup>

- foreign: mission.objective -> planets.name
- common: missions.objective <-> planets.name

**Capabilities:**

- select
- insert
- update

**Defaults:**

- default\_timeOfDay on column created

**Rules:**

- rule\_zeroOne on column flying

**Triggers:**

- insert: missionsInsUpdTrig
- update: missionsInsUpdTrig
- delete: missionsDelTrig

**Related Stored Procedures:**

- showMissionObjective
- showMissions

### 18.1.2 View Information

SQL views are virtual tables that offer a view of data contained in one or more physical tables within the database. The examples in this section use a view named *missionObjective* that contains columns from the *missions* and *planets* tables. The columns are brought together in the view using the SQL join operation, but when you query the columns, they appear to be in one table, which is actually the view, *missionObjective*.

---

<sup>1</sup> Because of the limited scope of our examples, both foreign and common keys have the same definitions.

If verbose mode is not active, we see:

```
1> show db missionObjective
```

```
View: missionObjective
```

```
Owner: dbo
```

colid	column	data type	length	nulls?
1	mission	name	30	No
2	scId	id	4	No
3	name	name	30	Yes
4	lgtYrsFromSun	float	8	Yes
5	hrsPerRotation	real	4	Yes
6	yrsPerRev	real	4	Yes

When we're in verbose mode, we get additional information, including:

1. **Capabilities**—This is the list of actions you can carry-out on the view. For example, if you have the “select” capability on the view, you can query it. If you are the owner of the view, the capabilities returns the word “all”, meaning that you have all capabilities on the view. If you have no capabilities, the word “none” is returned.
2. **Associated Tables**—The names of the tables referenced by the view.
3. **Associated Procedures**—The names of any stored procedures that reference the view.
4. **Description**—The SQL statement that defines the view, along with any header information associated with the view's definition.

For example:

```
1> set verbose on
```

```
1> show db missionObjective
```

```
View: missionObjective
```

```
Owner: dbo
```

colid	column	data type	length	nulls?
1	mission	name	30	No
2	scId	id	4	No

colid	column	data type	length	nulls?
-----	-----	-----	-----	-----
3	name	name	30	Yes
4	lgtYrsFromSun	float	8	Yes
5	hrsPerRotation	real	4	Yes
6	yrsPerRev	real	4	Yes

Capabilities:

- all<sup>1</sup>

Related Tables:

- missions

- planets

Related Stored Procedures

- showMissionObjective

```
/*
** VIEW
**   missionObjective
**
** FUNCTION
**   Joins information in the missions and objectives tables.
*/
create view missionObjective (mission, spacecraft, planet,
    lgtYrsFromSun, hrsPerRotation, yrsPerRev)
as
    select mission, scId, name, lgtYrsFromSun,
        hrsPerRotation, yrsPerRev
    from missions, planets
    where missions.objective *= planets.name
```

### 18.1.3 Stored Procedure Information

When dbView displays information about a stored procedure, it states the name of the procedure and owner. This is followed by a table listing of the specification for each parameter accepted by the stored procedure. For example:

```
1> show db showMissions

Procedure: showMissions
```

---

1 For this example, we have assumed that you own the view; you therefore have all capabilities on it.

```
Owner: dbo
```

colid	name	data type	length
1	@missionName	name	30

In verbose mode, dbView includes the following additional information:

1. **Capabilities**—For a stored procedure, the capability you have is either “execute” or “none”.
2. **Text**—This is the text of the stored procedure, including any header information associated with the procedure’s definition. The header information often gives you additional information about the syntax of the procedure and a description of its function. For example:

```
1> set verbose on
```

```
1> show db showMissions
```

```
Procedure: showMissions
```

```
Owner: dbo
```

colid	name	data type	length
1	@missionName	name	30

```
Capabilities:
```

```
- execute
```

```
/*
```

```
** PROCEDURE
```

```
** showMissions [missionName]
```

```
**
```

```
** FUNCTION
```

```
** Show mission information. If name is supplied,
```

```
** information for that mission.
```

```
*/
```

```
create procedure showMissions
```

```
@missionName name = null
```

```
as
```

```
begin
```

```
print " MISSION INFORMATION"
```

```
print " "
```

```
if @missionName = null
```

```
begin
```

```
        select mission, scId, objective, description, created
        from missions
        order by mission
    end
    else
    begin
        select mission, scId, objective, description, created
        from missions
        where mission = @missionName
    end
end
```

#### 18.1.4 Trigger, Default And Rule Information

Triggers, defaults and rules are integrity objects associated with objects in the database. You find their names when you use the *show db* command on a table. If you then supply the name of a trigger, default or rule with the *show db* command, dbView displays the complete text used to implement the integrity object, along with any header information that explains its use. In the following example, we examine the rule “rule\_zeroOne”; Triggers and defaults would display similar information.

Setting verbose to “on” has no effect for triggers, defaults or rules.

```
1> db help rule_zeroOne

Rule: rule_zeroOne
Owner: dbo

/*
** RULE
**   rule_zeroOne
**
** FUNCTION
**   A boolean function. Value must be 0 or 1. The pair can
**   signify binary sets like {no, yes}, {off, on},
**   {not OK, OK}, {stop, go}, etc. The values can also be used
**   for logical tests in programming languages like C.
*/
create rule rule_zeroOne as @value in (0, 1)
```

## 18.2 Illustra Database Objects

Planned but not currently implemented.

## 19 Defining and Running Reports<sup>1</sup>

You can use dbView to define and run simple database reports. By simple report we mean one that contains information returned by a single query statement—either an SQL SELECT statement, a stored procedure or a dbView macro that returns a single set of result rows. You create and run reports with the following commands:

*report <file name> [<history number>]*

Creates a report. The report specification is saved in the file whose name you supply. If you include a history number, the report will be generated using this command in the dbView history list.

*edit report < file name>*

Once you've created a report, you can alter its contents with this command.

*run report < file name> [<report name>]*

This command runs the report using the specification found in the file whose name you supply. If you also supply a report name, the report is written to a file by that name.

### 19.1 Sample Reports

We'll use a couple of examples to illustrate how reports are defined and generated. For both of the examples, we first show the report, the specification used to generate the report and finally we'll discuss the reporting topics that the example introduces.

#### 19.1.1 The Report mission.rpt

This report was generated using the query:

```
select mission, id, scId, description, created
from missions
order by mission
```

---

<sup>1</sup> The report specification design borrows ideas from the *Perl* language created by Larry Wall.

Running the report results in the following two page report. In the first record of the report, we have placed the values returned by the SELECT statement in bold type.

MISSION REPORT	
<b>Cassini Mission</b> IDs - Mission: 1	<b>Spacecraft: 72</b>
The Cassini Mission to Saturn	
Record Creation Date: Jul 28 1993 9:08:01:730AM	
<b>GLL Mission</b> IDs - Mission: 2	<b>Spacecraft: 35</b>
The Galileo Mission to Jupiter	
Record Creation Date: Jul 28 1993 9:08:01:790AM	
Wed Jul 28 09:45:19 1993	Page: 1

*Table 2: First Page Of The Report*

<b>MO Mission</b> IDs - Mission: 3	<b>Spacecraft: 91</b>
The Mars Observer Mission	
Record Creation Date: Jul 28 1993 9:08:01:810AM	
<b>VGR Mission</b> IDs - Mission: 4	<b>Spacecraft: 0</b>
The Voyager Mission to the outer solar system	
Record Creation Date: Jul 28 1993 9:08:01:840AM	
Wed Jul 28 09:45:19 1993	Page: 2

*Table 3: Second Page Of The Report*





message when you run the report. For example:

```
1> run report example.rpt
      Report specified 1 field format(s).
      There were 2 specified in your database statement.
```

And if there are more format specifications than query attributes, the error message will read like this:

```
1> run report example.rpt
      Report specified 3 field format(s).
      There were 2 specified in your database statement.
```

The last two format specifications deserves some special notice. The formats are not large enough to contain the entire value for the fields *description* and *created* in the SELECT statement, so the text is wrapped automatically within the specification's limits. Normally, wrapping occurs on word boundaries, but if the word is too long to fit in the format field specification, dbView breaks the text in the middle of a word. Wrapping only applies to the left justified field specification—the one you would use normally for text strings.

Also notice that the last field specification accepts values from the table attribute, *created*, which is a *date* data type in the database. dbView automatically makes the conversion to a character string.

## 5. Footer

The footer contains information that appears at the end of each page. Like all sections of the report specification, if the footer is empty—contains only a line with the “go” terminator on it—the footer is not used in the report. It can include boiler plate and any of the following special variables:

- *\$date*—The current date that looks like:  
Wed Jul 28, 93
- *\$longdate*—The current data and time that looks like:  
Wed Jul 28 09:45:19 1993
- *\$page*—The current page number of the report.

You can mix special variables and “boiler plate”. In the example we used:

```
Page: $page
```

for the page numbers in the finished report.

## 6. Page length [66]

This is the length of a report page. By default, it is 66 lines. For the example, we set the page length to 18 lines so the report would generate more than one page.

## 7. Mail to

This is the list of people or groups to which the report is automatically sent when you generate the report. If your report has a mailing list, but you don't want mail sent when you execute the report, use the *set* command:

```
1> set mailReport off
```

## 8. Printers

This is the list of printers—one specification per line—to which the report is automatically sent when it's executed. In the example, we have used a standard Unix print specification, but you can use any specification that works for your operating system. You could also use more sophisticated printer commands, *enscript* on SUN computers, for example.

If you don't want the printer list used for a particular report, use the *set* command:

```
1> set printReport off
```

Now that we have completed the specification, look at it as a whole. Notice that the text and field specification entered line up the way they will appear in the final report. dbView's report function offers something close to “what you see is what you get” kind of formatting.

To run the report, we enter the command:

```
1> run report mission.rpt
```

In this example we've seen:

- The sections of dbView's report.
- How a report is formatted using the different sections.
- The use of field specifications and “boiler plate” to create a multiline report.
- The wrap facility of left justified field specifications.
- The use of the special footer variables \$date and \$page.
- Page specifications.
- Mail list and printer specifications.

### 19.1.2 The Report planets.rpt

The next example uses the *planets* table to illustrate some advanced reporting features. As in the last example, we'll first run the report and then show the specification that created the it.

This report will use a *macro* command in place of an SQL statement. Since we want the macro's comment to appear at the time we run the report, verbose is set to “on”.

Also, we'll retrieve numbers into a right justified character field as well as into numeric format fields that used the “#” character to designate the placement of integer and real numbers

returned by a query.

We control the precision and numeric style of numbers returned in character fields with the set commands the *set doublePrecision*, *set singlePrecision* and *set reals*. Before we run the report, we'll make settings used by the report.

```
1> set verbose on

1> set doublePrecision 4

1> set reals g

1> run report planets.rpt
```

The macro 'planetList' retrieves information about the planets supplied as a list at the prompt. The list should be formatted like the following example:

```
'Jupiter', 'Saturn', 'Venus'
```

```
planets ['Jupiter', 'Saturn', 'Earth', 'Venus']:
```

Since the report contains a macro and *verbose* is set to "on", dbView displays the macro's comment when the report is run. In our example, we have accepted the current session's default value for the prompt *planets*. Before we look at the report, we'll show the macro that was used as part of this report:

```
1> set verbose on
1> show macro planetList

- planetList
---Command
print "The macro 'planetList' retrieves information about the"
print "planets supplied as a list at the prompt. The list"
print "should be formatted like the following example:"
print "      'Jupiter', 'Saturn', 'Venus'"

select number, name, lgtYrsFromSun, hrsPerRotation, yrsPerRev
from planets
where name in ($planets)
order by number
go
```

On the next page we'll show the output of the macro and then that of the report.

The results retrieved using the macro *planetList*:

number	name	lgtYrsFromSun	hrsPerRotation	yrsPerRev
2	Venus	1.142e-05	NULL	0.616
3	Earth	1.585e-05	24.0	1.00
5	Jupiter	8.234e-05	10.0	12.3
6	Saturn	0.0001510	10.3	29.0

(4 row(s) affected)

The same results retrieved using the report *planets* that calls the macro *planetList*

Planet List				
Num.	Name	Light Years From The Sun	Earth Hours Per Rotation	Years Per Revolution
2	Venus	1.142e-05	NULL	0.62
3	Earth	1.585e-05	24.000	1.00
5	Jupiter	8.234e-05	10.021	12.34
6	Saturn	0.0001510	10.273	29.02

Mon Jun 07 16:07:19 1993 -1-

*Table 4: The Formatted Report*

The specification for the report looks like this:

```

---Database command:
planetList
go
---Report title:
_____

P l a n e t   L i s t
_____

go
---Field header:
Num.      Name          Light Years From   Earth Hours       Years Per
          The Sun        Per Rotation      Revolution
_____
go
---Field format:
####    <<<<<<<<<<<<  >>>>>>>>>>>>  #####.###       ####.##
go
---Footer:
_____
$longdate                                -$page-
go
---Page length [66]:
20
go
---Mail to:
go
---Printers:
go
---End.

```

#### 1. Database command

We've supplied the name of the macro *planetList* instead of an SQL or stored procedure statement in this section.

#### 2. Report Title

We have again included a multiline header for this report.

#### 3. Field header

In the last example report, we didn't use the Field header section. Here we have expanded the table's column names into a format that more clearly describes the values in each column. If this report had been more than one page long, the field headers would have been repeated at the top of each page in the report.

#### 4. Field format

The *Num.* field uses the number format to align numbers in a right justified column.

The *Light Years From The Sun* column returns a double precision number into a right justified text field. Since it's text, this field uses the dbView set commands: *set reals g* and *set doublePrecision 4*. Notice how this format allows you to return mixed decimal and scientific notation in a single field.

The third field is double precision like the second, but it has its own specific number format. Since it's a number format, it ignores the values for the *set* commands. It specifies three digits to the right of the decimal point and that's what we see, even though the value for *set doublePrecision* is 4. (The set commands only effect character format fields.) The decimal number format also forces the numbers to align on the decimal point.

The third field also contains a NULL value, which is printed as such.

The last field is like the third field except that it is a single precision number and has a specification for two digits to the right of the decimal point. For reports, you can specify the precision for individual real number fields, which is not the case for data retrieved directly by dbView.

#### 5. Footer

The left side of the footer begins with a line that separates the report's body from the footer. The next line in the footer includes the special footer variables *\$longdate* and *\$page*. As far as possible, dbView report specifications show you the final look of the report. But, because the special variables are expanded when the report is generated, this is not possible for the footer section when special variables are included. Trial-and-error is the only way to get the alignment you want in this case.

#### 6. Page length [66]

We have set the page size to 20 to give you some idea of what a full page would look like. Notice the space between the last row and the page footer.

#### 7. Mail to

We have not included a mailing list for this report. We simply supplied the "go" terminator to end this section of the report.

#### 8. Printers

We have not included a printer list for this report.

In this report we have looked at the following additional capabilities of the report function:

- The use of macros in reports.
- Number formats using both character and number format specifications to change the representation of the numbers.
- The interaction of the *set* commands and the right justified character format when num-

bers are returned into such a field.

- The use of multiple lines in the footer of a report and the special variable *\$longdate*.

## 19.2 Summarizing dbView's Report Writing Capabilities

Now that you've seen a couple of sample reports, we summarized the report functions capabilities and restrictions so you can begin creating your own reports.

- Each section of a report specification is terminated with dbView's "go" terminator. If a section is left blank—it only contains a line with a "go" terminator, the section is not used in the report.
- The Database command section can contain an SQL SELECT statement, a stored procedure or a dbView macro command.
- The report only formats data returned as a single table of information; however, queries producing tabular results can include joins, nested selects, group bys, etc.
- All of the format sections can contain multiple format lines.
- The Field format section can contain both boiler plate and field format specifications.
- Attributes in the database query map to field specifications. The mapping starts with the first attribute mapping to the first field specification, starting at the upper right of the field specification section.
- The number of characters in a format specification is the maximum number of characters used by a retrieved value. In the case of left justified text, the line will be wrapped if the value is larger than the field.
- You can include the current date, date and time and page number in the footer section of a report using the special variables.
- By default a report contains 66 lines, but you set the number yourself.
- You can include a list of eMail addresses and printers to be used when the report is generated. Once specified in a report, these options can be suppressed using the *set mailReport* and *set printReport* commands.
- Reports are saved as files that are read into dbView when you edit or run a report.

### 19.2.1 Report Functions

- *report <file name> [<history number>]*—Create a new report.
- *edit report <file name>*—Edit an existing report.
- *run report <file name> [<report name>]*—Run the report whose specification is in the named



Format	Examples	Description
<<<<<<<<<<<<	1234 1234.123 0.1234E-6	Numbers returned in left justified character format The commands <code>set real</code> , <code>set singlePrecision</code> and <code>set doublePrecision</code> can be used to control number formats as a character string.  Note: You must supply enough characters in the format string to accommodate the “E” notation if you have specified its use.

Numbers returned in character format fields are effected by the following `dbView set` commands:

```
set reals { f | e | E | g | G }
```

```
set singlePrecision { on | off }
```

```
set doublePrecision { on | off }
```

#### 19.2.4 Special Variables Used In Report Footers

The following table summarizes the special variables that can be used in report footers.

Special Variable	Examples
<code>\$date</code>	Fri Jun 04, 93
<code>\$longdate</code>	Mon Jun 07 16:07:19 1993
<code>\$page</code>	1 (\$page) -1- (-\$page-) Page: 1 (Page: \$page)

#### 19.2.5 Options For Report Printing and Mailing

If your report contains a print command or a mail list, the report will execute these subcommands by default. You can turn-off either one or both using the following commands:

```
set printReport {on | off}
```

```
set mailReport {on | off}
```

Since these are set commands, their values are remembered by `dbView` from session to session—the are saved in the `.dbview` file—so turn them on again when you want them to take effect.

When you run a report that has a mail list or printer list associated with it, `dbView` will tell you whether or not it performed either of these functions. (It won't perform the functions

even when their defined in your report if you've set the *mailReport* or *printReport* commands to "off".) The messages you'll see are:

"Report mailed" or "Report not mailed; mailReport setting is turned off"

"Report sent to the printer" or "Report not printed; printReport setting is turned off".

These messages are only printed for those cases where a mail or printer list was specified.

### 19.2.6 More About The Print Section

When you include a print specification in a report, dbView concatenates the name of the file to print *at the end of the print command*. The file printed by dbView is the name of the report if you included one, or a temporary file that dbView creates for this purpose. If a temporary file is used, it's deleted once the file has been printed.

If you are using a print command that must include a file specification *within the command*, you can't use the report's print specification; but there is an alternative:

1. Open a data file just before you run the report.
2. Run the report.
3. Close the data file.
4. Execute your print command.
5. Remove the data file containing the report.

All of these operations can be placed in a dbView script file and executes by a single *script* command. We show an example of this procedure when we describe *scripts*, see "*Generating And Printing A Report*" on page 126.

### 19.2.7 Cancelling A Report Specification Command

When you begin a report specification, dbView opens a file for the specification. If you abort the session by using <control-c>, the file is closed and removed.

### 19.2.8 Error You May Encounter When Using Report Commands

1. If you supply the *report* command the name of a file that already exists, the command is aborted.

```
1> report mission.rpt
      File mission.rpt already exists.  Report definition
      aborted.
```

2. The number of field format specifications must match the number of attributes returned by the query statement. If there is a mismatch, dbView will return an error message.

If there are fewer format specifications than query attributes, dbView displays an error message:

```
1> run report example.rpt
      Report specified 1 field format(s).
      There were 2 specified in your database statement.
```

If there are more format specifications than query attributes, the error message is like:

```
1> run report example.rpt
      Report specified 3 field format(s).
      There were 2 specified in your database statement.
```

3. If you have altered the sections of a report or entered a specification that dbView does not understand, the *run report* command is aborted and you're given an error message.

```
1> run report bad.rpt
      Invalid report file.  Could not find section:
      ---Field format:
```

4. If a right justified ">" or numeric "#" field specification is not large enough for a value, the value is not printed. Instead, the field is filled with star characters "\*".

P l a n e t   L i s t				
Num.	Name	Light Years From The Sun	Earth Hours Per Rotation	Years Per Revolution
2	Venus	****	NULL	0.62
3	Earth	****	*****	1.00
5	Jupiter	****	*****	12.34
6	Saturn	****	*****	29.02

Tue Jun 08 16:45:24 1993 -1-

*Table 5: The Formatted Report With Numeric Overflow*

5. If the query associated with a report fails, you will see an error message following any

header information that would appear in the report. In the following example, the significant error message returned by the database server is set in italic type:

```
1> run report bad.rpt
```

```
MISSION REPORT
```

```
MDMS DBS WARNING busstop::dbView Tue Jun  8 16:38:03 1993
(Db: jar, MsgNo: 102, Svr: 15, St: 1)
Incorrect syntax near 'missions'.
      Report specified 5 field format(s).
      There was 0 specified in your database statement.
```

### 19.3 Using The History List For Report Generation

When you create a report, you can use a query command that is in dbView's history list by following the name of the report with the history number (see "*The History List*" on page 63 for more on the history list). This is a very good approach because it insures that the command you are going to use already works.

In the following example, we have included the 9th command from the history list as the report command:

```
1> report mission.report 9
```

When you now press <return>, you will see something like this:

```
---Database command:
1> select mission, scId
2> from missions
3> order by mission
```

where the SELECT statement was the 9th command in the history list.

### 19.4 Hints For Creating Reports

In this section we provide some hints that may make creating reports easier for you.

- Use the *show db* command to get the list of attributes and data types for the tables you want to include in your report. You can save this information to a file with the *open logFile* command and then print out the file.
- Create your query statement before defining a report. Either make a macro or get the statement from the history list.
- If you want to include comments in your report that will appear at run time, use a macro

and include the comments in the macro.

- Use a common suffix for your reports and put them all in one directory. Then make a macro something like this:

```
1> macro showReports
---Command
1> escape ls *.rpt1
2> go
---Comment
1> dbView reports.
2> go
```

Then you can get a list of your reports by executing the command:

```
1> showReports

dbView reports.

mission.rpt    planets.rpt
```

- Use the *report* command to enter all of the information needed in the different sections of a report, but don't worry about placement of each field. Then used edit report to adjust the placement of boiler plate and field specifications.
- Take the time to make sure that the number of attributes in the database query matches the number of format specifications in the *Field format* section of the report specification.
- If your *report* depends on the values of *set* commands or the definitions of macros, put all of the commands needed for the report, including the *run report* command, in a script file and run that when you want to generate a report.
- If you want to run multiple reports as a single document with each report beginning on a separate page, put all of the report commands in a script file. When they are printed, the output will look like a single report. If, on the other hand, you want all of the information to flow across page boundaries, open a data file—*open dataFile <file name>*—and dbView will write the output to the file. Then close it and print it with command in the script file.

---

1 The example uses the Unix list file command (`ls *.rpt`). Another operating system would require the equivalent command.

## 20 The Script Command

dbView can execute a set of commands from a file using the *script* command. The commands can be a mixture of dbView and SQL commands. The general rule is: if you can execute it interactively, you can execute it from a script file. Commands saved using the *open command-File* command can also be used as scripts.

The syntax for the command is:

```
script <command file name>
```

and an actual command would look like this:

```
1> script example.script
```

You can also use macro to customize script commands:

```
1> macro example  
---Command  
1> script example.script  
2> done
```

```
1> example
```

Script can be used to:

1. Condense a set of commands by placing them in a file and then running the file as a script.
2. Execute a set of SQL commands. If you want to create a database schema, put the CREATE commands in a script.
3. Execute a set of reports. If you are running multiple reports, put the reports into a script file. Also, if the reports depend on *set* command values and macros, include those commands in the same file.
4. Exporting data. If you commonly retrieve data and store it in a file and then call another program to read the file, you can put these commands in a script. (Call the other program using dbView's *escape* command.)
5. Archiving or moving database sets. You can retrieve data using dbView's *export format* and write the data to a file. Then, using the *escape* command you can invoke a database load utility to insert the data into another database.
6. Run demonstrations. You can put a set of SQL commands in a script file and then run it as a "demo".

## 20.1 Some Characteristics Of Scripts

### 20.1.1 Scripts Can Be Nested

A script file can execute other script files. That is, within a script file, you can include a command to run another script file. For example, if you have a set of scripts each one of which downloads data from a database table, you can create one script that calls all of the others; and in that way, you can download a set of tables with a single command. As requirements change, you can add or delete entries from the master script file.

### 20.1.2 Putting Comments In A Script File

dbView considers all lines in a script file to be commands. You can enter comments—internal documentation within the script file not to be executed—by following any comment lines with a dbView *reset* command. For example:

```
1> File: dbView.sql
2>
3> This script file creates all of the database objects
4> used to create the examples in the "dbView Tutorial"
5> reset

1>
```

As we add text, dbView starts to build a multi-line command; but at some point, we want the text to be ignored, because its meant to be a comment. We signal dbView to ignore all previous input using the reset command. Once we've done that, we're free to start a new command.

### 20.1.3 Pausing In A Running Script

If you have a set of SQL commands in a script, they are executed as a batch; as soon as one completes, the next one executes. You could stop the output using the *set page* command, but this causes dbView to stop on page boundaries. If you want to stop at the beginning of the SQL command so you can see what will be done next, use the *leslie* command<sup>1</sup>. The script will pause as soon as it reads the command and wait for you to hit <return> before continuing.

```
1> script leslie.script
    Running script file leslie.script.

1> select id, scId, objective
2> from missions
3> where mission = "GLL"
4> go
```

---

1 The command is named after Leslie Pieri who asked for this capability.

```
id          scId          objective
-----
           2          35  Jupiter
```

(1 row(s) affected)

```
1> *****
2> Example using the "leslie" command to pause
3> in a script file.
4> *****
5> reset
```

```
1> leslie
Please type <return> to continue: ← The script pauses here
```

```
1> select id, scId, objective
2> from missions
3> where mission = "Cassini"
4> go
```

```
id          scId          objective
-----
           1          72  Saturn
```

(1 row(s) affected)

```
1>      End of script file leslie.script.
```

#### 20.1.4 Rules To Remember When Running Scripts

1. All commands requiring a “go” terminator on the command line also require the terminator in the script file.
2. If you have a file open—command, data or log file—before you begin running a script; and then you open another file of the same type in the script without first closing the open file, dbView will give you a warning message. The new file will not be opened, but the script will continue to run.
3. Any changes made to the dbView environment in a script file remain once the script file has completed execution. Changes accumulate whether you enter them interactively or by using a script. For example, if you executed the command: *set timer on* in a script file; and then followed that by executing a SQL query interactively, dbView would display timing results for the query.

## 20.2 Example Script Files

To give you an idea of how scripts can be used, we'll look at three examples:

1. Loading SQL command into a database.
2. Generating a report where set and macro commands are used.
3. Exporting a data set and loading it into another database.

### 20.2.1 Loading SQL commands

Here's the beginning of the file that creates the database objects used to create the examples in this tutorial. The complete file can be found in Appendix A of this document. The file was executed using the *script* command:

```
1> script dbView.sql

*****
**
** File: dbView.sql
**
** Function: Creates objects used for examples in the dbView Tutorial.
**
** Creator: J. Rector
**
** Created: March, 1993
**
*****

reset1

drop table missions
drop table planets
drop view missionObjective
go

drop procedure showMissionObjective
drop procedure showMissions
drop procedure showPlanets
go

create table missions
(
  id          id,
  mission    shortName,
```

---

<sup>1</sup> Placing a *reset* command following the comment also allows us to run this script using Sybase's command line utility *isql*.

```
        scId      id,  
        objective  shortName  null,  
        flying    flag,  
        description description null,  
        created   timeOfDay  
    )  
  
go
```

### 20.2.2 Generating And Printing A Report

In this example, we run a script that generates the “planets” report, see *“The Report planets.rpt”* on page 110, where this report is discussed.

The script used to create the report sets dbView parameters, includes a macro file used by the report and then runs the report. The script file includes a file header which we treat as a comment. We’ve added a *reset* command following the file header to clear the command buffer so the comment is not included as part of the first command. This is the general way to handle comments in script files.

```
*****  
**  
** File: planets.script  
**  
** Function: Set up the environment and then  
** run the report "planets.rpt".  
**  
*****  
reset ← reset the command buffer after a comment  
** Define the values for the set commands  
** needed by the report.  
reset  
set reals g  
set doublePrecision 4  
set verbose on  
set mailReport off  
set printReport off  
** Include the macro file example.macros.  
** It contains the macro planetList used  
** in the report.  
reset  
include macro example.macro  
** Run the report.  
reset  
run report planets.rpt planets.out
```

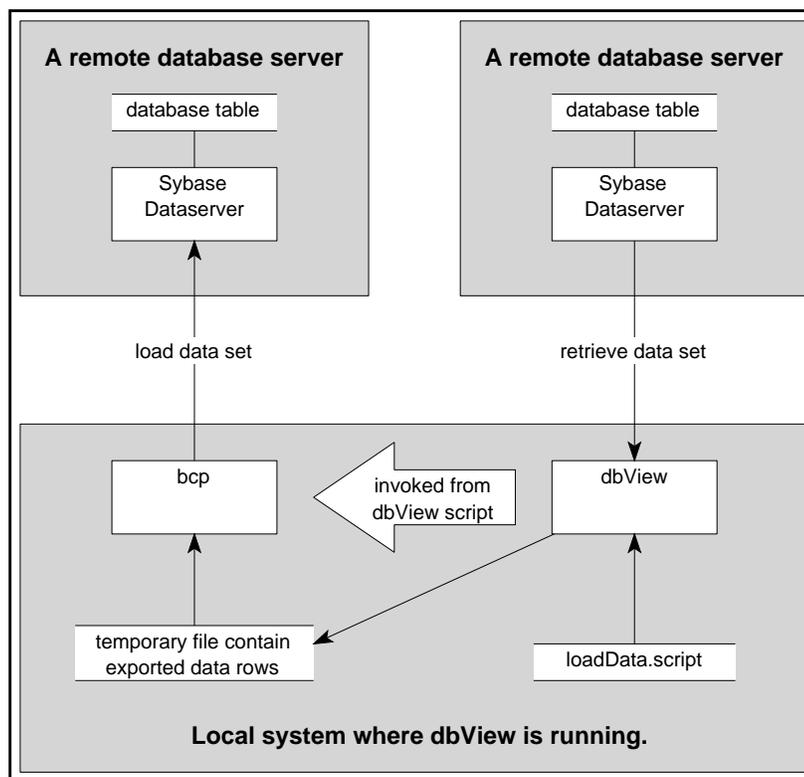
Now we can run the report using the script file in which these commands are saved:

```
1> script planets.script
```

The report sets up the dbView environment for the report, setting numeric specifications for real numbers, turning on verbose so the macro comment in the report will be seen, and suppressing any mail or print commands in the report. The script next includes the file of macros used by reports. Then the report is run and the output sent to the file `planets.out`.

### 20.2.3 Copying A Database Tables Contents

This report retrieves all of the data from a database table from one Sybase database server and loads it into a copy of the table on another Sybase database server. The data is retrieved by dbView and loaded by Sybase's utility program *bcp*. The illustration below shows the three machine environments and the location of each of the processes and files involved.



This script that makes the transfer uses global variables and macros that generalize the script so any table can be copied from one server to the other. The illustration shows that the database table must exist on in both databases—the script only copies the contents of one table into another, it does not make a copy of the table definition itself.

We'll describe the contents of the script in some detail because it brings together many of dbView's commands to solve a particular problem. The script file appears on the following pages.

Here's what the script file does:

1. The first command in the script, *set displayScriptCommands off*, suppresses the display of most of the commands in the script with the exception of:
  - Macro comments
  - Macro local variable prompts
  - Messages returned to dbView from the operating system.

You can see from the script that there are many commands in it that we don't need to see when we run it. Setting *displayScriptCommands* to "off" will allow us to design the script in such a way that it will have an informative look when it's run.

2. Next we have the documentation of the file treated as a comment within the script file. We create a comment like any other information to be accepted by dbView's command buffer, but we end it with the *reset* command so that it's not executed. We arbitrarily place two stars in front of each comment line so there's no chance that dbView will interpret something we type in as a command.
3. The next set of commands in the script are macros and global variable definitions that will be used by the script. (We could have put these commands in a macro file and included the macro file in the script.)

The first macro, *getTargetObject*, is used to define the value for the global variable *object*. We place the global variable definition in a macro so we'll be prompted for its value when we run the script.

We next use that global variable to create another one named *fileName*. We'll use this global variable to manage the temporary file containing the data *bcp* will read.

The next few macros create commands to: open a temporary data file, delete the file and select all of the data from the table defined by the global variable *object*. This set of global variables and macro examples was discussed earlier, see "*Referencing Global Variables In Macros*" on page 93.

We've purposely added comments to our macros so they will appear when we run the script—remember, we're going to run it having set *displayScriptCommands* to "off" and *verbose* to "on", so we'll see the macros' comments and prompts, but little else.

4. The last macro we define:

```
macro bcpCommand
escape bcp example.dbo.$$object in $$fileName -c -U sa -P
$password -S MDM1
go
Loading data into the "example" database
using bcp.
go
```

warrants its own description. The line in bold type is the macro command. It escapes to

the operating system and executes the Sybase *bcp* utility program that loads the data from the file defined by *\$\$fileName* into the table defined by *\$\$object*. The command also includes the special local variable *\$password*. When this macro runs, we'll be prompted for the password that *bcp* needs for the data server where the data will be loaded. The password won't be echoed because that is the nature of this special local variable—see "*The Special Local Variable \$password*" on page 84.

5. We next set the environment:

- Turn on *verbose* so macro comments will appear.
- Set *displayRows* off so the rows retrieved will not be written to the screen.
- Set *header* off so the data file read by *bcp* will not contain the table header.
- Set the *endField* and *endRow* values explicitly so *bcp* correctly interprets the end of fields and rows in the file.
- Set the *format* to *export* so the field and row delimiter are included in the output.

6. Now the work begins:

- The macro *getTargetTable* is executed to assign the name of the table to copy to the global variable *object*. (Since the global variable *fileName* is already defined, *dbView* knows how to construct its value using the value of *object* at run time.)
- We delete any existing copy of the temporary file we will use with the macro *deleteDataFile*. (The temporary file name is defined by the global variable *fileName*.)
- Open the temporary data file.
- Select the data. (Since there's a data file open, the rows returned will be written to that file. And since we're in export format, the file will be readable by *bcp*.)
- Close the data file.

At this point the data has been retrieved and written to the data file in the export format we defined.

7. Next we execute the macro *bcpCommand* described above. This loads the data from the temporary file into a table by the same name in the database `example`.

8. Once the data is loaded, we:

- Reset the *dbView* environment for interactive use.
- Delete the data file

- Remove the macros and global variables

```
set displayScriptCommands off

*****
**
** File: loadData.script
**
** Function: Retrieve the rows from a table and
** load it into the database named "example" for user "dbo"
** using the Sybase utility "bcp" (bulk copy).
**
*****
reset

** Define a macro that prompt for the name of the file
** to use when saving and loading data. The file name
** is stored as a global variable and used in macros
** that open the data file and load the database.
reset
macro getTargetTable
global object $targetTable
go
Enter the name of the table whose contents are to be
copied.
go

** Use the value of the global variable "object" concatenated
** with the string "Tmp" to define the global variable
** "fileName". (We use the escape character '\' to separate
** the global variable name $$object from the string "Tmp".)
reset
global fileName $$object\Tmp

** Create a macro to open the data file. A macro is
** used so the global variable holding the name of
** the file can be expanded into the "open dataFile"
** command.
reset
macro openDataFile
open dataFile $$fileName
go
go

** Create a macro to delete the data file.
reset
macro deleteDataFile
escape rm $$fileName
```

```
go
Deleting old data file if it exist.
go

** Create a macro to select all fields from a table.
** We use the global variable $$targetTable as the
** name of the table in the SELECT statement.
reset
macro selectData
select *
from $$object
go
Selecting data from target table.
go

** Create a macro for the bcp command. It will prompt for
** the Sybase password needed to load the data. The macro
** also contains the global variable that contains the
** name of the file to load.
reset
macro bcpCommand
escape bcp example.dbo.$$object in $$fileName -c -U sa -P
$password -S MDM1
go
Loading data into the "example" database
using bcp.
go

** Set verbose on so macro comments will appear.
** Set the dbView environment so only rows of data
** will be written to the file. Set up for export.
reset
set verbose on
set displayRows off
reset
set header off
set endField \t
set endRow \n
set format export

** Get the table name.
** Delete any old copy of the file. (Don't use this
** unless you confident that this will not lead to
** some untoward event!)
** Open the file.
** Retrieve the data.
```

---

```
** Close the file.
reset
getTargetTable
deleteDataFile
openDataFile
selectData
close dataFile

** Now load the data into the "example" database.
** The bcp command is in a macro that will prompt for the
** password. To execute the bcp command, we "escape" it.
reset
bcpCommand

** Reset the environment for normal interactive use.
** Remove temporary data file.
** Remove the macros and globals we've created.
reset
set verbose off
set displayRows on
set header on
set format table

deleteDataFile

remove macro getTargetTable
remove macro openDataFile
remove macro deleteDataFile
remove macro selectData
remove global object
remove global fileName
```

Since we want to run this script often, we'll shorten the command by including it in a macro:

```
1> macro loadData
  --Command
1> script loadData.script
2> done
```

Normally, you can't include a macro within a macro—we're calling a script that includes macros from the macro "loadData"—but in this case it's all right. dbView knows how to handle this special case.

Now we execute the macro which invokes the script and dbView displays the following:

```
1> loadData ← this is the macro command
      Running script file loadData.script.

1> set displayScriptCommands off

Enter the name of the table whose contents are to be
copied.

targetTable []: missions

Deleting old data file if it exist.

missionsTmp: No such file or directory

Selecting data from target table.

Loading data into the "example" database
using bcp.

password: ← the password is not echoed

Starting copy...

4 rows copied.
Clock Time (ms.): total = 1      Avg = 0      (4000.00 rows per
sec.)
      End of script file loadData.script.
```

## 21 dbView's Batch Mode

dbView can also be used in *batch mode*. In this mode, dbView never begins an interactive session; instead it executes the command placed in a script file. The name of the script file is supplied as a command line parameter to dbView. The syntax is:

```
dbView <script file>
```

For example, the following line, specified *at the operating system prompt*, would execute the contents of the script file `nightlyReports.script`:

```
% dbView nightlyReports.script
```

Since there is no way to supply dbView with the passwords to database servers you access from the script, this mode only works if your system supports the JPL password server. The function of the password server is beyond the scope of this guide. To find out more about it ask your DBA or look at the MDMS Mosaic Home Page:

```
http://www-mipl/mdms/MDMS.html
```

When using the password server, you must have the following defined:

1. You must be a known user with the password servers Kerberos domain.
2. Your database server user names and passwords must be defined in the password server.
3. You must have a valid Kerberos ticket, granted when you successfully execute the Kerberos *kinit* utility.

You can also execute dbView in batch mode as a background process. To do this the previous command becomes:

```
% dbView nightlyReports.script &
```

Ask your system administrator for help in this area if you're unfamiliar with background processing and its uses.

## 22 Error Messages

Most dbView error messages are printed as text without a header; but when the error comes from Sybase, you'll get an error message with a banner. In this section, we'll describe this special class of messages.

### 22.1 What's In An Error Message?

An error message begins with a banner line, optionally followed by a Sybase error number line, and ending with the message. This may seem like a lot of information to you, but it comes in handy if you need to report a bug. A developer has a much better chance of understanding what went wrong if this information is included. And, it should also tell you what went wrong. Let's look at each line in an error message to see what information it contains. While our discussion will cover all of the variations you can expect, we include an example for reference. In the example, we try to query for all the information in the table "noTable", which does not exist in the database.

```
1> select * from noTable
2> go
```

```
MDMS DBS WARNING milano::dbView Wed Jun 23 16:52:15 1993
(Db: master, MsgNo: 208, Svr: 16, St: 1)
Invalid object name 'noTable'.
```

#### 22.1.1 The Banner Line

From the example

```
MDMS DBS WARNING milano::dbView Wed Jun 23 16:52:15 1993
```

1. The acronym of the group responsible for the message. In dbView this will always be MDMS—the MIPS Data Management Subsection.
2. The type of error
  - DBSERVER—The message was generated by a database server.
  - DBLIB—The message was generated by the Sybase interface library.
  - PROGRAM—The message was generated in program code; in this case dbView's code.
3. The severity of the error:
  - INFORMATION—Not an error at all; just an informative message.
  - WARNING—The message is a warning.

- **ERROR**—More severe than a warning, something did not execute correctly.
  - **FATAL**—More severe than an error. You should never see this. If you do, it indicates that the program can not proceed.
4. The next two words are the name of the client machine and the program. The two are separated by a double colon. In the example above, the client machine is named *milano* and the program is *dbView*.
  5. The last item on the banner line is the date and time of the error.

### 22.1.2 The Sybase Error Number

From the example:

```
(Db: master, MsgNo: 208, Svr: 16, St: 1)
```

If the error came from Sybase there will be an additional line between the banner and the error message. For an error returned by a server (DBSERVER), the line contains the following information:

1. The name of the database if you are connected to the server.
2. The Sybase database server message number (MsgNo).
3. The severity (Svr) of the error.
4. The state (St) of the server.

If the error comes from the Sybase interface library (DBLIB), the line contains the following information:

1. The Open Client/C error number.
2. The severity (Svr) of the error.

Generally you don't need to be concerned with the information on this line, but the Sybase documentation contains information about all errors if you are interested. Always include the information in this line if you are reporting a bug. The developer getting your report will want it.

### 22.1.3 The Message

From the example:

```
Invalid object name 'noTable'.
```

The last item in an error message is the text of the message itself. You want to read this; it tells you the nature of the error.

Often you will receive more than one error message. Don't get unsettled by this. In a client/server environment, many pieces of software may be involved in any action you take. Multiple error messages just show the chain of services that reacted to your command. The first message is usually the important one for you. By reading it, you should understand the source of the error. If you don't, look at any other messages returned.

When reporting a suspected bug, always include all of the error messages generated by the command. You have a much better chance of getting fast results if you do this.

## 22.2 Some Common Login Errors

Connecting to a database server from dbView should be easy; but when you can't make a connection, it's very frustrating, so in this section we'll cover some common login errors. When an error occurs, dbView sends you an error message. The message usually has two or three lines to it. For now, just look at the last line of the message. (For an explanation of error message syntax, see "Error Messages" on page 136.)

In the following examples, we've highlighted the important part of each message in bold type.

### 22.2.1 Incorrect User Name Or Password

```
userName [franklin]: adams
password:
server [CATALOGDBS]:
database [catalog]:
```

```
MDMS DBS WARNING milano::General Delivery Tue Mar  2 10:35:40
1993
```

```
(Db: , MsgNo: 4002, Svr: 14, St: 1)
```

```
Login failed.
```

```
MDMS DBLIB WARNING milano::General Delivery Tue Mar  2 10:35:40
1993
```

```
MsgNo: 20014, Svr: 2
```

```
Login incorrect.
```

- **Response**—You've probably supplied an incorrect login value. Use the *connect* command to enter the correct values. If you don't have a user name and password, you'll need to get one from your database administrator.

### 22.2.2 Incorrect Server Name Or Server Name Not In Interfaces File

```
userName [franklin]:
password:
server [CATALOGDBS]: junk
database [catalog]:
```

```
MDMS DBLIB WARNING milano::General Delivery Tue Mar  2 10:41:08
1993
```

```
MsgNo: 20012, Svr: 2
Server name not found in interface file.
```

```
MDMS PROGRAM ERROR milano::dbView Tue Mar  2 10:41:08 1993
```

```
dbopen error for SQL command: junk server connection
```

- **Response**—You may have misspelled the server name. Use the `connect` command to enter the correct value. Or, you may have a problem with your Sybase environment, refer to the *Installation Guide*. In the latter case, see if the server name is in the `interfaces` file. If not, contact your database administrator for assistance.

### 22.2.3 Incorrect Database Name

```
userName [franklin]:
password:
server [CATALOGDBS]:
database [catalog]: catlog
```

```
MDMS DBS WARNING milano::dbView Tue Mar  2 10:43:56 1993
```

```
(Db: catalog, MsgNo: 911, Svr: 11, St: 2)
Attempt to locate entry in sysdatabases for database 'catlog'
by name failed - no entry found under that name. Make sure that
name is entered properly.
```

```
MDMS PROGRAM WARNING milano::dbView Tue Mar  2 10:43:56 1993
```

```
dbuse error for SQL command: catlog database
```

- **Response**—You may have misspelled the database name; we typed “catlog” when we wanted “catalog” in the example. At this point Sybase should have placed you in your default database. Execute the command “`sp_who`” to find out where you are. If you need a list of database names for the server, execute the command “`sp_helpdb`” and then connect to the database you want with the command “`use <database name>`” followed on the next line by the command terminator “`go`”.

### 22.2.4 Server Is Not Running

```
userName [franklin]:
password:
server [CATALOGDBS]:
database [catalog]:
```

```
MDMS DBLIB MSGFAILED milano::General Delivery Wed Mar 17
11:05:04 1993
```

```
MsgNo: 20009, Svr: 9
Unable to connect: SQL Server is unavailable or does not exist.
```

```
MDMS PROGRAM ERROR milano::dbview Wed Mar 17 11:05:04 1993
dbopen error for SQL command: catalog server connection
```

- **Response**—The server is not running so you can not connect to it at this time. Contact your database administrator to find out when the server will be back on-line.

### 22.2.5 Can't Reach Machine Named In Interfaces File

This is an example where the server name is correct, but the machine name associated with the server named in the `interfaces` file can't be reached. You will experience a delay of about a minute if this error occurs because `dbView` is attempting to make the connection and only gives up after the time-out period expires.

```
userName [franklin]:
password:
server [CATALOGDBS]:
database [catalog]:
```

```
MDMS DBLIB MSGFAILED milano::General Delivery Tue Mar  2
10:53:21 1993
MsgNo: 20013, Svr: 9
Unknown host machine name.
```

```
MDMS PROGRAM ERROR milano::dbView Tue Mar  2 10:53:21 1993
dbopen error for SQL command: CATALOGDBS server connection
```

- **Response**—The `interfaces` file you are using is incorrect. Contact your database administrator or system administrator who will correct the error.

# Database Bibliography

The books and manuals listed here offer an *introduction* to relational databases with special emphasis on Sybase. The list is by no means complete; but many of the books have additional references that will lead you to more specialized topics.

If you are new to databases — at least databases that use SQL — and your main goal is to learn how to query a database, look at the first couple of references in the section “The SQL Language”.

## 23 General Relational Database References

The books in this section are valuable if you want to learn more about relational database concepts. More specific topics, especially those related to the SQL language and Sybase, are covered in the following sections.

An Introduction To Database Systems; C. J. Date; Addison–Wesley; Volume I.  
*One of the standard academic texts on databases. This book is a good introduction for people who must design or understand relation database schemas.*

Relational Databases: Selected Writings; C. J. Date; Addison–Wesley.  
*A collections of papers on relational database design issues by one of the prominent members of the field.*

Principles Of Database And Knowledge–Base Systems; Jeffrey D. Ullman; Volumes I and II.  
*Covers a wide range of database topics in depth; many of them beyond the basics description of relational databases. The approach is formal, with lots of mathematical proofs.*

Database Security And Integrity; E. B. Fernandex, R. C. Summers, C. Wood; Addison–Wesley.  
*An older text on database security. Covers a lot of the issues, even if some of the solutions are dated. New topics like C2 and B2 level secure serves and Kerberos authentication are NOT covered.*

## 24 The SQL Language

If you need to know enough about databases to query one for data, then these are the books

you should consider. The last reference is more specialized than the first two.

There are many other books in the class. Look around at your favorite technical book store, you may find one you like better.

If you are primarily interested in Sybase, you should also look at the books in the next two sections.

Learning SQL; Wellesley Software; Prentice Hall; ISBN 0-13-528704-9  
*An entry level introduction to SQL with exercises. Good for starting on your own with no prior experience with the language.*

A Visual Introduction to SQL; J. Harvey Trimble, Jr. and David Chappell, Wiley, ISBN 0-471-61684-2  
*An entry level introduction to SQL using diagrams to show the relationship between SQL and database tables. Another good starter book for someone learning SQL on their own.*

The Practical SQL Handbook; Judith S. Bowman, Sandra L. Emerson and Marcy Darnovsky; Addison-Wesley; ISBN 0-201-62623-3.  
*The only introductory SQL book that discusses the Sybase implementation. The examples in the book use the Sybase example database "pubs"<sup>1</sup>.*

SQL & Relational Basics; Fabian Pascal; M&T Books; ISBN 1-55851-063-X  
*A good book to read once you've understood the very basics of SQL and want to go on to a better understanding of relational databases and the application of SQL.*

Introduction to SQL; Rick F. van der Lans; Addison-Wesley;  
*A good introduction to the SQL language with a lots of examples. The examples use a collection of tables defined in the book.*

A Guide to the SQL Standard, 3rd Edition; C. J. Date with Hugh Darwen; Addison Wesley.  
*Discusses the SQL/89 and SQL/92 standards. A good book to have if you want to know about the latest, standardized syntax for SQL. Sybase System 10 adopts the SQL/89 standard along with Sybase's extensions to the language, Transact-SQL.*

## 25 Books About Sybase

These books describe the Sybase Database Management System and Sybase's client/server architecture. The books discuss the SQL language, but go into other topics as well. These are valuable to people who need to know more about Sybase than just how to query the database.

The Guide To SQL Server; Alope Nath; Addison-Wesley.  
*This book is about the Microsoft implementation of the Sybase SQL Server for PCs; but most of what is in it applies to any Sybase SQL Server. Good introduction to the topic with only minor*

---

1 The script file that creates the pubs tables in a database is called `pubs.sql`. Contact the Sybase Data Manager for a copy of the file.

*PC dependencies discussed.*

A Guide To Sybase And SQL Server; D. McGoveran with C. J. Date; Addison–Wesley. *A good introduction to Sybase and the SQL Server. A little more academic than Nath's book. Does not cover System 10 topics.*

Sybase Architecture and Administration; John Kirkwood and Ellis Horwood; ISBN 0-13-100330-5.

*An introduction to Sybase, its client/server architecture, the Transact-SQL language and administration of Sybase Database servers. Does not cover System 10 topics.*

## 26 Sybase Manuals

The manuals described in this section are some of the ones supplied by Sybase Corporation with their products. To buy the books, you must contact Sybase Corporation.

Transact-SQL User's Guide

*This is the best, and most complete, introduction to Sybase's extended SQL language. The examples in the guide use the "pubs" database, an example database supplied with the Sybase SQL Server.*

Command Reference Manual.

*The definitive reference manual for the Transact-SQL language. Each command is defined and described individually. This manual is not a tutorial.*

Open Client DB-Library/C: Reference Manual.

*The description of the Sybase client interface using the C Language. Of particular interest to programmers who must access a Sybase server.*

Sybase Installation And Operations

*How to install and configure Sybase software. Of interested only to System and Data Administrators.*

System Administration Guide.

*Most things that a Data Administrator needs to know about are covered in this guide.*



---

## APPENDIX: A

# Example Database

This appendix contains the data found in the *missions* and *planets* tables used as examples in the *User* and *Quick Reference Guides*. The script that creates these tables and other, associated objects follows the table listings.

id	mission	scId	objective	flying	description	created
1	Cassini	72	Saturn	0	The Cassini Mission to Saturn	Jun 22 1993 5:58:10:403PM
2	GLL	35	Jupiter	1	The Galileo Mission to Jupiter	Jun 22 1993 5:58:10:460PM
3	MO	91	Mars	1	The Mars Observer Mission	Jun 22 1993 5:58:10:476PM
4	VGR	0		1	The Voyager Mission to the outer solar system	Jun 22 1993 5:58:10:493PM

**Figure 1:** *missions Table*

number	name	lgtYrsFromSun	hrsPerRotation	yrsPerRev
1	Mercury	6.30786185000e-06	211.284	0.241100
2	Venus	1.14223444000e-05		0.616400
3	Earth	1.58548960000e-05	24.0000	1.00000
4	Mars	2.42085509000e-05	24.6640	1.88200
5	Jupiter	8.23431695000e-05	10.0210	12.3430
6	Saturn	0.000151047719000	10.2730	29.0250
7	Uranus	0.000303459300000	10.8010	84.1100
8	Neptune	0.000475646880000	15.8980	165.592
9	Pluto	0.000662567170200		248.637

**Figure 2:** *planets Table*

---

```

*****
**
** File: dbView.sql
**
** Function: Creates objects used for examples in the dbView User's Guide.
**
** Creator: J. Rector
**
** Created: March, 1993
**
** Owner of objects: dbo
**
*****

reset

*****
**
** DOMAINS
**
*****

reset

sp_addtype name, "varchar(30)"
go
sp_addtype shortName, "varchar(15)"
go
sp_addtype flag, "tinyint"
go
sp_addtype flagArray, int
go
sp_addtype description, "varchar(255)"
go
sp_addtype radian, float
go
sp_addtype tinyId, tinyint
go
sp_addtype id, int
go
sp_addtype timeOfDay, datetime
go

*****
**
** DEFAULTS FOR DOMAINS
**

```

---

```

*****
reset
/*
** DEFAULT
**  default_timeOfDay
**
** FUNCTION
**  Uses the current database server time as the default.
*/
create default default_timeOfDay as getdate ()
go
exec sp_bindefault default_timeOfDay, timeOfDay
go

```

```

*****
**
** RULES FOR DOMAINS
**
*****

```

```

reset
/*
** RULE
**  rule_zeroOne
**
** FUNCTION
**  A boolean function. Value must be 0 or 1. The pair can signify
**  binary sets like {no, yes}, {off, on}, {not OK, OK}, {stop, go},
**  etc. The values can also be used for logical tests in programming
**  languages like C.
*/
create rule rule_zeroOne as @value in (0, 1)
go
sp_bindrule rule_zeroOne, flag
go

```

```

*****
**
** TABLES
**
*****
reset
/*
**
** TABLE

```

---

```

**      missions
**
** FUNCTION
**      Example spacecraft mission data used for dbView examples.
**
*/
create table missions
(
    id          id,
    mission     shortName,
    scId        id,
    objective   shortName    null,
    flying      flag,
    description  description  null,
    created     timeOfDay
)
go

create unique clustered index missionPK1 on missions (mission)
create index missionFK1 on missions (objective)
go

sp_primarykey missions, mission
go

/*
**
** TABLE
**      planets
**
** FUNCTION
**      Example table used for dbView examples.  Contains information
**      about the planets in the Solar System.
**
*/
create table planets
(
    number          tinyint,
    name            shortName,
    lgtYrsFromSun   float,
    hrsPerRotation  real      null,
    yrsPerRev       real      null
)
go

create clustered index planetsPK1 on planets (name)

```

---

go

sp\_primarykey planets, name  
go

sp\_foreignkey missions, planets, objective  
go  
sp\_commonkey missions, planets, objective, name  
go

```
*****  
**  
** VIEWS  
**  
*****
```

```
reset  
/*  
** VIEW  
**   missionObjective  
**  
** FUNCTION  
**   Joins information in the missions and objectives tables.  
*/  
create view missionObjective (mission, spacecraft, planet, lgtYrsFromSun,  
    hrsPerRotation, yrsPerRev)  
as  
    select mission, scld, name, lgtYrsFromSun, hrsPerRotation, yrsPerRev  
    from missions, planets  
    where missions.objective *= planets.name  
go
```

```
*****  
**  
** TRIGGERS  
**  
*****
```

```
reset  
/*  
** TRIGGER  
**   missionsInsUpdTrig  
**  
** FUNCTION  
**   The mission's objective value must be NULL or it must be in the
```

---

```

**          "planets" table before it can be used for insert or update.
*/
create trigger missionsInsUpdTrig
on missions
for insert, update
as
begin
    declare @objective name

    select @objective = objective from inserted

    if @objective = null
    begin
        return
    end

    if not exists ( select * from planets where name = @objective)
    begin
        print "Mission's objective not found in 'planets' table."
        rollback transaction
    end
end
go

/*
** TRIGGER
**          missionsDelTrig
**
** FUNCTION
**          Only the owner of the table can delete a row.
*/
create trigger missionsDelTrig
on missions
for delete
as
begin
    if not exists (select * from sysobjects
        where id = object_id ("missions")
        and uid = user_id())
    begin
        print "Only the table's owner can delete rows."
        rollback transaction
    end
end
go

```

---

```

*****
**
** PROCEDURES
**
*****

reset
/*
** PROCEDURE
**     showMissions [missionName]
**
**     FUNCTION
**     Show mission information.  If name is supplied, information for that
**     mission.
*/
create procedure showMissions
@missionName name = null
as
begin
    print "          MISSION INFORMATION"
    print " "
    if @missionName = null
    begin
        select mission, scld, objective
        from missions
        order by mission
    end
    else
    begin
        select mission, scld, objective
        from missions
        where mission = @missionName
    end
end
go

/*
** PROCEDURE
**     showPlanets [planetName]
**
**     FUNCTION
**     Display information about planets in the Solar System.
*/
create procedure showPlanets
@planetName name = null
as

```

---

```

begin
    if @planetName = null
    begin
        select name, lgtYrsFromSun, hrsPerRotation, yrsPerRev
        from planets
        order by number
    end
    else
    begin
        select name, lgtYrsFromSun, hrsPerRotation, yrsPerRev
        from planets
        where name = @planetName
    end
end
go

reset
/*
** PROCEDURE
**     showMissionObjective [mission]
**
** FUNCTION
**     Display information about the planets visited by spacecraft missions.
*/
create procedure showMissionObjective
@mission name = null
as
begin
    if @mission = null
    begin
        select mission, planet
        from missionObjective
    end
    else
    begin
        select mission, planet
        from missionObjective
        where mission = @mission
    end
end
go

*****
**
** CAPABILITIES

```

---

```
**
*****
```

```
reset
```

```
grant select, insert, update on missions to public
grant select on planets to public
grant select on missionObjective to public
grant exec on showMissions to public
grant exec on showPlanets to public
grant exec on showMissionObjective to public
go
```

```
*****
```

```
**
```

```
** DATA
```

```
**
```

```
*****
```

```
reset
```

```
insert into planets (number, name, lgtYrsFromSun, hrsPerRotation, yrsPerRev)
values ( 1, "Mercury", 6.30786185E-6, 211.284, 2.411E-1)
insert into planets (number, name, lgtYrsFromSun, hrsPerRotation, yrsPerRev)
values ( 2, "Venus", 1.14223444E-5, NULL, 6.164E-1)
insert into planets (number, name, lgtYrsFromSun, hrsPerRotation, yrsPerRev)
values ( 3, "Earth", 1.58548960E-5, 24.000, 1.000)
insert into planets (number, name, lgtYrsFromSun, hrsPerRotation, yrsPerRev)
values (4, "Mars", 2.42085509E-5, 24.664, 1.882E0)
insert into planets (number, name, lgtYrsFromSun, hrsPerRotation, yrsPerRev)
values (5, "Jupiter", 8.23431695E-5, 10.021, 12.343)
insert into planets (number, name, lgtYrsFromSun, hrsPerRotation, yrsPerRev)
values (6, "Saturn", 1.51047719E-4, 10.273, 29.025 )
insert into planets (number, name, lgtYrsFromSun, hrsPerRotation, yrsPerRev)
values (7, "Uranus", 3.03459300E-4, 10.801, 84.110)
insert into planets (number, name, lgtYrsFromSun, hrsPerRotation, yrsPerRev)
values (8, "Neptune", 4.75646880E-4, 15.898, 165.592)
insert into planets (number, name, lgtYrsFromSun, hrsPerRotation, yrsPerRev)
values (9, "Pluto", 6.625671702E-4, NULL, 248.637)
go
```

```
insert into missions (id, mission, scld, objective, description, flying)
values (1, "Cassini", 72, "Saturn", "The Cassini Mission to Saturn", 0)
insert into missions (id, mission, scld, objective, description, flying)
values (2, "GLL", 35, "Jupiter", "The Galileo Mission to Jupiter", 1)
insert into missions (id, mission, scld, objective, description, flying)
values (3, "MO", 91, "Mars", "The Mars Observer Mission", 1)
```

---

```
insert into missions (id, mission, scld, objective, description, flying)
values (4, "VGR", 0, NULL, "The Voyager Mission to the outer solar system", 1)
go
```

# Index

## B

bug reporting 15

## C

Can 29

cancel 40

    database command 40

    in a report specification 118

    with reset command 40

changing directories 67

command execution 39

commands

    cancel 23

    close commandFile 66

    close dataFile 66

    close logFile 66

    connect 31

    directory 67, 88

    edit macro 75

    edit report 105, 115

    escape 60, 79, 128

    exit (see exit) 25, 27

    expand global 93

    expand macro 94

    global 92

    help 61

    history 63, 77

    history list 120

    include macro 89

    last 57, 64

    leslie (pause) 123

    macro 72

    open commandFile 66

    open dataFile 66

    open logFile 66

    print (in macros) 73

    remove global 95, 130

    remove macro 87, 130

    rename macro 82

    repeat 86

    replace macro 90

    report 105, 115

    reset 40

    run report 105, 115

    save macro 87

    script (see scripts) 122

    set (see set also) 42

    show db 97

    show file 67

    show global 93

    show macro 74

    show servers 29

    show set 42

## D

database

    accessing 30

    catalog information 97

database catalog 97

database connection

    common errors 29, 138

    initial 21, 28

    re-connect 31

    retry on error 28

    setting default database 32

    timeout 46

    time-out period 29

database servers

    listing 29

database, locating 30

dbView

    installing 16

    starting 21, 27

    Sybase environment 16

## E

edit 23, 41, 44, 75, 79

environment 16, 46

errors 90

    database 42

    database server 136

    login 138

escape 79

escaping to the operating system 60

exit 25, 27, 31

    nosave 27, 46

    when macros are not saved 88

export 50, 51  
  copy and paste with 52  
  endField 44  
  endRow 44

## F

files 129  
  commandFile 66, 70  
  dataFile 66, 71  
  logFile 66, 68  
  show open files 67  
Format  
  numbers 44  
format 44  
  column spacing 46  
  header 44, 53  
  list 24  
  numbers 45, 54, 117  
    decimal 54  
    mixed format reals 57  
    scientific notation 56  
  page 45, 53  
  query results 48, 49, 50

## G

global variables 92  
  defining 92  
  displaying contents in a macro 94  
  expanding global variable definition 93  
  in macros 93  
  remove global 95  
  saving 96  
  saving to a file 87  
  showing definition of 93  
  underlined 95

## H

help 61  
history  
  length of history list 44  
history list 63, 72, 77, 78  
  length of 64

## I

interfaces file 29

## L

last 64

## M

macro  
  show macro  
    command  
      show macro 83  
macros 72  
  comments 128  
  default macro file 89  
  defining 72, 76  
  displaying global variables in 94  
  editing 75, 81  
  executing 73  
  exiting with changes pending 88  
  global variables 92, 93  
  in scripts 128  
  including macros from files 89  
  local variables 81, 85  
    defaults 82  
    persistent default 83  
    session default 83  
    special (\$password) 84  
  redefining dbView commands with 78  
  removing 87  
  rename 82  
  repeat 86  
  replacing files 90  
  saving  
    commands  
      save macro 88  
  saving to a file 87  
  scripts with 122, 134  
  sharing files 91  
  show macro 74  
  undefined global variables 95  
  using in reports 110  
  using the history list with 77  
  variables  
    special (\$password) 128

## P

page size 45  
password 84, 128  
pause 123  
prompts  
  default value 21

## R

relation databases 141  
relational databases 142

- release notes 15
- repeating macro commands 86
- report
  - mailing 45
  - printing 45
- reports 105
  - character string formatting 108
  - common errors made in 118
  - database command 113
  - database commands 107
  - defining 105
  - defining with history list 120
  - editiing specification 105
  - field format 108, 113
  - field header 108, 113
  - footer 109, 114
  - format specifications
    - character strings 116
    - numbers 110, 113
  - formats specifications
    - numbers 116
  - hints for creating 120
  - mail to 109, 114, 117
  - page length 109, 114
  - printers 110, 114, 117, 118, 126
  - report title 107, 113
  - rules about 115
  - running 105
  - using macros in 110
- S
- scripts 122
  - comments 128
  - examples
    - copying a database tables contents 127
    - loading SQL commands 125
    - printing a report 126
  - including comments in 123
  - invoking from a macro 134
  - macro comments 128
  - nesting script files 123
  - pausing in 123
  - rules for running 124
  - setting the environment for 129
  - suppressing command display 128
  - using macros in 128
  - using passwords in 128
- set 42, 43
  - defaultMacroFile 43, 89
  - displayRows 43, 47
  - displayScriptCommands 44, 128
  - doublePrecision 44, 55, 57, 111
  - doublePresision 117
  - editor 44
  - endField 44, 51
  - endRow 44, 51
  - feedback 44, 58
  - format 44, 48
    - export 50
    - list 49
    - table 48
  - header 44, 53
  - history 44, 64
  - mailReport 45, 114, 117
    - page 45, 53
  - printReport 45, 110, 114, 117
  - reals 45, 57, 111, 117
  - show set 42
  - singlePrecision 45, 55, 57, 117
  - spaces 46
  - the Sybase set command 58
  - timeout 46
  - timer 46
  - verbose 46, 99, 101
- show db
  - stored procedures 102
  - tables 98
  - triggers, defaults & rules 104
  - verbose 99, 101, 103
  - views 100
- SQL 22, 58, 79, 97, 125, 141
  - executing 39
  - stored procedures 23, 24, 30, 139
  - timing 46
- stored procedures (see SQL) 24
- Sybase 16, 141, 142, 143
  - interfaces file 29
- V
- verbose 46, 99, 101, 103, 111