dBASE IV®

Version 2.0

Programming in dBASE IV

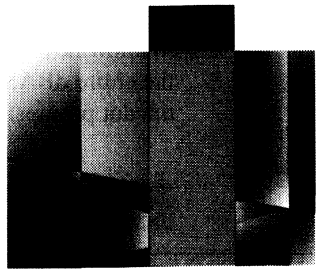
BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001

PRINTED IN IRELAND

10 9 8 7 6 5 4 3 2 1

Contents

Introduction



The first part of this four-section manual is a reference to the dBASE IV® programming language. It describes how to design and write a dBASE® application using the built-in program editor and pseudo-compiler of dBASE IV.

In addition to the programming language, the dBASE IV Control Center provides four design tools: queries, forms, reports, and labels generators. The open-architecture Control Center lets you launch your own custom programs from these design tools. Since version 1.0, these design tools and the Applications Generator have been generating program code by using the *dBASE IV Template Language.*

The second part of this manual explains the dBASE IV template language and how it is used to generate the design surface programs. You can change the program code generated by the design surfaces by altering the templates used by these design surfaces. One change to a template will alter all subsequent programs generated by that template.

The third part of this manual provides an overview of SQL (Structured Query Language), and describes how you can use SQL commands in dBASE IV.

The fourth part of this manual describes some of the ways you can optimize your system, dBASE IV environment, and your dBASE applications for better performance. It also discusses dBASE IV's memory management and high-performance filter optimization techniques.

What You Should Already Know

In order to get the most out of this book, you should be familiar with the dBASE IV Control Center. You should know how to create a database file and build screen forms, reports, labels, and queries (unless you plan to code these yourself). See *Using dBASE IV* for more information on the Control Center.

You should also be familiar with dBASE IV at the dot prompt. That doesn't mean you have to know every dBASE command, but you should be comfortable doing everyday file management tasks at the dot prompt, such as INDEXing a database and LISTing data. Chapter 1 of *Language Reference* provides an overview of the dot prompt.

Here are some things you *don't* need to know before using this book. You don't need to know how to program in any other language. You don't need to know how to use the entire dBASE IV menu system (as long as you can accomplish equivalent tasks at the dot prompt).

NOTE
If you're eager to create programs in dBASE IV and you don't want to bother with programming just yet, you can use the dBASE IV Applications Generator to develop very sophisticated programs without writing any code. See Using dBASE IV *for more information.*

How to Use This Manual

This manual isn't meant to be read from cover to cover. It is a reference book, so use it to look up specific tasks that you want to accomplish.

The book consists largely of *code fragments*, small sections of code lifted out of a larger program. Providing code fragments rather than full programs permits the inclusion of many more examples. Many of the code fragments come from the business application system provided with your sample files. The sample programming code provided contains the complete code for the programs in that application.

To get the most out of this book, refer to *Language Reference* whenever you are not entirely clear about how a command or function used in a code example works.

Printing Conventions

This book uses several printing conventions:

- New items and book titles are printed in italics.
- The names of files, fields, index tags, and procedures begin with a capital letter.
- The names of memory variables, windows, and menus are printed in lowercase letters. In text (as opposed to program code) they appear in italics to distinguish them from other words in the sentence.
- Program code is printed in a special typeface so that the characters align exactly as they do on your screen.
- Variations on a code fragment are shown preceded by the entry *Variation* in italic type.

Special Comment Sections

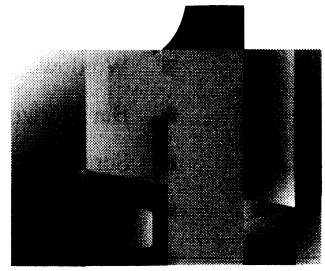
You'll find some information in special comment sections. The icon shown here sets the material in these commentaries apart from other text on the page. (The icon is also the dBASE symbol for adding a comment to a line of code.)

The purpose of these special comment sections is to save you reading time. You can decide whether you want to read a given comment section by a quick glance at its title. If you choose to skip over the material, a single horizontal line at the end of the section tells you where the chapter text begins again.

dBASE IV® Language

Introduction to Programming
Memory Variables
Working with Data Types
Program Architecture
Setting Up the Environment
The Look of Your Application
Designing Menus and Lists
Input: Getting Data from the User
Processing: Ordering the Database File
Processing: Searching for Data
Processing: Relating and Restricting Data
Output: General Considerations
Output: Getting Data from the System
Data Security and Integrity
Compiling, Debugging, and Linking
Low-level File Functions
Design and Data Surface Programs

Introduction to Programming



This chapter provides an introduction to dBASE IV programming. It is primarily intended for the novice programmer. If you have already programmed extensively in the dBASE language or another language, feel free to skip this chapter or skim it.

What This Chapter Covers

The chapter covers the following topics:

- What is dBASE programming?
- A sample program
- Routines and application programs
- Programming conventions
- Basic terminology

What is dBASE Programming?

You already know how to issue dBASE commands at the dot prompt. dBASE programming automates this process. First you create a *command file* or *program* containing the commands you want to execute. Then you *execute* the command file to run the commands contained in it. Programming in the dBASE language saves you from having to type the same set of commands repeatedly at the dot prompt.

dBASE IV programming provides both power and flexibility. Below is a list of some important features of dBASE IV. You may not recognize all the items listed, but as you read this chapter and use the book as a reference, they will become more familiar to you.

- A program compiler
- A program editor
- Memory variables and arrays
- Programming constructs

- Program setup commands and system variables
- Windows and color control
- Bar (horizontal) and pop-up (vertical) menus
- Screen forms and data validation
- Special memo field handling
- Production index files
- A special construct for searching database files
- Multiple child, multiple file relations
- Custom print jobs and formatted reports
- Tools for ensuring data security and integrity
- A program debugger
- Tools for distributing an application

A Sample Program

In this section, you create and run a short dBASE program that displays or prints a customer's name in the Names database file with a specified zip code.

For the sake of simplicity, this sample program assumes 5-digit zip codes and contains minimal error-checking and print or screen formatting. The code examples throughout this book show many ways to enhance a simple program like this one.

dBASE IV is not case sensitive. You can enter commands at the dot prompt in uppercase or lowercase. They are shown in this manual in uppercase for emphasis.

Creating the Sample Program

Open the dBASE IV program editor (or the editor you've installed in Config.db) in one of the following ways:

- From the Control Center, select **<create>** from the **Applications** panel and then choose **dBASE program** from the prompt box that appears.
- From the dot prompt, type MODIFY COMMAND Zip_name and press ↵.

A blank program editor screen appears. The menus across the top of the screen and the status bar across the bottom are similar to those used throughout the menu system (see *Using dBASE IV* for more information).

In the program editor window, type the program shown below exactly as it appears. Press ↵ at the end of each line to advance to the next one. Use the standard navigation and editing keys to correct any errors you make (see *Using dBASE IV* for help with the dBASE IV editor).

NOTE

Your sample files are probably not stored in the c:\dbase\samples directory. Supply the correct directory name in the SET PATH command. Also, omit the commands SET PRINTER ON and SET PRINTER OFF if you want the output to list to your screen instead of printing.

```
* Zip_name.prg
CLEAR ALL
CLEAR
SET STATUS OFF
SET TALK OFF
SET SAFETY OFF
SET PATH TO c:\dbase\samples
mzip = SPACE(5)
USE Names
INDEX ON Zip + Lastname + Firstname TO Zip_name
@ 10,10 SAY "Ready to print address listing for zip code"
@ 12,10 SAY "Enter zip code (press RETURN to cancel): ";
        GET mzip PICTURE "99999"
READ
IF "" = TRIM(mzip)
   RETURN
ENDIF
SEEK mzip
IF .NOT. FOUND()
   ?? CHR(7)
   WAIT "Zip code not found. Press any key."
   RETURN
ENDIF
SET PRINTER ON
? TRIM(Lastname) + ", " + Firstname
? Address
? TRIM(City) + ", " + State, Zip
? Phone
?
SET PRINTER OFF
WAIT
CLOSE ALL
SET TALK ON
SET SAFETY ON
SET STATUS ON
RETURN
* EOP: Zip_name.prg
```

Check your code for accuracy and make any needed changes. Once the program is correct, press **Ctrl-End** to save it. If you accessed the program editor from the Control Center, dBASE IV prompts you for a filename. Enter Zip_name at the prompt.

Running the Sample Program

The program you just saved is now stored in the current directory on your disk. You can run it from either the Control Center or the dot prompt. (Be sure your printer is on if you SET PRINTER ON in your program.)

- From the Control Center, select **Zip_name** from the **Applications** panel and then choose **Run application** from the prompt box that appears. Select **Yes** from the confirmation box.
- From the dot prompt, type DO Zip_name and press ↵.
- From the operating system prompt, where you start dBASE IV, type dBASE Zip_name and press ↵.

When you first execute this program file, dBASE IV compiles the source code (a file with a .prg extension) into executable object code (a file with a .dbo extension) before it runs your program. If TALK is ON, you will see a message indicating the program line numbers as they are compiled.

If the program runs, you will see the screen shown in Figure 1-1. (If it doesn't, see the comment box below.)

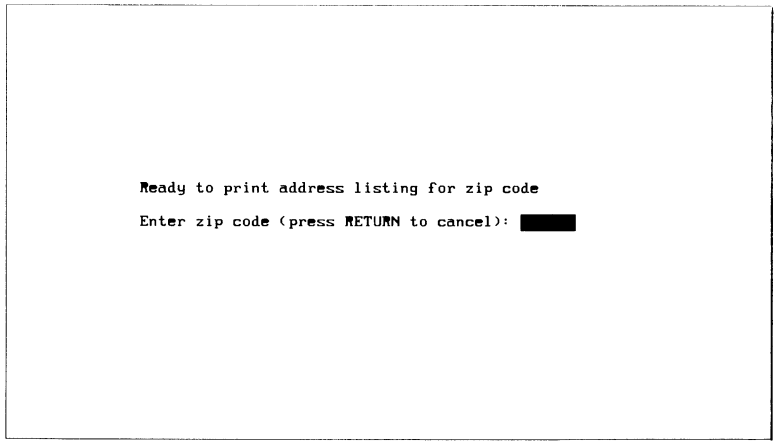


Figure 1-1 Sample program screen

Enter the zip code 20002 at the prompt. A record from the Names database file appears (see Figure 1-2).

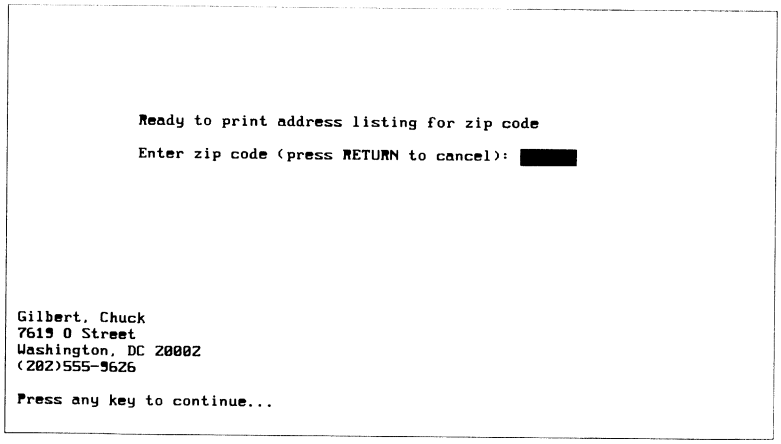


Figure 1-2 Sample program output

Debugging the Sample Program

If the original or modified version of your sample program didn't run, follow the instructions in the Modifying the Sample Program section to display the program code again in the program editor. Then use the list of tips below to locate and correct possible errors. (dBASE IV provides a program *debugger* to help you locate and correct program errors in large or complex programs. See Chapter 15 for more information.)

Here are some typical programming errors:

- Misspelling a command, function,variable name, or filename in the program.
- Omitting parentheses () or quotation marks " ". Be sure to provide both opening and closing delimiters of the same type. Allowed delimiters are single or double quotation marks or square brackets.
- Omitting the beginning or ending command in a programming construct (such as IF...ENDIF or SCAN...ENDSCAN).
- Omitting the semicolon (;) continuation character when a command is longer than one line.
- Omitting the asterisk (*) or the double ampersand at the beginning of a comment line.
- Supplying the wrong PATH to files used in the program.

Modifying the Sample Program

Now modify the sample program. After pressing a key to continue:

- From the Control Center, select **Zip_name** from the **Applications** panel and then choose **Modify application** from the prompt box that appears.
- From the dot prompt, enter MODIFY COMMAND Zip_name.

Insert the SCAN and ENDSCAN commands shown below in italics. Indent by three spaces the lines of code between them. (Surrounding commands are shown for context.)

```
* Zip_name.prg
.
.
.
SET PRINTER ON
SCAN WHILE Zip = mzip
   ? TRIM(Lastname) + ", " + Firstname
   ? Address
   ? TRIM(City) + ", " + State, Zip
   ? Phone
   ?
ENDSCAN
SET PRINTER OFF
WAIT
   .
   .
   .
```

Check your entries. Save and run the modified application as you did before. At the prompt, enter the zip code 20002 again. This time two records print or appear (see Figure 1-3).

```
Enter zip code (press RETURN to cancel): 20002

Gilbert, Chuck
7619 O Street
Washington, DC 20002
(202)555-9626

Rizzo, Ann
20384 Capitol Ave.
Washington, DC 20002
(202)555-3818

Press any key to continue...
```

Figure 1-3 Modified program output

Understanding the Sample Program

Here is a detailed explanation of the sample program:

1. The first line states the name of the program. The asterisk at the beginning of the line tells dBASE IV that this is a comment line and should not be executed.
2. The CLEAR ALL command closes any open dBASE files and releases memory variables. The CLEAR command clears the screen.
3. The four SET commands establish the working environment of the program. They turn off the status bar, suppress system responses to commands, let files be overwritten without a warning message, and tell dBASE IV where to find the Names database file used in the program.
4. The next command assigns the memory variable *mzip* a value of five spaces. The program uses this variable to hold the zip code you enter at the prompt.
5. The USE command opens the Names database file.
6. The INDEX command arranges the records in Names in zip code order. Then it arranges records with the same zip code in last name order, and records with duplicate last names in first name order.
7. The @...SAY command displays the quoted prompt at row 10, column 10 on the screen.
8. The @...SAY...GET command displays the quoted prompt at row 12, column 10. The GET displays the current value of *mzip* and awaits your entry of a zip code. The PICTURE clause limits your entry to five numbers. The semicolon (;) at the end of the first line tells dBASE IV that the command continues on the next line.
9. The READ command activates the GET, which reads your entry into the variable *mzip*.
10. The three lines beginning with IF " " = TRIM(mzip) and ending with ENDIF terminate the program if you press ↵ with no entry at the prompt.
11. The SEEK command looks for the first record in Names with a zip code matching the one you entered. Since the records are in zip code order (step 6 above), any other records with this zip code follow directly.
12. The IF .NOT. FOUND( )...ENDIF lines test to see if a record was found with the SEEK command. If no records are found matching the zip code you entered, the ?? CHR(7) command sounds a warning bell and the WAIT command displays a message that no zip codes were found. After you acknowledge the message, control returns to the dot prompt or Control Center, wherever you started Zip_name.
13. The SET PRINTER ON command instructs dBASE IV to direct output to the printer. (You might have omitted this line.)

14. The SCAN command (inserted when you modified the program) tells dBASE IV to step through the database file, performing the commands between the SCAN and the ENDSCAN on each record as long as the zip code matches the one you entered.

15. The first ? command prints the last name and first name separated by a comma and a space. The TRIM() function trims trailing blank spaces after the last name to remove gaps.

16. The next three ? commands print the contents of the Address, City, State, Zip, and Phone fields. The last ? command prints a blank line after each listing, so the records don't run together in the output.

17. The SET PRINTER OFF command stops directing output to the printer.

18. The WAIT command causes the system to pause and display the message **Press any key to continue...**. Without this command, output to the screen would scroll too quickly to read.

19. The CLOSE ALL and SET commands restore the default working environment.

20. The RETURN command returns control to the dot prompt or the Control Center, depending on where you started the program.

21. The final comment line indicates the end of the program.

Note the significance of the SCAN...ENDSCAN construct in the sample program. Before you added these commands, the program printed only the first record with a zip code of 20002. Introducing the SCAN caused the program to look for additional records with that zip code.

Routines and Application Programs

The sample program you just created could be a subprogram or *routine* in a larger program. You can create *application programs* (large programs that accomplish many tasks) by combining smaller routines. This book provides code examples for typical routines that might go into a complete application.

Figure 1-4 shows how the sample Zip_name program might fit into a larger application. First, the application sets up the program environment and displays the main menu. Next, each option of the main menu activates a submenu. Here, the **List records** option opens the **List Records** submenu.

Each submenu option performs specific program tasks. In this case, when the user selects **Zip code order**, a prompt asks for the output device (screen or printer). If the user chooses **P**, the program SETs the PRINTER ON before running the Zip_name program.

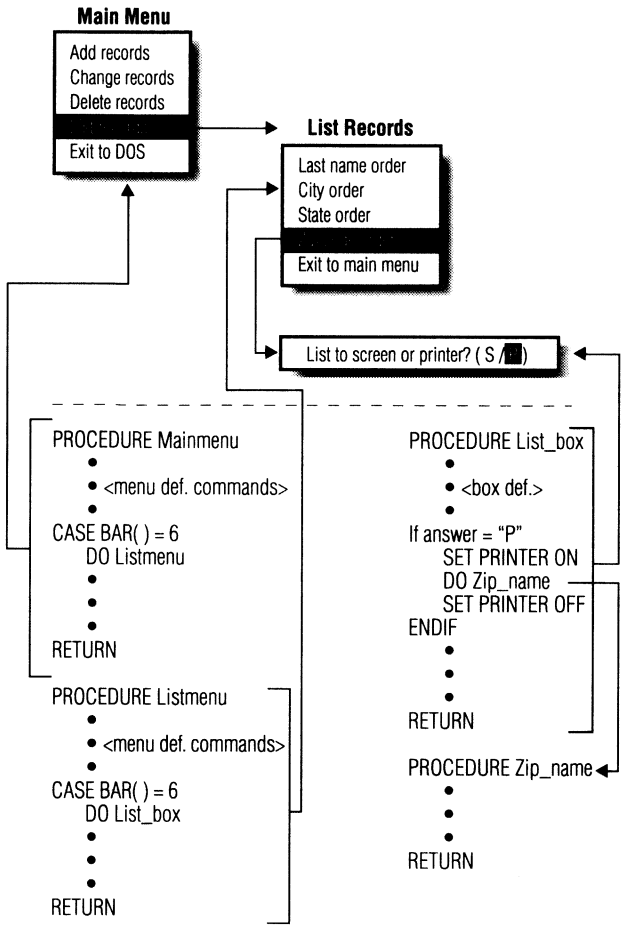


Figure 1-4 Hypothetical application containing Zip_name

Programming Conventions

Your programs should be easy to read so other programmers can maintain them, and so you can recall details of programs you wrote in the past. Adhering to the suggested conventions listed here will make your dBASE programs easier to read and maintain.

- Begin each program with a *header* that includes the program's name, author, and date of creation (see the Program Headers section in Chapter 5).
- Assign descriptive names to files, fields, memory variables, and other objects. A name like X will mean nothing to others maintaining your programs, or perhaps even to you in time. Don't use dBASE command and function names for objects.

- Comment your code liberally. Precede comments by an asterisk (*) if the comment is on a separate line, or a double ampersand (&&) if the comment is on the same line as the code. (To save space, the code fragments in this book are not heavily commented. The application code in the sample programming code is commented as recommended here.)
- Indent by three spaces commands nested within programming constructs such as IF...ENDIF, DO CASE...ENDCASE, and SCAN...ENDSCAN. Chapter 4 explains programming constructs. Exception: Do not indent text in a TEXT...ENDTEXT construct, as this text is reproduced exactly as it appears in your code.
- Optionally, start each new clause of the @...SAY...GET command on a new line, aligned under the previous clause.
- Type a semicolon (;) at the end of a command line if you want to continue the command on the next line. dBASE IV allows up to 1,024 characters per command line, but you might still want to limit lines of code to 80 characters. All the code is then visible on the editor screen.

Break a long character string as follows:

```
?? "Type the first letter of an item name to select it, " +;
   "or highlight and press RETURN."
```

NOTE
SQL uses the semicolon to indicate the end of a command. When you SET SQL ON, you cannot use the semicolon as a continuation character. See the third section of this manual for SQL programming information.

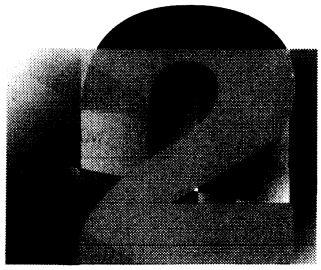
Basic Terminology

In general, new terms are defined when they first appear in this manual. However, the following terms are used throughout the book and so are defined here.

A *program* is a set of instructions that enables a computer to solve a problem. An *application* (also called an *application program* or *application system)* is a program or set of programs that performs a set of tasks. While all applications are programs, not all programs are applications. A *sub-application* is an application within a larger application.

A *programmer* is an individual who writes programs. A *developer* writes applications to be marketed or distributed. A *user* or *end-user* is an individual for whom the program or application is intended. The programmer and user are one and the same if you're writing programs for your own use.

Memory Variables



dBASE IV stores temporary or working data in *memory variables.* You should understand how memory variables work before you begin programming in the dBASE language.

What This Chapter Covers

This chapter provides a brief overview of memory variables and memory variable arrays. It discusses the following topics:

- What memory variables are
- Naming memory variables
- Initializing memory variables and memory variable arrays
- Using memory variables and memory variable arrays
- Public and private variables
- Releasing memory variables and arrays
- Using memory files
- Filename and macro substitution

About Memory Variables

A memory variable is a temporary storage location for data. The data in a memory variable is stored in memory, with a name that references its location (see Figure 2-1).

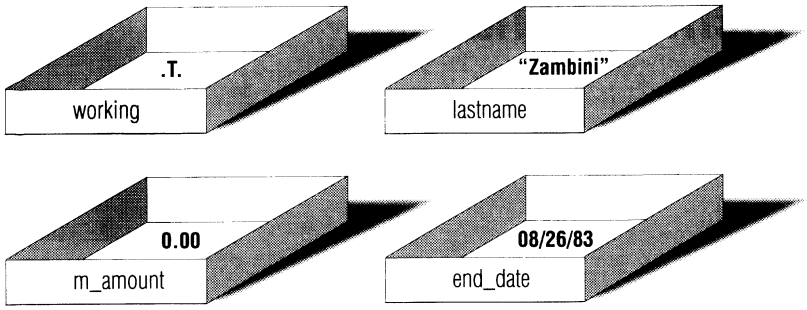


Figure 2-1 Memory variables

Memory variables can contain character strings, numbers, dates, or logical values (.T. or .F.). There are no memo type memory variables.

Programs use memory variables for many purposes. For example, a memory variable can hold user-entered data for validation, working data for computation, or a frequently used screen prompt for display.

You refer to a memory variable by its name, not its contents. You can vary the contents without changing the name. This means that you can use the same memory variable repeatedly throughout a program.

NOTE
In regular text throughout this book, the names of memory variables are shown in italics.

Naming Memory Variables

The name of a memory variable may be up to ten characters long. It may include letters, digits, and underscores, but no spaces. Start the name with a letter, and avoid using dBASE reserved words (command and function names).

You should establish conventions for naming memory variables. You will find that your own conventions develop to complement your programming style as you write more programs. Following are some conventions that are common in dBASE programs:

- Choose a name that describes what the memory variable does, such as *cost* to describe a variable holding cost data.
- If the variable corresponds to a field in the database file, give it a similar name.

- Start memory variable names with the letter *m* or the characters *m_* to prevent confusion with an identical field name. If you need all ten characters for variable names, use the *m->* pointer instead. For example, the field Qty_onhand would become the variable *m->qty_onhand.*
- Indicate the data type of the variable and whether it is global or local in its name. For example, the name *lc_title* suggests a local character variable holding a title. (See the Public and Private Variables section for a discussion of global and local variables.)

Initializing Memory Variables

To *initialize* a memory variable, assign a value to it. dBASE IV automatically gives the variable the data type of the data that you put in it. Initializing a memory variable overwrites existing PUBLIC memory variables with the same name. Therefore, to modify a memory variable, just reassign it. The data type of the variable will reflect the new data. PRIVATE memory variables are not written over.

You can initialize memory variables in several ways:

- Using the STORE command: *STORE "Exit? Y/N" TO prompt1*
- Using the equivalent assignment statement: *prompt1 = "Exit? Y/N"*

Use STORE to initialize several memory variables at once:

```
STORE 0.00 TO subtotal, mtotal, cash
```

Other ways of using the equivalent assignment are:

```
. CLERALL
. INPUT "ENTER 1:" TO x && x will be created
. ? x
   1
```

Another example of assignment is:

```
. USE FILE
. SUM MATHFIELD to xx && xx will be created
. ? xx
```

Table 2-1 shows some typical memory variable definitions. Note that character strings are delimited, and dates are converted to date type data with the CTOD() function or the date literal delimiter, {}.

CTOD() takes a character string argument. The curly braces take a literal date value.

Table 2-1 Initializing memory variables

Type	Example
Character	mfirstname = "Weston" prompt1 = "That's incorrect ... try again" STORE SPACE(20) TO mlastname STORE " " TO choice, answer
Numeric	part_qty = 20 m_cost = 0.00 STORE 0.00 TO subtotal, total, cash
Date	today = DATE() birthday = CTOD("11/13/85") date_trans = {}
Logical	finished = .T. STORE .F. TO erased, not_valid

Initializing Memory Variable Arrays

A *memory variable array* is a matrix of memory variables arranged in rows and columns. Memory variable arrays can be either one- or two-dimensional. A one-dimensional array has only one row, whereas two-dimensional arrays have two or more rows. The total number of *elements* in an array is equal to the number of rows multiplied by the number of columns (see Figure 2-2). An array can have up to 1,170 elements.

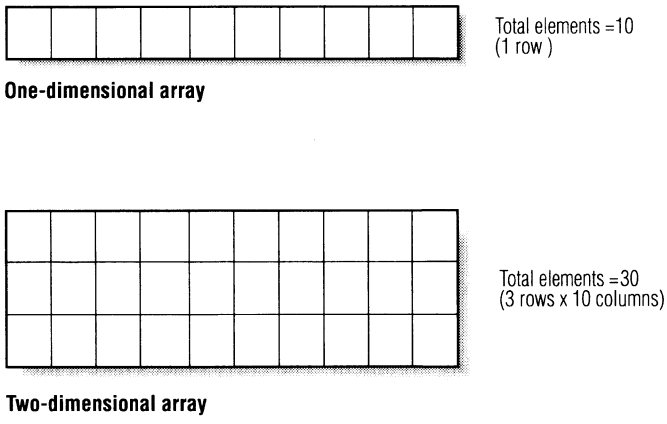


Figure 2-2 One- and two-dimensional arrays

To initialize a memory variable array, DECLARE the array structure and then assign values to the elements. Array names follow the same naming conventions as memory variables.

Reference an array element with the name of the array followed by the coordinates in square brackets. One-dimensional arrays have one coordinate (row); two-dimensional arrays have two coordinates (row and column) separated with a comma. A row is like a .dbf record. A column is like a .dbf field.

Here's how to declare a two-dimensional array with 33 rows (one for each record in the Goods database file):

```
DECLARE  cost_aray[33,2]
```

You can STORE different types of data TO different elements in a single array. Each element has a data type, but the array as a whole does not.

The following commands assign a part number and cost from the first record to the first two elements of the *cost_aray* array:

```
USE  Goods  ORDER  Vendor_id
cost_aray[1,1]  =  Part_id
cost_aray[1,2]  =  Cost
```

You can STORE a memory variable to an array, and also STORE the value of one array element to another.

Using Memory Variables and Arrays in Programs

Table 2-2 shows some of the typical ways in which memory variables are used in programs.

Table 2-2 Using memory variables in programs

Purpose	Example
Handle screen messages	prn_warn = "Printer not ready" mbell = CHR(7) @ 10,10 SAY mbell + prn_warn
Hold data for validation and entry into .dbf file	mpart_id = SPACE(10) @ 10,10 SAY "Enter part no." GET mpart_id READ DO Chk_dup WITH mpart_id
Hold data for searching	mpart_id = SPACE(10) @ 10,10 SAY "Enter part no." GET mpart_id READ SEEK mpart_id

(continued)

Table 2-2 Using memory variables in programs *(continued)*

Purpose	Example
Hold data for processing	maxx = 100 m->qty_2order=0 @ 10,10 SAY "Enter order quantity"; GET m->qty_2order READ IF m->qty_2order > maxx ? "Quantity ordered is too large" ENDIF
Hold data for computation	STORE 0.00 TO amt_lstbil, amt_lst_pd @ 10,10 SAY "Enter amount of last bill"; GET amt_lstbil PICTURE "99999.99" @ 12,10 SAY "Enter amount last paid"; GET amt_lst_pd PICTURE "99999.99" READ oldbalance = amt_lstbil - amt_lst_pd ? oldbalance
Hold data for report filters	USE Employee m->date_hired = { } @ 10,10 SAY "Enter earliest hire date"; GET m->date_hired READ SET FILTER TO Date_hired >=m->date_hired GO TOP LIST OFF NEXT 10 Lastname, Date_hired

The following example stores the part number and cost for each record in the Goods database file to an array and displays their values:

```
SET TALK OFF
USE Goods ORDER Vendor_id
DECLARE  cost_aray[33,2]
rrow = 1
SCAN
    cost_aray[rrow,1] = Part_id
    cost_aray[rrow,2] = Cost
    ? "Part no.", cost_aray[rrow,1], "    Cost", cost_aray[rrow,2]
   rrow = rrow + 1
ENDSCAN
SET TALK ON
```

Public and Private Variables

dBASE IV provides two classes of memory variables: public and private. A *public* variable is available in all program modules, no matter where you initialize it. A *private* variable operates only in the module where you initialize it and all subordinate modules.

In Figure 2-3, *balance* is available in all programs, even though it's declared PUBLIC in a subprogram. However, *m_amount* is available only in the program where it is initialized and in all dependent subprograms. The *m_cash* variable is available in one subprogram only.

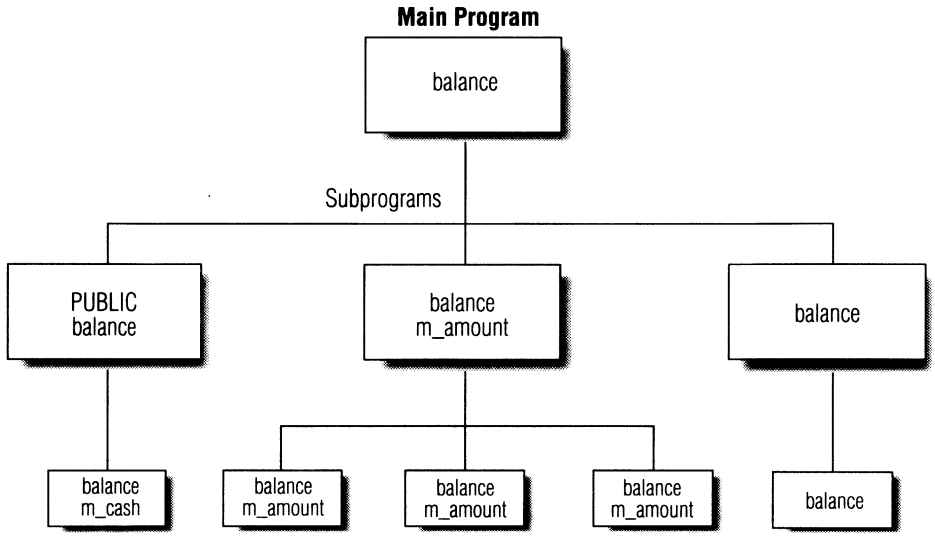


Figure 2-3 Public and private variables

In dBASE IV programs, memory variables are private unless you declare them public. So if you initialize a variable in a subprogram, it will be private to that subprogram and all programs the subprogram calls. dBASE IV releases the variable when the subprogram RETURNs control to the calling program.

Arrays are public by default. To create a private array, you must make the array name private before you declare the array:

```
PRIVATE  cost_aray
DECLARE  cost_aray[33,2]
```

You can hide a variable from a higher-level variable of the same name by explicitly declaring it PRIVATE. This lets you use standard names for variables that do the same thing throughout your program. For example, in Figure 2-4, you can use the private *balance* variable in the Sub1 subroutine and in any subprogram the subroutine calls. When program control RETURNs to the original module, the variable automatically reclaims its original value of 350.00.

Here is the code to initialize the PRIVATE *balance* variable:

```
PRIVATE  balance
balance  =  175.00
```

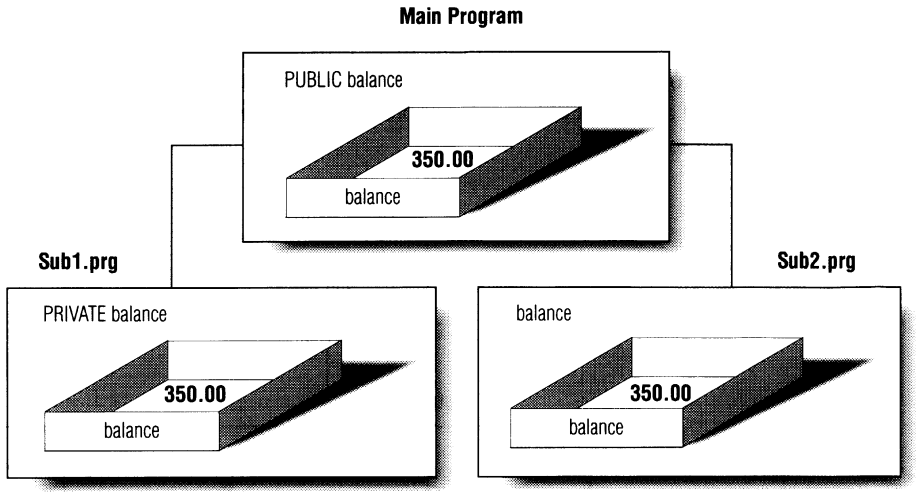


Figure 2-4 Hiding a public variable

If you want a variable to be available to the entire program, you must declare it PUBLIC unless you initialize it in the main (top-level) program module. dBASE IV never clears public variables unless explicitly told to do so (see the following section).

Here's how to declare *balance* PUBLIC:

```
PUBLIC  balance
balance  =  350.00
```

Releasing Memory Variables and Arrays

dBASE IV clears all private memory variables automatically when a program finishes, or when the subprogram where they were initialized terminates. However, public variables must be explicitly CLEARed or RELEASEd.

Use the RELEASE command to erase specified memory variables and arrays from memory. Use CLEAR ALL or CLEAR MEMORY to erase all variables at once. QUITting dBASE IV automatically erases all variables from memory.

Table 2-3 shows different ways to erase memory variables from memory. Use the same commands to erase memory variable arrays.

Table 2-3 Erasing memory variables

Operation	Example
Erase a single memory variable	RELEASE mcost
Erase more than one variable	RELEASE mcost, mbalance
Erase all variables with names starting with m	RELEASE ALL LIKE m*
Erase all variables except with names starting with m	RELEASE ALL EXCEPT m*
Erase all public variables	CLEAR MEMORY or CLEAR ALL

Note that in a program, RELEASE ALL deletes only private memory variables created in the current procedure or a lower level procedure.

WARNING
Because you might inadvertently clear variables that you want to keep, be careful when using CLEAR MEMORY or CLEAR ALL.

Using Memory Files

If you plan to reuse memory variables, SAVE them to a memory file. Later, you can RESTORE them to active memory from the memory file. dBASE IV gives all memory files the extension .mem. You can give a different extension to memory variable files when you save them.

The following dot prompt commands initialize several memory variables and a memory variable array, and then SAVE the definitions to the file Setup.mem.

```
. STORE SPACE(20) TO mlastname, mfirstname
. STORE " " TO cust_id, emp_id, part_id, po_number
. part_qty = 0
. date_trans = {}
. invoiced = .F.
. DECLARE vend_aray[2,2]
. STORE 0 TO vend_aray[1,1], vend_aray[1,2],;
            vend_aray[2,1], vend_aray[2,2]
. SAVE TO Setup
```

Use the RESTORE command to read the contents of a memory file back into memory. Both individual variables and memory variable arrays will be RESTOREd to memory as defined.

When you RESTORE memory variables from a file, you automatically RELEASE all variables currently in memory unless you instruct dBASE IV to retain them with the ADDITIVE clause. To RESTORE the contents of Setup.mem but not RELEASE any other variables in memory, enter the command:

```
RESTORE FROM Setup ADDITIVE
```

dBASE IV does not save the public or private status of a variable to the memory file. If you RESTORE a memory file within a program, the variables come back private (the default status). Use the PUBLIC command before the RESTORE command to make them public:

```
PUBLIC mlastname, mfirstname
RESTORE FROM Setup ADDITIVE
```

You can add variables to an existing memory file and delete variables from it. At the dot prompt, RESTORE the memory file. Then initialize any additional memory variables or arrays into memory, and RELEASE any variables you no longer want. Finally, SAVE back to the memory file. The SAVEd file will reflect your changes. Use this technique to revise memory files during program development.

Filename Substitution

Wherever a dBASE command expects a filename, you can substitute a memory variable enclosed in parentheses or an expression containing at least one operator or function. This is known as *filename substitution.* For example, you can READ a filename into a memory variable and then USE the variable:

```
filename = SPACE(8)
@ 10,10 SAY "Enter filename: " GET filename
READ
USE (filename)
```

You can also store an index tag or filename in a memory variable:

```
filename = "Employee"
mtag = "Emp_id"
USE (filename) ORDER (mtag)
```

You can even combine a filename expression with a literal string. Suppose your system has separate archive files for each year: Orders88, Orders89, and so on. You can COPY TO the appropriate file as follows:

```
myear = RIGHT(DTOC(DATE()),2)
COPY TO "orders" + myear
```

Macro Substitution

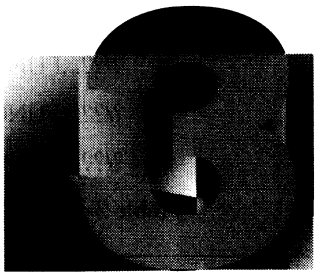
While filename substitution is the recommended method wherever dBASE IV expects a filename, use *macro substitution (&)* to supply more complex expressions. Use a period to signal the end of a macro. For example, you can store a directory path to a memory variable and then use the variable in a SET PATH statement:

```
path  =  "c:\dbase\samples"
SET  PATH  TO  &path.
```

As with filename substitution, you can concatenate macro expressions:

```
path1  =  "c:\dbase"
path2  =  "\samples"
SET PATH TO &path1.&path2.
```

Working with Data Types



dBASE IV provides six data types: character, fixed, floating, date, memo, and logical. This chapter contains summaries of how to format and manipulate the different types of data, as well as examples of how to use them in program code.

What This Chapter Covers

This chapter covers the following topics:

- Handling character and numeric data
- Handling date and time data
- Handling memo field data
- Handling logical data
- Converting data types

See Chapters 2 and 4 of *Language Reference* for more information on the commands and functions cited in this chapter.

Handling Character Data

Much of the data processed by your programs will be character data. Remember that character data can consist of numbers as well as character strings, as long as you don't use the numbers in mathematical computations.

Use the following commands and functions to format character data.

- UPPER() — Changes to uppercase
- LOWER() — Changes to lowercase
- LTRIM() — Trims leading blanks
- RTRIM() — Trims trailing blanks
- TRIM() — Trims all blanks

- ?/??...FUNCTION "T" — Trims all blanks
- ?/??...FUNCTION "I" — Centers a string
- ?/??...FUNCTION "J" — Right-justifies a string
- ?/??...FUNCTION "B" — Left-justifies a string

Table 3-1 shows some common ways to manipulate character data in a program.

Table 3-1 Manipulating character data

Task	Example
Measure length of string	? LEN(Part_name)
Find position in substring	? AT("Tue","Monday,Tuesday,...")
Extract a substring	? SUBSTR(Part_id,7,4)
Extract from left	? LEFT(Part_name,9)
Extract from right	? RIGHT(mpart_id,4)
Insert one string into another	? STUFF(mterms,8,2,mdays)
Repeat a string	? REPLICATE("*",16)
Concatenate strings	? mfirstname + mlastname
Compare strings by value	? mpart_id1 = mpart_id2
Compare strings using wildcards	LIST Part_id FOR; LIKE("*-700-??30",Part_id)

dBASE IV compares each character until it runs out of characters on the right side of the relational operator. For instance, this comparison:

```
. ? "abcd" = "abc"
```

gives a **.T.** (true) response. But this comparison:

```
. ? "abc" = "abcd"
```

gives a **.F.** (false) response. If you SET EXACT ON, however, dBASE IV compares the strings exactly and both comparisons return a .F.

The following code examples show how to use these data manipulation capabilities in a programming environment. The first brief routine does an error procedure if a string of spaces is encountered:

```
IF "" = TRIM(entry)
   DO Error
ENDIF
```

The example below uses both the – (minus) and + (plus) concatenation operators to print a city, state, and zip code in the form *Contoocook, NH 03229.* Note that the – (minus) operator moves any blank spaces filling out the City field to the right of *mstate.* This aligns the zip codes vertically.

```
mstate = ", " + State + "  "
? City - mstate + Zip
```

You can extract a substring without knowing its position in the primary string. In the following routine, the AT() function identifies the starting position of *mpart* in the Part_name field. LEN() determines the length of the substring.

```
mpart = "sofa"
?  SUBSTR(Part_name,  AT(mpart,Part_name),LEN(mpart))
```

The following example uses AT(), LEFT(), and SUBSTR() to print the items in a list. The variable *mfilelist* contains a list of filenames separated by commas. The first line within the DO WHILE loop prints the first word in the list, up to but not including the first comma. The second line stores the rest of the string back to *mfilelist.* Note that the AT() functions within the DO WHILE loop determine the substring length and starting position, respectively. The DO WHILE condition causes this process to continue as long as there are commas in *mfilelist,* that is, until only one word remains. Then the loop ends and the program prints the last filename in the list.

```
mfilelist  =  "Acct_rec,Client,Employee,Orders,Vendors"
DO  WHILE  "," $ mfilelist
     ? LEFT(mfilelist,  AT(",",mfilelist)-1)
     mfilelist  =  SUBSTR(mfilelist,  AT(",",mfilelist)+1)
ENDDO
?  mfilelist
```

Handling Numeric Data

This section explains how to format numbers and perform numeric calculations ina program.

Numeric Data Types

dBASE IV supports both *Binary Coded Decimal* (BCD, type N) and *floating point* (type F) numbers. This represents a change from earlier versions of the dBASE product, which supported floating point numbers only.

BCD numbers contain a decimal representation of the number. You normally use BCD numbers for accounting applications. The use of BCD numbers eliminates the problems with numeric comparison and rounding errors that occur with type F numbers. This ensures that balances total properly.

Use CREATE and MODIFY STRUCTURE to define fixed and floating point numeric fields. The data type of a numeric memory variable depends upon how you create it. If you create it from a field, the type will match the field type. If you create it from a numeric constant, the variable will be type N. If you create it by a command or function, the type will depend on the command or function.

Use FLOAT(<BCD expression>) to convert a BCD number to floating point, and FIXED(<floating point expression>) to convert a floating point number to BCD. When you combine both BCD and floating point numbers in an expression, dBASE IV automatically converts the BCD numbers to floating point.

Formatting and Manipulating Numbers

Here is a brief summary of the commands, functions, and PICTURE templates and FUNCTIONs that format numeric data. See *Language Reference* for more information on the individual commands, functions, or options.

- To round off numbers: ROUND()
- To show a number in exponential notation: FUNCTION "^"
- To limit display to digits, signs, and blanks: PICTURE "999" or PICTURE "###" (actual number of 9s or # signs varies)
- To show zero values as a blank string: FUNCTION "Z"
- To show leading zeros as zeros, dollar signs, or asterisks: PICTURE "L", PICTURE "$", or PICTURE "*"
- To show a number in standard currency format: FUNCTION "$"
- To specify and position the currency symbol: SET CURRENCY TO and SET CURRENCY LEFT/right
- To show CR after positive number, DB after negative: FUNCTION "C" and FUNCTION "X", respectively
- To show negative numbers in parentheses: FUNCTION "("
- To specify desired decimal symbol and number of decimals: SET POINT TO and SET DECIMALS TO
- To specify separator character: SET SEPARATOR TO
- To center, right-justify, or left-justify data: FUNCTION "I", FUNCTION "J", and FUNCTION "B", respectively

Table 3-2 shows how to manipulate numbers for use in programs. This table shows only the statistical functions. dBASE IV also provides arithmetic, trigonometric, and financial functions. See Chapter 1 of *Language Reference* for more detail, and for an explanation of operator precedence.

Table 3-2 Manipulating numeric data

Task	**Example**
Compute arithmetic expression	? (3*PI())/4
Count items	CALCULATE CNT() TO mcount or COUNT TO mcount
Sum values	CALCULATE SUM(mcost*mquantity) TO mtotal or SUM mcost*mquantity TO mtotal
Average values	CALCULATE AVG(Cost) TO mavg_cost or AVERAGE Cost TO mavg_cost
Compute standard deviation and variance	CALCULATE STD(mcost) TO mstd_cost and CALCULATE VAR(mcost) TO mvar_cost
Compute minimum value*	CALCULATE MIN(Cost) TO mmin_cost or ? MIN(mdate1,mdate2)
Compute maximum value*	CALCULATE MAX(Date_trans) TO mmax_date or ? MAX(mhired,mfired)

* MIN() and MAX() with CALCULATE return the smallest or largest value in the named field. As stand-alone functions, they return the smaller or larger of two values.

CALCULATE can carry out more than one operation in a single pass through the database file. For example:

```
CALCULATE AVG(Cost), STD(Cost) TO mavg_cost, mstd_cost
```

The following example counts how many orders have been placed by a particular customer. After arranging the Orders database file by Cust_id, the routine GETs a customer number from the user. It SEEKs the entry in the file, and COUNTs how many orders (records) the customer has made. It also CALCULATEs the average quantity of parts ordered by the customer.

```
STORE 0 TO morder_cnt, mavg_quant
mcust = SPACE(6)
INDEX ON Cust_id TAG Cust_id
USE Orders ORDER Cust_id
@ 10,10 SAY "Enter customer ID: " GET mcust
READ
SEEK mcust
CALCULATE CNT(), AVG(Part_qty) TO morder_cnt, mavg_quant;
          WHILE mcust = Cust_id
```

The next example returns the percent difference, *m_perc,* between two numbers. Using the IIF() function, the routine returns **N/A** if *m_old* has a 0 value (to avoid dividing by 0). Otherwise, it computes the percent difference in the standard manner, converts it to a character string, and displays it with a % symbol.

```
m_perc = IIF(m_old = 0,"N/A", STR(((mnew - m_old)/m_old)*100,5,0)) + "%"
```

Handling Date Data

Although it lets you *display* dates in a variety of formats, dBASE IV *processes* date fields and date type memory variables as special numbers. The following commands and functions format date type data:

- To specify international date format: SET DATE TO
- To display 4-digit year: SET CENTURY ON
- To define separator character: SET MARK TO
- To return DD Mon YY or Mon DD, YY: DMY() or MDY()
- To return day of week by number or name: DOW() or CDOW()
- To return day of month by number: DAY()
- To return month by number or name: MONTH() or CMONTH()
- To return year: YEAR()

dBASE IV considers Sunday the first day of the week.

Table 3-3 shows how to manipulate date data in programs.

Table 3-3 Manipulating date data

Task	Example
Return system date	? DATE()
Find earliest of two dates	? MIN(mdate1,mdate2)
Find latest of two dates	? MAX(mdate1,mdate2)
Compare dates	? date1 < date2
Check for blank date	? {} = mdate
Check for non-blank date	? .NOT. {} = mdate
Add number of days to date	? DATE() + 45
Subtract number of days from date	? DATE() – 30
Find days between two dates	? mdate1 – mdate2

You can use the LIKE() function to COUNT the number of records in a given month. For example, the following routine COUNTs the number of March records in the Orders database file:

```
USE Orders
COUNT FOR LIKE("03/??/88",DTOC(Date_trans)) TO March_cnt
```

The next example initializes *mdiff* with the difference between today's date and the transaction date. The routine issues a reminder letter if the account is between 30 and 45 days past due, and a stronger letter if the account is 45 or more days past due.

```
mdiff = DATE() - Date_trans
DO CASE
   CASE mdiff > 30 .AND. mdiff < 45
      DO Dunlettr
   CASE mdiff >= 45
      DO Nasty
ENDCASE
```

Note that dBASE IV stores fractions of days. This may be useful for some applications, but can also cause some comparisons to fail unexpectedly. For example:

```
. indate = {07/01/1988}
07/01/88
. outdate = indate + .25
07/01/88
. ? indate = outdate
.F.
```

While *indate* and *outdate* are on the same day, dBASE IV internally stores a value for *outdate* that is 1/4 day greater than the value of *indate*.

If you don't want differences of less than a day to show up in a comparison, convert the date type data to character type before comparing:

```
. ? DTOC(indate) = DTOC(outdate)
.T.
```

Handling Time Data

While there is no time data type, you can use the TIME() function, SET HOURS, and SET CLOCK to format time data.

- To return system time: ? TIME()
- To specify 12- or 24-hour format for the clock display: SET HOURS TO [12/24]
- To display and position clock: SET CLOCK ON and SET CLOCK TO

TIME() always returns a character string in 24-hour format. You can store the current time in a memory variable:

```
mnow = TIME()
```

You can then split *mnow* into substrings if you want to use only part of the time. For example, if the time is 2:30 and 43 seconds (14:30:43), the instruction

```
mnow = LEFT(mnow,5)
```

puts the string 14:30 into *mnow*.

Validating time data entered by the user requires a short routine like the following. If you're using a 24-hour clock, change 12 to 24.

```
mtime = SPACE(5)
@ 10,10 GET mtime PICTURE "99:99" VALID;
     VAL(LEFT(mtime,2)) >= 0 .AND. VAL(LEFT(mtime,2)) <= 12;
     .AND. VAL(RIGHT(mtime,2)) >= 0 .AND. VAL(RIGHT(mtime,2)) < 60;
     ERROR "Enter hours from 0 to 12, minutes from 0 to 59"
READ
```

Handling Memo Field Data

Memo fields are actually character data, but dBASE IV maintains them in a separate database text file (.dbt).

Memo fields can contain up to 64K. If your application requires large memo fields, increase the block size using SET MBLOCK or the MBLOCK command in Config.db (see Chapter 2 of *Getting Started with dBASE IV* for more information).

You can format memo fields using the following commands and system memory variables. (See Chapter 5 of *Language Reference* for information on the system variables.)

- To change display width: SET MEMOWIDTH
- To display in a window: @...SAY...GET WINDOW
- To change left margin: _lmargin
- To change right margin: _rmargin

Table 3-4 shows how to work with memo fields in programs. Note that you can use APPEND MEMO to read the contents of any file, even a program object (.dbo) file, into a memo field. Also, REPLACE converts character data to memo fields and vice versa. In converting memo fields to character fields, REPLACE truncates the data to fit the assigned field width.

Table 3-4 Manipulating memo field data

Task	**Example**
Find length of field	? LEN(Comment)
Find where string starts in memo field	? AT("Cost = $615.00",Comment)
Extract string	? SUBSTR(Comment,12,5)
Extract string from left	? LEFT(Comment,10)
Extract string from right	? RIGHT(Comment,10)
Append text from file	APPEND MEMO Comment FROM Sample
Overwrite text from file	APPEND MEMO Comment FROM Sample OVERWRITE
Replace text in field	REPLACE Note WITH Note1
Add text to field	REPLACE Note WITH Note1 ADDITIVE
Copy memo field elsewhere	COPY MEMO Comment TO Sample
Copy without overwrite	COPY MEMO Comment TO Sample ADDITIVE
Display/print field	LIST Comment or ? Comment
Return number of lines	? MEMLINES(Comment)
Return specific line	? MLINE(Comment,2)

The following hypothetical code produces the output shown below it:

```
SET MEMOWIDTH TO 20
? Merchant
?
? MEMLINES(Merchant)
?
? MLINE(Merchant,3)

The quality of mercy
is not strained. It
falleth as a gentle
rain from Heaven...

4

falleth as a gentle
```

```
SET MEMOWIDTH TO 30
? Merchant
?
? MEMLINES(Merchant)
?
? MLINE(Merchant,3)

The quality of mercy is not
strained. It falleth as a
gentle rain from Heaven...

3

gentle rain from Heaven...
```

You can print the line containing the phrase *gentle rain* even if you don't know which line it is. The next routine, which works with any multiple-word phrase, first assigns the width of the memo field to *mwidth.* Then it uses AT() to find *mposition,* the starting position of the phrase.

Next, the routine computes *m_line,* the line where the phrase starts, by dividing *mposition* by *mwidth.* Since *mwidth* is 35 characters, the first 35 characters of the memo field fall on line 1, characters 36 through 70 on line 2, and so on. The phrase *gentle rain* starts at character 55, which falls on line 2. But note that 55 divided by 35 is 1.57. The routine uses the CEILING() function to increase the line number to 2. This function increments *m_line* whenever *mposition/mwidth* is fractional. Finally, the routine prints *m_line.*

```
mwidth = 35
SET MEMOWIDTH TO mwidth
msearch = "gentle rain"
mposition = AT(msearch,Merchant)
m_line = CEILING(mposition/mwidth)
? MLINE(Merchant,m_line)
```

Variation — When the Phrase Wraps on Two Lines

If a phrase wraps at the right margin, the previous routine only prints the words on the first line. To print the entire wrapped phrase, add the following DO WHILE loop to the routine. For each pair of words, this code uses the RIGHT() function to define *msearch* as the substring from the end of the phrase back to the beginning of the second word. Then it sets *mposition* as the beginning of *msearch.* If *m_line* (calculated as before) has a different value, the routine prints this new line as well. The loop continues as long as additional words, with spaces between them, remain in the phrase.

```
DO WHILE " " $ msearch
   moldline = m_line
   msearch = RIGHT(msearch,LEN(msearch) - AT(" ",msearch))
   mposition = AT(msearch,Merchant)
   m_line = CEILING(mposition/mwidth)
   IF m_line <> moldline
      ? MLINE(Merchant,m_line)
   ENDIF
ENDDO
```

Handling Logical Data

Logical data can have only two values, .T. (true) or .F. (false). Thus, logical variables are handy for controlling program flow or checking the status of conditions.

Logical data is used:

- In a positive condition, such as *LIST FOR invoiced*
- In a negative condition, such as *LIST FOR .NOT. invoiced*
- To form complex expressions, such as *IF .NOT. (EOF() .AND. FOUND())*

Because logical fields are Boolean (true or false), the field *is* the condition. Thus, you don't need to supply an expression. You can simply use *FOR Exempt* rather than *FOR Exempt = .T.*, although either form will work.

You can compare two logical fields, such as *FOR Exempt = Status.* dBASE IV evaluates the expression and returns another logical value (.T. or .F.) based on the comparison.

TIP

Use string equivalents instead of .T. and .F., which might confuse users. For example, you can make .T. appear as Y and .F. as N:

```
@ 10,10 SAY "Print now? (Y/N)" GET mnow PICTURE "Y";
     VALID mnow $ "YN" ERROR "Enter Y or N only"
```

The .OR. logical operator is actually an *inclusive or.* In other words, *X .OR. Y* means *X or Y, or both X and Y.* dBASE IV doesn't provide an *exclusive or,* but you can easily construct one using logical operators:

```
(X .OR. Y) .AND. .NOT. (X .AND. Y)
```

The following routine uses the logical operator .NOT., plus the FOUND() and EOF() functions to process the possible outcomes of a search with SET NEAR ON. If *mkey* (the vendor code) is found, FOUND() is set to .T. and the first CASE prints the vendor data. If *mkey* is not found, the record pointer stops at the next record in alphabetical key order. Thus, in the second CASE, EOF() is .NOT. true and the appropriate message prints. OTHERWISE, a message indicates that the record pointer is at the end of the file.

```
USE Vendors ORDER Vend_id
SET NEAR ON
mkey = Vendor_id
SEEK mkey
DO CASE
   CASE FOUND()
      ? Vendor_id, Vendor, Phone, City
   CASE .NOT. EOF()
      ? "Vendor not found"
      ? "Next vendor (alphabetically) will display"
      ? Vendor_id, Vendor, Phone, City
   OTHERWISE          && EOF() = .T.
      ? "Vendor not found"
      ? "Record pointer at bottom of file"
ENDCASE
```

Converting Data Types

Sometimes you need to convert data from one type to another. For example, if you want to @...SAY text that includes both character strings and a date or number, you convert the non-character data to character type. Similarly, calculations with mixed types require that all data be converted to one type.

When you convert the data types of fields or memory variables, you are not changing field types in the database file, nor are you modifying the structure of the database file. Table 3-5 summarizes how to convert data types.

Table 3-5 Converting data types

Conversion	Example
Character to numeric	mpage_no = VAL(mpage_no)
Numeric to character	? "Amount is $" + STR(mamount,5,2) or ? "Page no.: " + STR(_pageno,3)
Numeric to integer	? INT(mamount)
Floating to BCD	? FIXED (<floating point number>)
BCD to floating	? FLOAT (<BCD number>)

(continued))

Table 3-5 Converting data types *(continued)*

Conversion	Example
Character to date	Order_date = CTOD("09/12/88")
Literal to date	{}
Date to character	? "The date is " + DTOC(DATE())
Date to string (for indexing)	INDEX ON DTOS(Date_trans) + Cust_id; TAG Orders
Logical to character	mfinished = IIF(mfinished,"Yes","No")
Memo to character	REPLACE mstring WITH mmemo
Character to memo	REPLACE mmemo WITH mstring

The following code fragments represent some typical data type conversions.

To index by date and amount ordered, convert both the date and numeric fields to character strings:

```
INDEX ON DTOS(Date_order) + STR(Amount,8,2) TO Date_amt
```

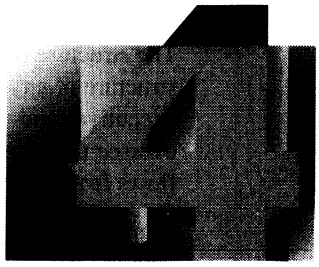
The following code extracts the two characters for month and year from the date (assume the MM/DD/YY format). Then it concatenates them with the customer ID number to generate a unique invoice number:

```
this_year  = RIGHT(DTOC(DATE()),2)
this_month = LEFT(DTOC(DATE()),2)
invoice_no = cust_id + this_year + this_month
```

This next example increments *m_id,* an employee identification code such as B1234, by one. To do this, it converts the four digits to numeric data using the VAL() function, and increments the resulting number by one. Then it uses STR() to convert the number back to a character string, which it concatenates with the single letter on the LEFT() end of the identification code.

```
m_id = LEFT(m_id,1) + STR(VAL(RIGHT(m_id,4)) + 1,4,0)
```

Program Architecture



The techniques of structured programming help you write efficient programs that are easy to understand and maintain. dBASE IV supports procedures and procedure files as well as several programming constructs for this purpose.

What This Chapter Covers

This chapter covers the following topics:

- Structured programming
  - Top-down design and modular programming
  - Procedures, user-defined functions, and procedure files
  - Parameter passing
- Programming constructs

Structured Programming

Structured programming is an organized approach to programming that produces code you can understand and maintain. If you use the techniques of structured programming, your dBASE programs will be easier to debug, maintain, and expand. In the long run, this will increase your productivity and make your applications more marketable.

The section Top-Down Design and Modular Programming is a general discussion of structured programming. The Procedures and Procedure Files section explains how to write structured programs in dBASE IV.

Top-Down Design and Modular Programming

Structured programming entails top-down design and modular programming. *Top-down design* breaks an application into many specific tasks. The breakdown is hierarchical, with higher level functions activating lower level tasks. Top-down design naturally leads to *modular programming,* in which you code each task as a separate program module.

Top-Down Design

The structure of an application designed from the top down resembles the hierarchical structure of a typical company. In a hierarchically organized company, top-level executives and managers control the system and delegate work. Mid-level supervisors oversee the work being done. Low-level workers perform the actual tasks. *Control* flows from top to bottom; *work* flows from bottom to top.

Similarly, in an application designed from the top down, the top-level main menu controls the entire system and delegates work to the rest of the system. Mid-level menus or modules oversee the work performed solely by low-level program modules. These modules contain the program's data entry, validation, processing, and output capability.

Top-down design has distinct advantages. Smaller tasks are easier to understand and code than large ones, so organizing a large project into manageable units saves you many wrong turns as you write the code. Moreover, this approach allows you to delegate modules to other programmers if you are working in a team.

Figure 4-1 diagrams the Business application provided with your sample files, which was designed from the top down. It consists of the main Business program, seven subapplications, and a Library file containing procedures used throughout the application.

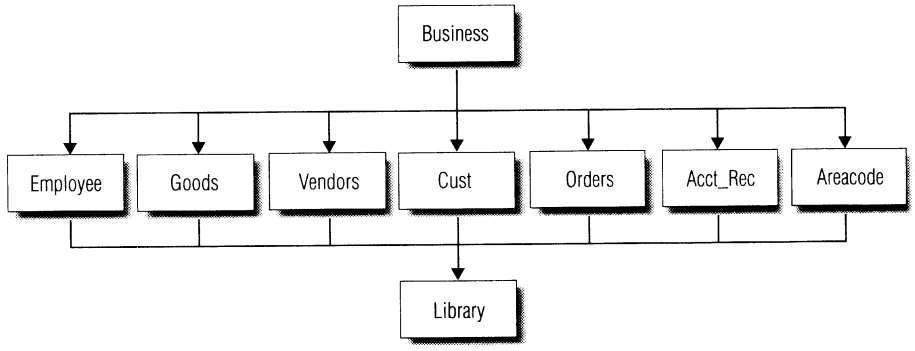


Figure 4-1 Architectural diagram of the Business application

Modular Programming

When designing your application from the top down, you identify a set of very specific tasks you want your application to perform. These tasks form the basis for distinct program modules.

A modular program has the following attributes:

- Each module performs only one basic task
- Each module is only called by modules above it and only calls modules below it
- Each module has only one entry point
- Upon completion of a module, program control usually returns to the calling program
- Interaction between modules is kept to a minimum
- Where possible, the program reuses modules

In contrast to top-down *design,* you should *code* your program from the bottom up. Outline and write each module separately, starting with the most specific modules and working up to the higher level ones.

Coding from the bottom up lets you prototype your application one module (task) at a time. You can test each module after you write it and, where appropriate, demonstrate the module to potential users in order to obtain preliminary feedback on your application. Debug each module as much as possible before combining it with others. (Chapter 15 discusses the dBASE IV debugger.)

When you've completed a number of modules, activate them from a main menu so users can try out the application. Use program *stubs* in place of modules that you haven't yet developed. (A program stub is a message telling the user that the selected task is not yet operational.)

Coding from the bottom up helps you identify modules used repeatedly throughout an application. Put these modules in a library file. For example, Library.prg in the Business application contains modules called by the sub-applications to add, edit, and delete records. Using library files results in more efficient code, smaller file sizes, and shorter development schedules, because you only have to document, debug, maintain, and revise each module once. You can use the modules in library files repeatedly throughout an application, and in other applications that you write later.

Procedures and Procedure Files

The previous section discussed structured programming on a conceptual level. This section shows you how dBASE IV supports structured programming with procedures, user-defined functions, and procedure files.

A *procedure* is a dBASE IV program module that is executed from a higher level or *calling* procedure. Each procedure begins with the command PROCEDURE <procedure name> and ends with a RETURN statement. (A RETURN statement is not required, because it is always implied.) A program (.prg) file can contain up to 963 procedures (limited in practice by available memory). Each procedure can have up to 65,520 bytes of generated code.

To execute a procedure, simply DO <procedure name> at the appropriate point in your program. Here is a typical procedure, taken from the Business application:

```
PROCEDURE List_rec
   record_num = RECNO()
   ACTIVATE WINDOW lister
   answer = " "
   CLEAR
   @ 0,0 SAY "------------LIST RECORDS------------";
         COLOR &c_red.
   SCAN WHILE .NOT. answer $ "rR"
      LIST OFF NEXT 10 &list_flds.
      WAIT "Press spacebar to continue or R to return to OPTION MENU";
           TO answer
      CLEAR
   ENDSCAN
   DEACTIVATE WINDOW lister
   GO record_num
RETURN
```

A program can also call procedures from an external file known as a *procedure file.* Generally, the procedures in a procedure file are basic routines called by more than one program in one or more applications. Although you can only open one procedure file at a time, you can access an unlimited number of procedures by opening, successively, any number of procedure files.

The following code executes the Set_env procedure, one of many procedures in Library.prg. The first command closes any previously opened procedure file and opens Library.prg. The second command runs the procedure.

```
SET PROCEDURE TO Library
DO Set_env
```

Using separate procedure files simplifies the task of repairing or revising a program, because you can just substitute a replacement procedure file for the faulty one. However, putting procedures in the main program makes the application run more quickly.

To conserve memory, don't open a procedure file until your program needs it. Then keep it open as long as necessary, rather than closing and reopening it. This minimizes disk access time.

In your ending program code, use CLOSE PROCEDURE or SET PROCEDURE TO without a filename to close the last procedure file.

See the Procedure Libraries section in Chapter 15 for added library support in dBASE IV.

User-Defined Functions

Like the internal dBASE functions, user-defined functions (UDFs) consist syntactically of the function name followed by parentheses, which sometimes contain a list of parameters. Unlike the dBASE functions, you code UDFs yourself.

UDF procedures always begin with the command *FUNCTION <function name>* and end with *RETURN <expression>*. The FUNCTION command distinguishes a UDF from an ordinary procedure, which begins with the PROCEDURE command. Note that you *must* specify a RETURN expression in a UDF, whereas you *cannot* do so in an ordinary procedure.

Prof_mgn() is a simple user-defined function from Orders.prg that calculates profit margins:

```
FUNCTION  Prof_mgn
    PARAMETERS cost, price
    margin = ROUND((price - cost)/price*100,1)
RETURN  margin
```

You also cannot use certain commands, such as BROWSE and EDIT, recursively in the same work area (see the User-Defined Functions section in Chapter 1, and the FUNCTION command in Chapter 2 of *Language Reference* for further information). Don't give a UDF the same name as a dBASE function, since dBASE IV will execute its own function rather than the UDF.

Compiling Program Code

The program code in a command (.prg) file is known as *source code*. Before it can execute a program, dBASE IV compiles it into *tokenized code*, assigning the file a .dbo extension. The dBASE IV pseudo-compiler strips out comments and unnecessary spaces, and checks the code for syntax errors such as unclosed programming constructs (see the Programming Constructs section later in this chapter).

Use the DO or COMPILE command to activate the dBASE IV pseudo-compiler. DO compiles the source code (if it can't find compiled tokenized code) and then executes the program. COMPILE generates a tokenized file without executing it.

The dBASE IV MODIFY COMMAND editor automatically deletes the previous .dbo file when you modify a program. dBASE IV then generates a new .dbo file the next time you DO the program. If you're using an external editor, SET DEVELOPMENT ON (the default) instructs dBASE IV to recompile the .prg file if the .dbo file was created earlier. See Chapter 15 for more information on the dBASE IV pseudo-compiler.

Parameter Passing

Procedures enable you to use identical modules over and over. Sometimes, however, you want to reuse a module but include a few changes. To keep down the size of your program, write a single procedure. Then DO the procedure WITH different parameters in different parts of your program. This technique is known as *parameter passing*.

The following Plus_tax procedure calculates the total bill by computing the tax on the amount of purchase and adding it to the subtotal. The PARAMETERS command identifies the parameters being passed to the procedure. The number of declared parameters does not have to match the number of passed parameters.

```
mtotal = 0
DO Plus_tax WITH subtotal, tax

PROCEDURE Plus_tax
    PARAMETERS mvalue, mtax
    mtotal = ROUND(mvalue * mtax,2) + mvalue
RETURN
```

Programming Constructs

In its simplest form, a dBASE program or *command file* is just a sequence of dBASE commands stored in a file. Each command executes only once. However, the real power of the dBASE language lies in its *programming constructs.* They control the logic or *flow* of a program — the way program control moves from one part of the program to another as the program runs. These programming structures establish the order in which commands execute, the conditions under which they execute, and how often they execute.

A typical program uses four types of program logic: sequential processing, choice, repetition, and program interrupts. dBASE IV provides programming constructs for each of these:

- Sequential processing — DO, RUN, and CALL
- Choice constructs — IF...ENDIF and DO CASE...ENDCASE
- Repetition constructs — DO WHILE...ENDDO and SCAN...ENDSCAN
- Program interrupts — ON ESCAPE, ON KEY, ON ERROR, ON READERROR, and ON PAGE

Sequential Processing

The simplest form of sequential processing involves executing the commands in a program in sequence. More complex sequential processing can involve a call to a procedure within the program or even to an external program. dBASE IV provides three sequential processing commands:

- DO executes a procedure within the main program or in a separate program such as a procedure file.
- RUN executes an external program at the operating system level.
- CALL executes a machine language object program that you have previously LOADed.

Figure 4-2 illustrates sequential processing.

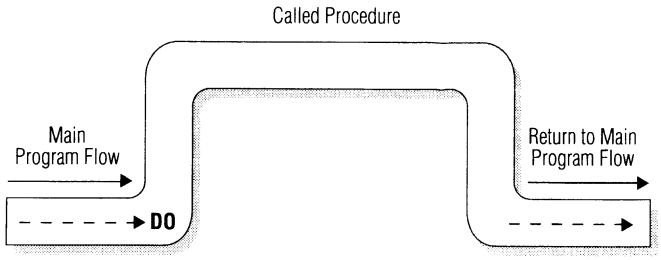


Figure 4-2 Calling a program or procedure (DO, RUN, CALL)

DOing a Procedure or Program

The command DO <filename> executes the named procedure or program, which may be in the current program file or in a separate program or procedure file (see the Procedures and Procedure Files section above). When dBASE IV encounters the DO command, it *calls* the procedure or program, performs the instructions in that routine, and then returns control to the appropriate level.

The final RETURN command in a procedure passes control back to a higher level of the program. A RETURN command with no argument returns control to the calling program or procedure. RETURN TO MASTER returns program control to the highest level calling procedure. This saves stepping back through several procedures or programs to get to the main program.

The first DO command in the following example calls the procedure Filter, which occurs with slight variations within each of the Business sub-applications. That procedure executes Filt_ans from the Library.prg file. The series of DOs is known as a *call chain.*

```
SET PROCEDURE TO Library
DO Filter
DO Filt_ans
```

RUNning an External Program

The RUN command (abbreviated as !) runs an external command or program (.exe) at the operating system level. The RUN() function can rollout dBASE IV from RAM, allowing you to execute a different external program. See *Language Reference* for more information on the RUN() function.

CALLing an Object File

You can execute a binary (.bin) file from within dBASE programs using the CALL command. dBASE IV treats binary files as modules rather than external programs. These modules must be externally compiled, then converted to binary format before you can CALL them.

The LOAD command places a routine in memory, and the CALL command executes it. The memory limit for the files LOADED into memory is 64K. See Chapter 2 of *Language Reference.*

You can also use the CALL() function to execute modules written in other languages. CALL() has up to eight arguments. The first names the entry point you are CALLing, and the remaining arguments pass parameters to it. See Chapter 4 of *Language Reference* for more information.

Choice Constructs (IF...ENDIF, DO CASE...ENDCASE)

A dBASE program can execute different tasks in different circumstances or conditions. dBASE IV offers two conditional programming constructs: IF...ENDIF and DO CASE...ENDCASE.

Single Conditions (IF...ENDIF)

When dBASE IV encounters an IF...ENDIF construct, it executes the commands in the construct when the IF condition holds, but ignores them when the IF condition is not true (see Figure 4-3).

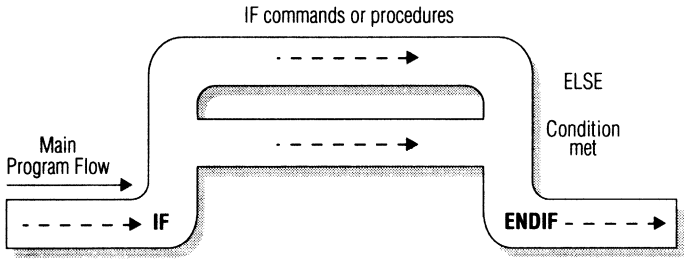


Figure 4-3 Single condition (IF...ENDIF)

The routine shown here prints an order form when the quantity on hand of an item is at or below the order point. The ELSE clause causes a report of the stock on hand to print if that condition is not met.

```
order_pt = 5
IF Qty_onhand <= order_pt
    REPORT FORM Ord_form NEXT 1 TO PRINTER
ELSE
    DO Stok_now
ENDIF
```

Variation — In this modification of the previous example, the order form only prints if the quantity on hand is at or below order point *and* the variable *do_form* is set to true (.T.):

```
order_pt = 5
IF Qty_onhand <= order_pt
   IF do_form
      REPORT FORM Ord_form NEXT 1 TO PRINTER
   ENDIF
ELSE
   DO Stok_now
ENDIF
```

Multiple Conditions (DO CASE...ENDCASE)

When dBASE IV encounters a DO CASE...ENDCASE construct, it determines which of several conditions is true and executes the associated commands (see Figure 4-4). DO CASE...ENDCASE handles several alternatives where IF...ENDIF handles at most two.

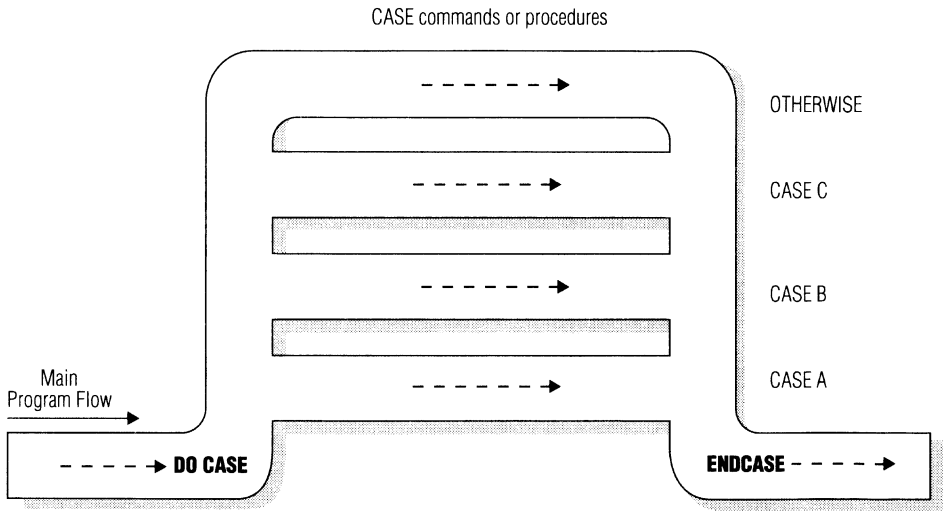


Figure 4-4 Multiple conditions (DO CASE...ENDCASE)

CASE statements are typically used to execute a different command or procedure for each item in a menu. The following procedure, Main, defines the main menu of the Business application. Chapter 7 explains the dBASE IV menu-building commands and functions used here.

```
PROCEDURE Main
   DO CASE
      CASE BAR() = 3
         DO Employee
      CASE BAR() = 4
         DO Cust
      CASE BAR() = 5
         DO Vendors
      * <more CASEs>
      CASE BAR() = 13 .OR. BAR() = 14
         RELEASE ALL
         SET CLOCK ON
         DO Colo_RESET
         IF BAR() = 13
            RETURN TO MASTER
         ELSE
            QUIT
         ENDIF
   ENDCASE
RETURN
```

You can include an OTHERWISE clause in a DO CASE...ENDCASE to handle all possibilities not covered by the CASEs.

Repetition Constructs (DO WHILE...ENDDO, SCAN...ENDSCAN)

The process known as *looping* means your program is performing the same steps repeatedly while a given condition is true. The program steps through the commands, loops back, steps through them again, and so on, until the condition no longer holds true. Then it performs the command immediately following the loop.

dBASE IV handles loops with the DO WHILE...ENDDO and SCAN...ENDSCAN constructs (see Figure 4-5).

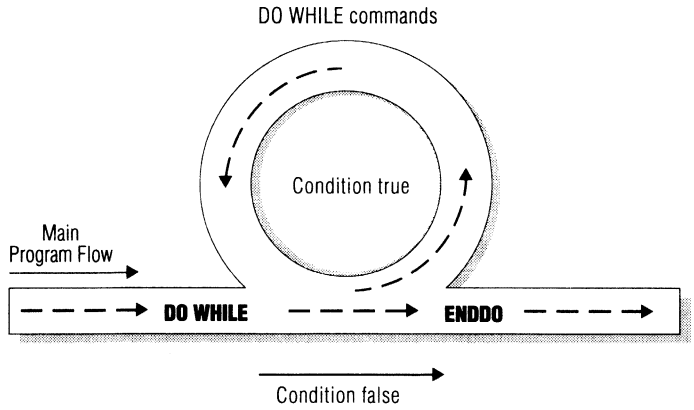


Figure 4-5 Looping (DO WHILE...ENDDO, SCAN...ENDSCAN)

Perhaps the most common use of a loop is to print or process all the records in a database file. To do this, tell dBASE IV to keep running the commands in the loop as long as it *doesn't* reach the end of the file (DO WHILE .NOT. EOF()). The following routine prints the specified fields for all records in Goods:

```
USE Goods ORDER Part_id
DO WHILE .NOT. EOF()
    ? Part_id, Part_name AT 15, Descript
   SKIP
ENDDO
```

You can accomplish the same thing in fewer lines of code with LIST, as shown below.

```
USE Goods ORDER Part_id
LIST Part_id, Part_name AT 15, Descript
```

Another popular loop uses the condition DO WHILE .T. This condition causes an infinite loop, because true is always true. Use it to display a menu until the user chooses an item.

Finally, many loop conditions pertain to characteristics of the records themselves. For example, you might want to list all California records to the screen (assume an index tag on State):

```
USE EMPLOYEE
INDEX ON State TAG State
USE Employee ORDER State
SEEK "CA"
SCAN WHILE State = "CA"
    ? Lastname, Firstname, Address1, City
ENDSCAN
```

You can exit a DO WHILE loop before dBASE IV has processed all the commands in the loop. The EXIT option transfers control to the command immediately following the next ENDDO command. Use it when you want to give the user a chance to exit a DO WHILE loop prematurely. The LOOP option starts the loop over. See Chapter 2 of *Language Reference* for more information.

Program Interrupts

dBASE IV provides a useful set of *program interrupts* to interrupt program execution when certain events occur (see Figure 4-6). These program interrupts are all variants of the ON command: ON ESCAPE, ON KEY, ON ERROR, ON READERROR, and ON PAGE. Put ON commands in your program setup code if you want them to be in effect during the entire program.

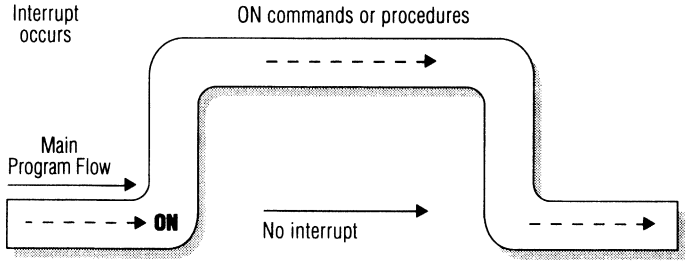


Figure 4-6 Program interrupts (ON commands)

Acting on a Keypress (ON ESCAPE, ON KEY)

Use ON ESCAPE and ON KEY to call a procedure when the user presses a key. ON ESCAPE tests for the **Esc** key, ON KEY for any other key you specify.

ON KEY LABEL <key label> executes a command or routine when the user presses the designated key. ON KEY with no LABEL executes the associated command or routine when the user presses any key. In that case, ON ESCAPE takes precedence over ON KEY. In other words, if a user presses **Esc**, the ON ESCAPE routine executes, not the ON KEY routine. ON KEY LABEL will trap any key the user presses.

The following command stops a report when the user presses **Esc.** The Stop_rpt procedure is in Invoices.prg (see the sample code).

```
ON ESCAPE DO Stop_rpt
```

In the following routine, the user can press any key to stop printing. The INKEY() function clears from the typeahead buffer the key that activated the ON KEY procedure.

```
ON KEY DO Stop_prt
REPORT FORM Sample TO PRINTER

PROCEDURE Stop_prt
   i = INKEY()
   choice = " "
   DO WHILE .NOT. UPPER(choice) $ "PS"
      WAIT "Press S to stop printing, P to proceed";
           TO choice
   ENDDO
   IF UPPER(choice) = "S"
      RETURN TO MASTER
   ENDIF
RETURN
```

Acting on an Error (ON ERROR, ON READERROR)

Use ON ERROR and ON READERROR to call an error routine when a dBASE or data entry error occurs.

ON ERROR only traps dBASE system errors, such as a syntax error, not operating system errors. The following code activates an error help routine when a dBASE error occurs:

```
ON ERROR DO Err_help

PROCEDURE Err_help
   ACTIVATE WINDOW alert
   DO Warnbell
   ? ERROR()
   ? MESSAGE()
   WAIT "A system error has occurred. Press any key to continue."
   DEACTIVATE WINDOW alert
RETURN
```

ON READERROR executes a command or procedure if the user enters invalid data. It is only used during full-screen operations like APPEND, EDIT, and READ. See Chapter 2 of *Language Reference* for more information on this command. The Checking Data for Errors section in Chapter 8 of this manual discusses data validation.

Acting on a Page Break (ON PAGE)

ON PAGE activates a page break routine when the specified line number is reached during report output. The following page break routine prints a centered page number at the bottom of the page, ejects the paper, and prints the date at the top of the next page.

```
ON PAGE AT LINE 55 DO Page_brk

PROCEDURE Page_brk
   _wrap = .T.
   _alignment = "CENTER"
   ? "-" + LTRIM(STR(_pageno,3,0)) + "-"
   ?
   EJECT PAGE
   _alignment = "LEFT"
   ?? MDY(DATE()) AT 0
   ?
RETURN
```

Chapter 13 explains the ON PAGE command in more detail and shows other header and footer procedures.

Nesting Control Structures

Each of the basic control structures can operate within others. This is known as *nesting.*

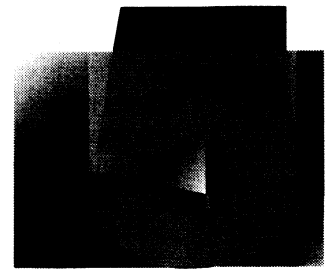
The following excerpt from the Barpop procedure nests an IF...ENDIF within an IF...ENDIF within a DO CASE...ENDCASE. The CASE statement controls what happens when the user picks bar 17 (the **Index database** option) from the **Option** menu. The outer IF attempts to put the database file in exclusive use if the program is running on a network. If the outer IF is successful, the inner IF calls the Indexer procedure to create the index and then reopens the file for shared use. The ELSE portion of the outer IF executes when the program is not running on a network. It simply calls Indexer, since exclusive use isn't necessary to create indexes in single-user dBASE IV.

```
DO CASE
   CASE BAR() = 17
      IF NETWORK()
         old_tag = ORDER()
         USE (dbf) EXCLUSIVE
         IF net_choice <> 27
            DO Indexer
            SET EXCLUSIVE off
            Use (dbf) ORDER (old_tag)
         ENDIF
      ELSE
         DO Indexer
      ENDIF
   * <more CASE statements>
ENDCASE
```

TIP

Be sure to provide a closing command for each construct. To make your code easier to read, indent each level of nesting by three spaces. Indenting also helps you determine at a glance whether you've ended each control structure properly.

Setting Up the Environment



The program setup code establishes the working environment and initializes global memory variables used throughout the program. It also displays the main menu and calls the subprograms activated by it.

What This Chapter Covers

This chapter discusses the main setup code of the Business sample application. It covers the following topics:

- Program headers
- Setting up the environment
- Initializing memory variables
- Displaying the main menu
- Opening and relating files
- Cleaning up the environment

Program Headers

Every program should start with a *program header* that includes the program name, what the program does, the programmer's name, and a revision history. There is no mandatory format for the program header. Here is a possible format:

```
*PROGRAM NAME:   <name of program>
*PURPOSE:         <purpose of program>
*WRITTEN BY:      <programmer`s name>
*LAST CHANGED:   <date and time last revised>
```

Establishing the Working Environment

The working environment of a program includes the default configuration of the screen, printer, and keyboard, and other aspects of the environment in which your program runs. Using SET commands and system memory variables, you can establish a default working environment for your program. In the program setup area, define only environmental characteristics that will remain constant throughout your application.

Table 5-1 lists SET commands commonly used in program setup code. For ON/OFF SET commands, the table shows the setting recommended for use in programs. (See Chapter 3 of *Language Reference* for more information on the SET commands.)

Table 5-1 SET commands commonly used in program setup

Operation	SET Command
Configure the screen display	
Specify a type of monitor	SET DISPLAY TO
Set a default color	SET COLOR TO SET COLOR OF...TO
Define a default border	SET BORDER TO
Configure the clock	SET CLOCK ON SET CLOCK TO
Prevent access to design mode	SET DESIGN OFF
Configure screen messages	
Suppress the status bar	SET STATUS OFF
Suppress system messages that show on row 0 when STATUS is OFF	SET SCOREBOARD OFF
Suppress system help messages	SET HELP OFF
Suppress warning messages before overwriting files	SET SAFETY OFF
Suppress system responses to certain commands	SET TALK OFF
Suppress redisplay of command lines	SET ECHO OFF
Specify line number for messages	SET MESSAGE TO

(continued)

Table 5-1 SET commands commonly used in program setup *(continued)*

Operation	SET Command
Configure data handling and display	
Specify the number of decimals	SET DECIMALS TO
Allow inexact comparisons	SET EXACT OFF
Ignore DELETEd records	SET DELETED ON
Show column headings	SET HEADING ON
Insert space between expressions output with ?/??	SET SPACE ON
Move pointer to end of file if search fails	SET NEAR OFF
Select destination for printer output	SET PRINTER TO
Configure the keyboard	
Configure function keys	SET FUNCTION...TO
Keep **Esc** from interrupting the program	SET ESCAPE OFF
Control the keyboard buffer	SET TYPEAHEAD TO
Configure drives and directories	
Set the default drive	SET DEFAULT TO
Set the default drive and directory	SET DIRECTORY TO
Set the file search path	SET PATH TO
Make DBF( ), MDX( ), and NDX( ) return full path with filename	SET FULLPATH ON
Configure the bell	
Define pitch and duration	SET BELL TO
Suppress the system bell	SET BELL OFF

Table 5-2 lists some system memory variables for the program setup area, showing suggested settings where practical. The system variables shown here control print settings that are likely to remain constant throughout a program. See Chapters 12 and 13 of this book and Chapter 5 of *Language Reference* for more information on the dBASE IV system memory variables.

Table 5-2 System memory variables used in program setup

Operation	System memory variable
Specify a printer driver file	_pdriver = "<filename>"
Specify a print form file	_pform = "<filename>"
Set printer pitch	_ppitch = "<pitch>"
Set printer to letter quality	_pquality = .T.
Define starting print code	_pscode = "<start code>"
Define ending print code	_pecode = "<end code>"
Do a form feed before PRINTJOB...ENDPRINTJOB	_peject = "BEFORE"
Print defined boxes	_box = .T.

The program setup code below is an excerpt from Set_env, the setup procedure in Library.prg.

First the routine initializes several memory variables to hold the current environmental settings. Then it uses the SET( ) function to capture the current settings and store them to the variables. Finally, a series of SET commands defines the settings required by the program.

```
PROCEDURE  Set_env
    PUBLIC clock, esca, stat, talk       && <other variable definitions>
    clock  =  SET("CLOCK")
    esca   =  SET("ESCAPE")
    colo   =  SET("COLOR")
    stat   =  SET("STATUS")
    talk   =  SET("TALK")
    * <Other SET() functions>

    SET CLOCK OFF
    SET ESCAPE OFF
    DO  Colo_RESET
    SET STATUS OFF
    SET TALK OFF
    * <Other SET commands>
RETURN
```

Initializing Global Memory Variables

After setting up the working environment, Set_env initializes memory variables for the program. Here is the code that initializes the global memory variables used throughout the Business application.

```
PUBLIC  erased,  not_valid,  rec_is_dup,  filters_on,  lookup_ok,  choice
PUBLIC  record_num,  net_choice
PUBLIC  target,  look_dbf,  matchchar,  scanfield
STORE  .F.  TO  erased,  not_valid,  rec_is_dup,  filters_on
lookup_ok  =  .T.
STORE  ""  TO  choice,  subset
STORE  0  TO  record_num,  net_choice
```

Displaying the Main Menu

Most dBASE programs have one main menu from which all submenus branch. This allows just one entry and one exit from the program. The main menu appears whenever control returns to the main program.

Here is the code that defines the main menu of the Business application. See Chapter 7 for more information on the dBASE IV menu-building capabilities.

```
PROCEDURE Main_def
   DEFINE POPUP mainmenu FROM 7,27 TO 22,50;
    MESSAGE "Press first letter of menu choice or highlight and press <Enter>"
   DEFINE BAR  1 OF mainmenu PROMPT "====== MAIN  MENU ======" SKIP
    DEFINE BAR  2 OF mainmenu PROMPT "      Databases:" SKIP
    DEFINE BAR  3 OF mainmenu PROMPT " EMPLOYEES"
    DEFINE BAR  4 OF mainmenu PROMPT " CUSTOMERS"
    DEFINE BAR  5 OF mainmenu PROMPT " VENDORS"
    DEFINE BAR  6 OF mainmenu PROMPT " INVENTORY"
    DEFINE BAR  7 OF mainmenu PROMPT " ORDERS"
    DEFINE BAR  8 OF mainmenu PROMPT " ACCOUNTS RECEIVABLE"
    DEFINE BAR  9 OF mainmenu PROMPT " AREA CODES"
    DEFINE BAR 10 OF mainmenu PROMPT "     Utilities:" SKIP
    DEFINE BAR 11 OF mainmenu PROMPT " PRINT INVOICES"
    DEFINE BAR 12 OF mainmenu PROMPT " BACK UP/RESTORE DATA"
    DEFINE BAR 13 OF mainmenu PROMPT " RETURN TO dBASE"
    DEFINE BAR 14 OF mainmenu PROMPT " QUIT dBASE"
    ON SELECTION POPUP mainmenu DO Main
RETURN
```

Opening and Relating Files

The Business application program encompasses seven separate application programs. Each of them uses its own database files, indexes, and screen procedures.

The following code from Orders.prg opens and relates the Orders, Goods, Cust, and Employee database files with their respective index files. Comparable routines can be found in Employee, Cust, Vendors, Goods, and Acct_rec.

The USE commands open each database file ORDERed by its own index tag in its respective work area. The SET RELATION command relates Orders to each of the other three database files. GO TOP synchronizes the related files. (Chapters 9 and 11 discuss indexing and relating database files, respectively.)

```
SELECT 1
USE Orders ORDER Order
USE Goods ORDER Part_id IN 2
USE Cust ORDER Cust_id IN 3
USE Employee ORDER Emp_id IN 4
SET RELATION TO Part_id INTO Goods, Cust_id INTO Cust,;
                Emp_id INTO Employee
GO TOP
```

TIP

In a high-level program, open only files that you intend to use throughout the application. Keep as few files as possible open at once to minimize the risk of catastrophic data loss. Try to open files only when you need them, and close them as soon as you are finished with them. If you must leave database files open in your application, SET AUTOSAVE ON.

Cleaning Up the Environment

If you are writing a turnkey application that will run only from the operating system, you don't need to restore the environment because your application will exit dBASE IV when it terminates. (You'll still want to restore settings modified within procedures.)

If your application runs from within dBASE IV, however, it must restore the environment before terminating. (Programmers often call this *housekeeping.)*

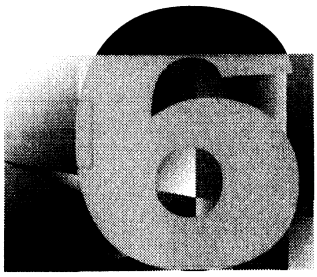
To properly clean up the environment that you established earlier, close all open database files, release memory variables, restore the working environment to its original settings, and return users to the point from which they entered the program.

The code shown below, from the Barpop procedure in Library.prg, executes when the user selects **Quit to Main Menu** in Vendors, Orders, and the other sub-applications within Business. The first three commands close all files and release windows and memory variables. The RELEASE ALL command does not release popups, so the main Business popup will still display.

The Rest_env procedure (also in Library.prg) restores the environment to the user's original settings, which were captured in Set_env.

```
CLOSE  DATABASES
CLEAR  WINDOWS
RELEASE  ALL
ON  ERROR
DO  Rest_env
CLEAR
RETURN  TO  MASTER

PROCEDURE  Rest_env
    SET CLOCK &clock.
    SET ESCAPE &esca.
    SET STATUS &stat.
    SET TALK &talk.
    * <other SET commands>
RETURN
```

The Look of Your Application



You can improve the look of your application using the techniques discussed in this chapter. Appropriately placed lines, boxes, windows, and color make data entry and menu screens easier to read, and give the entire application a more polished look.

What This Chapter Covers

The topics covered in this chapter are:

- Displaying lines and boxes
- Working with windows
- Working with borders
- Working with colors

This chapter focuses on the screen environment, and therefore does not discuss how to *print* lines and boxes. For more information on printing lines and boxes, see Chapter 12 of this book. Chapter 17 of *Using dBASE IV* discusses interface design principles.

Some Terminology

Before reading this chapter, you should be clear about the difference between boxes and windows. *Boxes* are rectangles consisting of characters drawn on the screen, usually surrounding related items in a screen form. Text that exceeds the perimeter of the box overwrites the border (see Figure 6-1).

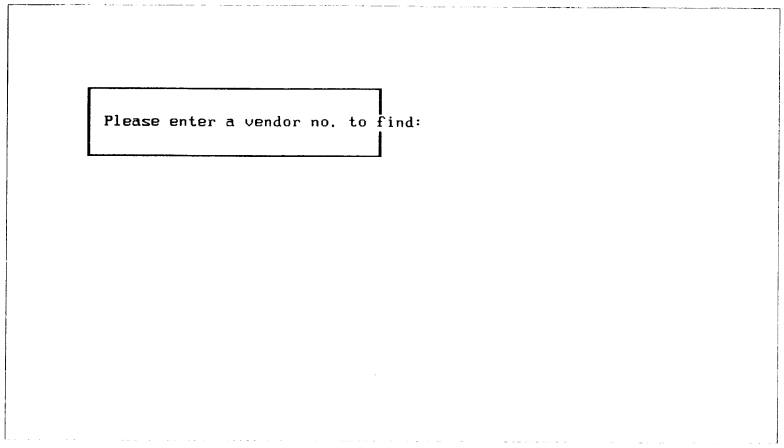


Figure 6-1 String exceeding box dimensions

Windows are rectangular areas of the screen in which program operations occur. You use windows to LIST output or display memo fields, prompts, and messages. Text that exceeds the window dimensions wraps or scrolls within the window (see Figure 6-2).

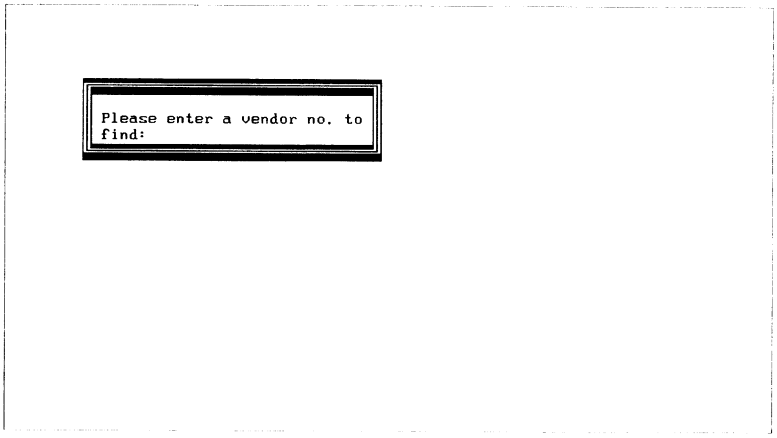


Figure 6-2 String exceeding window dimensions

Drawing and Clearing Lines and Boxes

You can code lines or boxes like those shown in Figure 6-3 with the dBASE IV @...TO command. This command lets you position the line or box on the screen and define a variety of possible borders. Many of the examples in this section are based on the screen form design shown in the figure. To see the figure in color, select **VENDORS** from the Business application main menu.

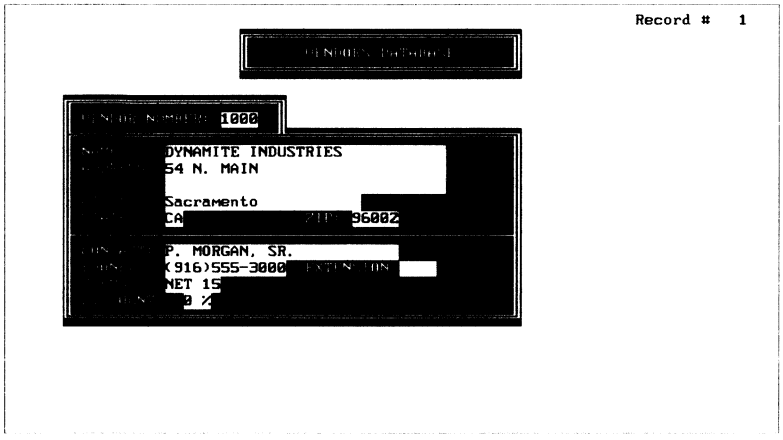


Figure 6-3 Vendor screen form

NOTE
The @...TO command draws lines and boxes on the screen only. Use DEFINE BOX to print lines and boxes (see Chapter 12).

Drawing Lines on the Screen

You can draw lines in screen forms to visually separate entry fields. The command that draws the horizontal line above the bottom half of the form in Figure 6-3 is:

```
@ 14,5 TO 14,52 COLOR R/W
```

Note that horizontal line definitions have the same beginning and ending *row* coordinates. To draw a vertical line on the screen, keep the *column* coordinates the same.

To draw lines with a character of your choice (including graphic characters), include the character with the @...TO command. Here's how to draw a 20-character line on the screen with the character *.

```
@ 14,24 TO 14,43 "*"
```

Drawing Boxes on the Screen

Use the @...TO command to position boxes on the screen, specify their dimensions, and define their border characters. The following code draws the boxes around the form title and the vendor number prompt in Figure 6-3 above:

```
@ 1,22 TO 3,53 COLOR B/W
@ 5,4 TO 7,27 COLOR R/W
```

The SET BORDER TO command also affects the border string of both lines and boxes.

Clearing Lines and Boxes

The @...CLEAR TO command clears lines and boxes from the screen. The following example displays a prompt in a box. Then it CLEARs the area within the box and displays a second prompt:

```
@ 10,10 TO 14,60
@ 12,11 SAY "Please enter a vendor no. to find: " GET m->vendor_id
READ

@ 11,11 CLEAR TO 13,59
@ 12,11 SAY "Please enter a part no. to find: " GET m->part_id
READ
```

Working with Windows

Windows define the area of the screen where operations take place. While a window is active, all screen input and output occurs within its borders. The system treats the top left position in the window as coordinate 0,0 and adjusts any screen coordinates you specify accordingly.

Figure 6-4 shows a data LISTing in a window.

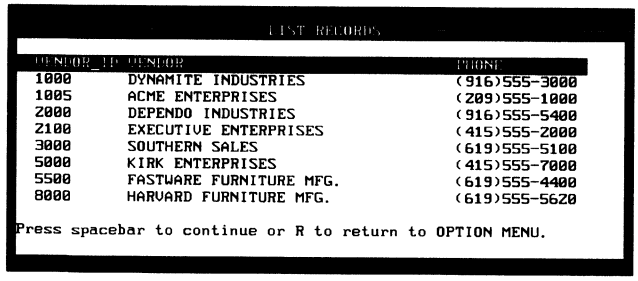


Figure 6-4 LISTing to a window

The next two code examples refer to this window. See the More Uses of Windows section for other examples of how to use windows in your applications.

Defining a Window and Its Borders

To DEFINE a WINDOW, give it a name (limited to 10 characters), provide upper left and lower right screen coordinates (limited to screen dimensions), and, optionally, define its border and colors. You can define up to 20 windows in memory at one time.

The Bar_def procedure in Library.prg defines the window shown in Figure 6-4 as follows:

```
DEFINE WINDOW lister FROM 5,5 TO 22,70 PANEL COLOR &c_list.
```

(The Working with Colors section later in this chapter shows the contents of *c_list.)*

You can make the border a single line, a double line, an inverse video panel, or a character string of your choice. See DEFINE WINDOW and SET BORDER in *Language Reference* for more details.

Activating and Deactivating Windows

You ACTIVATE and DEACTIVATE defined windows to control when they appear on the screen. When you ACTIVATE a WINDOW, it covers any text in the same area of the screen. When you DEACTIVATE the WINDOW, the text reappears.

Here's the complete code from Library.prg for the LISTing shown in Figure 6-4.

```
PROCEDURE List_rec
   record_num = RECNO()
   ACTIVATE WINDOW lister
   answer = " "
   CLEAR
   @ 0,0 SAY "----------------- LIST RECORDS ----------------" ;
         COLOR &c_red.
   SCAN WHILE .NOT. answer $ "rR"
      LIST OFF NEXT 10 &list_flds.
      WAIT "Press spacebar to continue or R to return to OPTION MENU" ;
           TO answer
      CLEAR
   ENDSCAN
   DEACTIVATE WINDOW lister
   GO record_num
RETURN
```

If there are two windows on the screen and you DEACTIVATE the active one, the remaining window is ACTIVATEd automatically. If two or more windows remain, the most recently ACTIVATEd one becomes the new active window.

When you DEACTIVATE a WINDOW, the window disappears from the screen but not from memory. The Managing Window Definitions section explains how to erase window definitions from memory.

Activating the Screen

Occasionally, you will need to write to the screen outside the current window. The command ACTIVATE SCREEN resets the full screen coordinates but leaves the previously active window on the screen. Output goes to the full screen. Any output falling within the boundaries of the previously active window displays on top of the window, and is lost when you reactivate or deactivate the window.

Certain commands, such as ?/??, LIST, or CLEAR, can cause the screen to scroll. In that case, the previously active window scrolls out of view.

When you want all output to display inside the window again, simply ACTIVATE the WINDOW. The window appears at its original screen coordinates.

Managing Window Definitions

If you plan to use a set of window definitions in several applications, SAVE them to a disk file. dBASE IV attaches a default .win extension to the window definitions filename.

```
SAVE WINDOW alert, lister TO Win_defs
```

To save all windows, *SAVE WINDOW ALL TO Win_defs*. You can then RESTORE the definitions when you want to use the windows in a program. The window that was active when you SAVEd the definitions will again be active. You can restore specified windows, or all windows using ALL.

```
RESTORE WINDOW alert, lister FROM Win_defs
```

WARNING
The RESTORE WINDOW command overwrites any window definitions in memory.

Once you SAVE your window definitions to a file, you can CLEAR or RELEASE them during your application without losing the definitions. This increases the amount of available memory. Use CLEAR WINDOWS to erase *all* windows from the screen and remove their definitions from memory. Use RELEASE WINDOWS to erase *specified* windows from the screen and remove their definitions from memory:

```
RELEASE WINDOWS alert, lister
```

More Uses of Windows

There are many ways other than the LIST window shown above to use windows in application programs. This section discusses three other popular window arrangements:

- Pick lists
- Memo fields in a window
- Dialog boxes containing choices or information prompts

The Providing Menu Help section in Chapter 7 shows how to code a help window.

Pick Lists

You can provide a look-up window to display information not contained in the main database file. This lets users access data from other files, such as employee names, department codes, or part numbers. Figure 6-5 shows a part number look-up pick list.

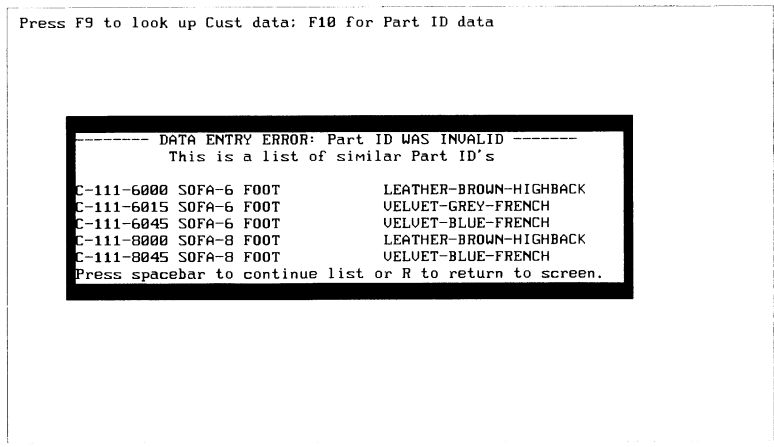


Figure 6-5 Look-up window

The code for this example is in Chapter 8 in the Checking Against a Database File section. The previous example in that chapter, in the Checking Against a List section, also uses a look-up window.

Memo Windows

You can display the contents of a memo field in a window, for users to read or edit (see Figure 6-6).

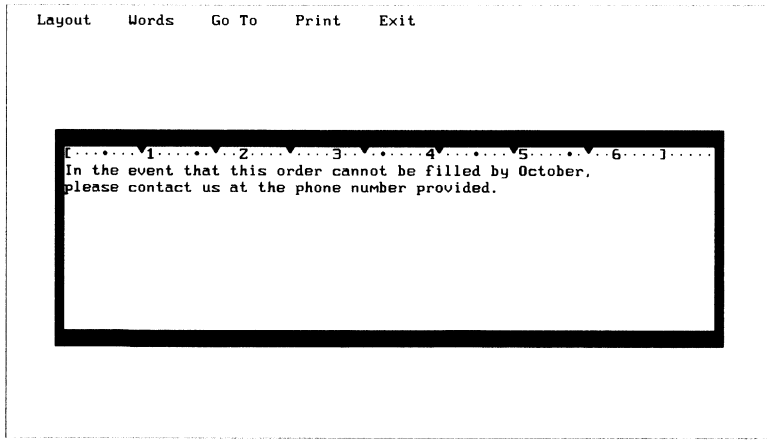


Figure 6-6 Memo window

The following lines of code open the window shown and display the Notes memo field in the window when the user presses **Ctrl-Home**. When the user presses **Ctrl-End**, the window closes and any changes are saved. This code fragment would normally be part of a screen format file.

```
DEFINE WINDOW memo_windo FROM 7,4 TO 19,75 PANEL COLOR &c_list.
@ 17,14 SAY "NOTES: " GET Notes WINDOW memo_windo
```

Dialog Boxes

Dialog boxes display prompts with options for user responses. The dialog box shown in Figure 6-7 permits users to skip back and forth in the database file.

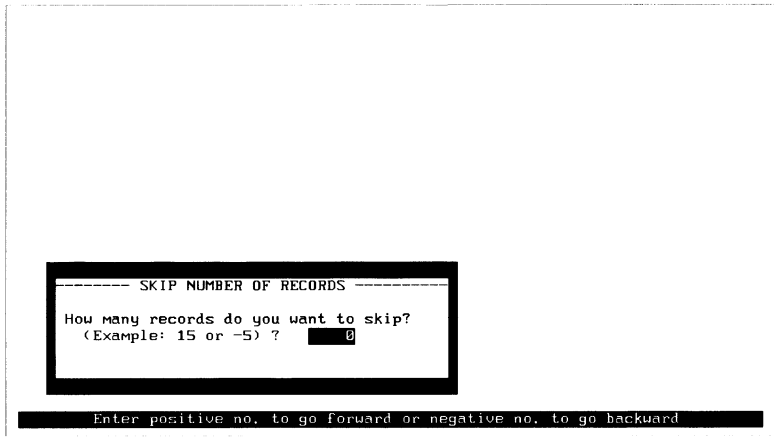


Figure 6-7 Dialog box

The code for this dialog box is part of the Skip_rec procedure in Library.prg. First it opens the *alert* window and prompts the user for the number of records to skip. Then it stores the user's entry in the variable *numb_2skip,* closes the window, and SKIPs the specified number of records.

```
ACTIVATE WINDOW alert
@ 0,0 SAY "------------ SKIP NUMBER of RECORDS -----------"
@ 2,1 SAY "How many records do you want to skip?"
@ 3,0 SAY "   (Example: 15 or -5) ?   " GET numb_2skip PICTURE "9999";
  MESSAGE "Enter positive # to go forward or negative # to go backward"
READ
DEACTIVATE WINDOW alert
SKIP numb_2skip
```

Working with Borders

The SET BORDER command is used to specify the characters used in borders for menus, windows, boxes, and lines. You can define four types of borders:

- Single line (the default — SET BORDER TO SINGLE)
- Double line (SET BORDER TO DOUBLE), may display as a single line on your terminal
- Inverse video panel (SET BORDER TO PANEL)
- Invisible border (SET BORDER TO NONE)

You can also design artistic borders using a border definition string with the SET BORDER command. For example, the following commands first restore the borders to their single-line default values, and then define single-lined borders with double-lined corners. The parameters refer to the top, bottom, left, and right sides of the rectangle, and then to its top left, top right, bottom left, and bottom right corners. The numbers are the ASCII values for the characters to appear. (Using a comma leaves a setting unchanged.)

```
SET BORDER TO
SET BORDER TO ,,,,201, 187, 200, 188
```

Options of the DEFINE BOX and DEFINE WINDOW commands override the global settings provided by SET BORDER, as shown earlier in this chapter.

Working with Colors

Color settings will only have effect on a color monitor, and are ignored on a monochrome system. Even if you're working on a monochrome system, you may want to use color settings in your application so that it's portable to color monitor systems.

Some of the code examples in this chapter include color settings. These examples use the COLOR option of the individual commands to set colors. dBASE IV also has global color commands, such as SET COLOR TO, SET COLOR OF...TO, and @...FILL TO.

Table 6-1 shows how to set the colors of objects you might use in an application. It does not include options for controlling the color of Control Center objects. If your application lets users access the Control Center, use SET COLOR OF and settings in your Config.db file to set colors throughout the menu system. See SET COLOR in *Language Reference*.

Table 6-1 Setting colors

To color	Use
Screen forms	
All parts of form	SET COLOR TO
@...SAYs	@...SAY...GET...COLOR
GET fields	@...SAY...GET...COLOR or SET COLOR OF FIELDS
User-entered text	@...SAY...GET...COLOR (enhanced option)
Message line	SET COLOR OF MESSAGE
Navigation line	SET COLOR OF MESSAGE
Error messages	SET COLOR OF MESSAGES
Help messages	SET COLOR OF MESSAGES
Lines	@...TO COLOR
Boxes	
Border	@...TO COLOR
Interior	@...FILL TO...COLOR
Text inside	SET COLOR OF NORMAL
Windows	
Border	DEFINE WINDOW...COLOR
Interior	DEFINE WINDOW...COLOR
Text inside	SET COLOR OF NORMAL
Bar menus	
Menu options	SET COLOR OF MESSAGES (bright)
Dimmed options	SET COLOR OF MESSAGES (dim)
Selected options	SET COLOR OF HIGHLIGHT
Pop-up menus and lists	
Border	SET COLOR OF BOX
Menu options	SET COLOR OF MESSAGES (bright)
Dimmed options	SET COLOR OF MESSAGES (dim)
Selected options	SET COLOR OF HIGHLIGHT

(continued)

Table 6-1 Setting colors *(continued)*

To color	Use
Dialog Boxes	
Border	SET COLOR OF BOX
Entry blank	SET COLOR OF FIELDS
Clock	SET COLOR OF INFORMATION
Status bar	SET COLOR OF INFORMATION

This excerpt from the Set_env procedure in Library.prg provides the basic color settings used throughout the sample Business application. The ISCOLOR() function determines whether the user's system has a color card. See SET COLOR in *Language Reference* for an explanation of the individual color settings.

```
PUBLIC c_standard, c_data, c_fields, c_popup, c_alert, c_list, c_red, c_blue
PUBLIC c_yellow, c_yelowhit, c_green, c_blink
IF ISCOLOR()
   c_standard = "W/B, BG+/R, B"
   c_data = "B/W, R/BG, B"
   c_fields = "B/BG"
   c_popup = "B/W, GR+/R"
   c_alert = "GR+/R, B/W, R/G"
   c_list = "W+/G, GR+/B, GR+/GR"
   c_red = "R/W"
   c_blue = "B/W"
   c_yellow = "GR+/B"
   c_yelowhit = "GR+/W"
   c_green = "G/W"
   c_blink = "GR+*/B"
ELSE
   STORE "W+/N, N/W" TO c_standard, c_data, c_fields, c_popup, c_alert, c_list
   STORE "W" TO c_red, c_blue, c_yellow, c_yelowhit, c_green
   c_blink = "W+*/N, N/W"
ENDIF
```

The easiest way to set colors for a screen form is to issue a SET COLOR TO command before the screen form code, and then reset the colors afterward:

```
SET COLOR TO &c_data.
* <screen form code>
SET COLOR TO &c_standard
```

To vary from the SET COLOR TO definitions without changing the global settings, define a box around the screen form and color it with @...FILL TO. The next example, which comes from the Backgrnd procedure in Vendors.prg, shows the lines, boxes, and fills used in the Vendors screen form. Note that three separate @...FILL TOs are required to fill in the background around the title, the VENDOR NUMBER prompt, and the data entry area.

TIP

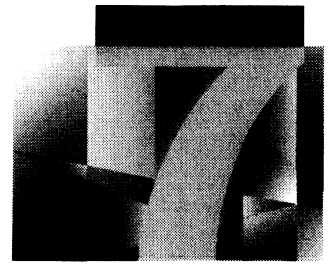
You should generally code lines and boxes first, then fills, and finally @...SAYs. Put line code before box code so that a line extending to the box borders doesn't break the border. You can control whether the box fill color bleeds into the box border by careful definition of the fill coordinates. For example, the box coordinates in the code below are 8, 4 and 19, 53 while the fill coordinates are 9, 5 and 18, 52. This places the fill color within the box perimeter rather than over it.

```
@ 14,5 TO 14,52 COLOR &c_red.
@ 1,22 TO 3,53 DOUBLE COLOR &c_blue.
@ 5, 4 TO 7,27 DOUBLE COLOR &c_red.
@ 8, 4 TO 19,53 COLOR &c_red.
@ 2,23 FILL TO 2,52 COLOR &c_blue.
@ 6, 5 FILL TO 6,26 COLOR &c_red.
@ 9, 5 FILL TO 18,52 COLOR &c_red.
<@...SAYs defining screen background>
```

This last routine displays customer balances of $1,000 or over in red, balances between $100 and $1,000 in yellow on white, and balances under $100 in green.

```
IF ISCOLOR()
   DO CASE
      CASE oldbalance >= 1000
         bal_color = c_red
      CASE oldbalance >= 100
         bal_color = c_yelowhit
      CASE oldbalance < 100
         bal_color = c_green
   ENDCASE
ELSE
   bal_color = "W"
ENDIF
@ 14,45 SAY m->oldbalance PICTURE "999,999.99" COLOR &bal_color.
```

Designing Menus and Lists



This chapter explains how to use the dBASE IV menu-building commands to create a menu interface for your application. If you prefer, you can use the Applications Generator provided with dBASE IV to build menu interfaces (see *Using dBASE IV).*

What This Chapter Covers

The topics covered in this chapter are:

- Defining a pop-up menu
- Defining a horizontal bar menu
- Defining a pull-down menu
- Defining a list
- Defining a main menu

The emphasis is on how to build a menu system, rather than how to design one. See the Using the Applications Generator section in *Using dBASE IV* for a discussion of the principles of menu design.

Overview

The dBASE IV menu-building commands and functions can be used to design a variety of menu objects. Figure 7-1 identifies these objects in a generic menu interface. Note the distinction between a *horizontal bar menu* and *pop-up* or *pull-down* menus.

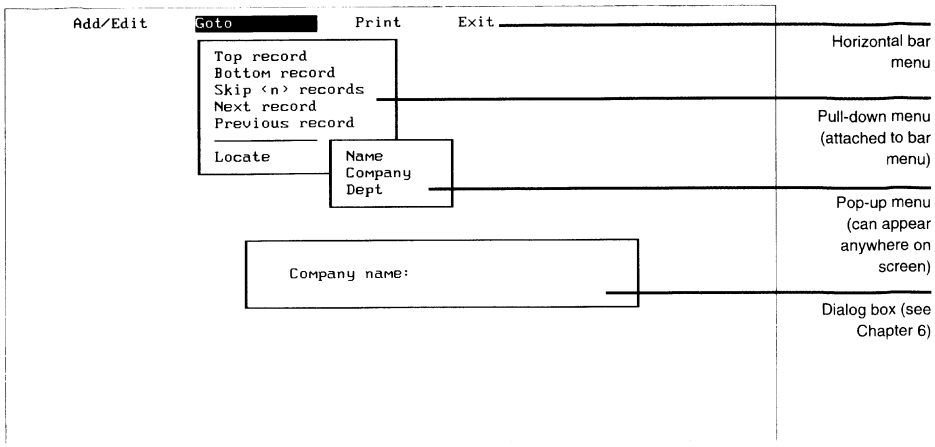


Figure 7-1 Names of menu objects

The next several sections of this chapter show how to build pop-up menus. The Defining Other Menu Objects section shows how to code horizontal bar menus, pull-down menus, lists, and an application main menu.

The main steps in coding a menu system are:

1. Define each menu and its items.
2. Specify menu item actions.
3. Activate the main menu.
4. Deactivate and release menus when no longer needed.

You can save development time by using the Applications Generator to define the menu system for your application. The Applications Generator creates .prg files containing dBASE code, which you can edit directly.

WARNING
As with memory variables, dBASE IV overwrites menu and menu item definitions of the same name without notification.

If you change a .prg file in the program editor and subsequently make and save changes in the Applications Generator, your changes to the .prg file will be overwritten. To save development time, build the menu system in the Applications Generator. Then fine-tune it with the program editor.

Defining a Pop-Up Menu

A *pop-up menu* is a list of selectable items framed by a border. Users select items by typing the first letter of the item name or moving the highlight to the item and pressing ↵.

Defining the Pop-Up Menu and Its Items

DEFINE POPUP and DEFINE BAR define a pop-up menu and its items but don't define item actions. The following excerpt from the Bar_def procedure in Library.prg defines the pop-up menu used throughout the Business application.

The DEFINE POPUP statement provides the upper left and lower right screen coordinates for the popup. The DEFINE BAR statements define the menu prompts. They refer to the lines of the pop-up menu by number: the first (top) line is bar 1, the second bar 2, and so on. Be sure to number the items in vertical order of appearance; the highlight moves in the menu in ascending numeric order.

Note that bar 1 in this example defines the menu title rather than an item. The SKIP keyword causes the highlight to skip over this title, since it is not a selectable menu item.

dBASE IV displays the prompts flush left against the pop-up border. If you want a space between the border and the first character of the prompt, include it in the prompt string as shown.

```
PROCEDURE Bar_def
  DEFINE POPUP main_mnu FROM 2,58 TO 22,78;
  MESSAGE "Press first letter of menu choice, or highlight and press <Enter>"
  DEFINE BAR  1 OF main_mnu PROMPT "==  OPTION MENU  ==" SKIP
  DEFINE BAR  2 OF main_mnu PROMPT " Add record"
  DEFINE BAR  3 OF main_mnu PROMPT " Edit record"
  DEFINE BAR  4 OF main_mnu PROMPT " Delete record"
  DEFINE BAR  5 OF main_mnu PROMPT "------------------" SKIP
  DEFINE BAR  6 OF main_mnu PROMPT " Next record"
  DEFINE BAR  7 OF main_mnu PROMPT " Previous record"
  DEFINE BAR  8 OF main_mnu PROMPT " Top record"
  DEFINE BAR  9 OF main_mnu PROMPT " Bottom record"
  DEFINE BAR 10 OF main_mnu PROMPT " Skip records"
  DEFINE BAR 11 OF main_mnu PROMPT " Find record"
  DEFINE BAR 12 OF main_mnu PROMPT "------------------" SKIP
  DEFINE BAR 13 OF main_mnu PROMPT " List records"
  DEFINE BAR 14 OF main_mnu PROMPT " Output reports"
  DEFINE BAR 15 OF main_mnu PROMPT;
  " Group records" SKIP FOR dbf =  "ACCT_REC"
  DEFINE BAR 16 OF main_mnu PROMPT " Count records"
  DEFINE BAR 17 OF main_mnu PROMPT " Index database"
  DEFINE BAR 18 OF main_mnu PROMPT " Help"
  DEFINE BAR 19 OF main_mnu PROMPT " Quit to MAIN MENU"
  .
  .
  .
RETURN
```

Variation — Putting Blank Lines Between Items

If you want a blank line to appear between two menu items, omit the corresponding bar number. This causes the highlight to automatically skip over the intervening blank line. For example, define bar 13 immediately after bar 11 to skip bar 12.

Variation — Skipping Based on a Condition

You can make the highlight skip a particular item under specified conditions. The following modifications of the above code make the Find, Edit, and Count items unavailable when there are no records in the database file:

```
DEFINE BAR 3 OF main_mnu PROMPT "Edit record";
   SKIP FOR RECCOUNT() = 0
DEFINE BAR 11 OF main_mnu PROMPT "Find record";
   SKIP FOR RECCOUNT() = 0
DEFINE BAR 16 OF main_mnu PROMPT "Count records";
   SKIP FOR RECCOUNT() = 0
```

Specifying Menu Actions

ON SELECTION POPUP specifies a procedure that contains the instructions for each menu item action. A menu item can DO a procedure, execute a dBASE IV command, or even ACTIVATE another pop-up menu (see the Activating the Pop-up Menu section).

In the following code, the CASE statements in Barpop define the actions for each item of *main_mnu.* Because bar 1 is the menu title, the numbering of the CASEs starts with bar 2. Only a few of the CASEs are shown here. See the sample code for the complete Barpop procedure and the many procedures called by it.

```
ON SELECTION POPUP main_mnu DO Barpop

PROCEDURE Barpop
   DO CASE
      CASE BAR() = 2        && BAR() = 1 is title
         DO Add_new
      CASE BAR() = 3
         DO Edit
      CASE BAR() = 4
         DO Eraser
      CASE BAR() = 6        && Next record
         DO Skip_rec WITH 1
      CASE BAR() = 7        && Previous record
         DO Skip_rec WITH -1
      CASE BAR() = 8        && Top record, in active index order
         GO TOP
      CASE BAR() = 9        && Bottom record, in active index order
         GO BOTTOM
   <additional CASE statements>
   ENDCASE
   .
   .
   .
RETURN
```

Activating the Pop-Up Menu

Only one line of code is required to activate the defined pop-up menu:

```
ACTIVATE POPUP main_mnu
```

Be sure to define the pop-up menu before activating it. The order that the code must follow is:

```
DO Bar_def
ON SELECTION POPUP main_mnu DO Barpop
ACTIVATE POPUP main_mnu

PROCEDURE Bar_def
   *<procedure code shown above>
RETURN

PROCEDURE Barpop
   *<procedure code shown above>
RETURN
```

dBASE IV checks your pop-up coordinates when you ACTIVATE the POPUP, not when you DEFINE it.

Deactivating and Releasing the Pop-Up Menu

When you ACTIVATE a POPUP in your menu system, dBASE IV leaves any popups already activated on the screen and places the new one on top of them. It is up to you to DEACTIVATE POPUPs when you no longer want to display them.

When you DEACTIVATE the POPUP, the pop-up menu and *any menus and procedures called by it* terminate. Remaining code in a procedure is not executed (as if you had RETURNed from the procedure). DEACTIVATE POPUP returns control to the level at which the pop-up menu was called. If the popup was activated with ON PAD, DEACTIVATE POPUP returns control to the menu bar.

CLEAR or RELEASE menu definitions from memory when you no longer need them. You cannot CLEAR or RELEASE menus while they are active, so DEACTIVATE them first.

Providing Menu Help

Use ON KEY with the menu-building commands to provide context-sensitive menu help to users. The following routine (not in Business) provides help on a given menu item when the highlight is on it and the user presses **F1 Help**. The POPUP() function identifies the various menus in the system, while the BAR() function identifies the items in each menu.

```
ON KEY LABEL F1 DO Menuhelp

PROCEDURE Menuhelp
   ACTIVATE WINDOW Help_w
   DO CASE
      CASE POPUP() = "MAIN_MNU"
         DO CASE
            CASE BAR() = 19
               ? "Exit to Main Menu"
            CASE BAR() = 2
               ? "Add new record to database file in index order"
            CASE BAR() = 3
               ? "Find or retrieve record by searching key field"
            .
            . <BAR()s and help text for other items>
            .
         ENDCASE
      CASE POPUP() = "REPT_MNU"
         DO CASE
            CASE BAR() = 4
               ? "Exit to Option menu"
            CASE BAR() = 2
               ? "Print list of items from this database file"
            CASE BAR() = 3
               ? "Print mailing list for this database file"
         ENDCASE
      .
      . <POPUP()s and CASE statements for other menus>
      .
   ENDCASE
   WAIT "Press any key to exit Help"
   DEACTIVATE WINDOW Help_w
RETURN
```

Defining Other Menu Objects

This section shows how to define a horizontal bar menu, a pull-down menu, a list, and a main application menu.

NOTE
Many applications use dialog boxes to prompt the user for information. The More Uses of Windows section in Chapter 6 shows how to code a dialog box.

Horizontal Bar Menus

You can define a *horizontal bar menu* that runs across the top of the screen, similar to many spreadsheet menus. Figure 7-2 shows a horizontal bar menu.

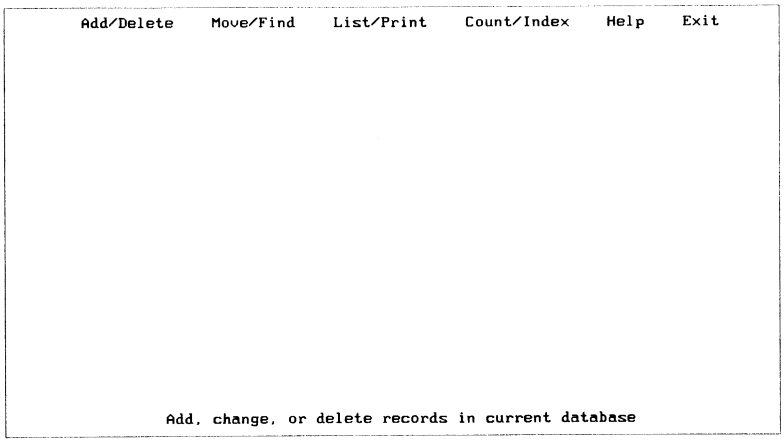


Figure 7-2 Horizontal bar menu

The dBASE IV menu-building commands for horizontal bar menus are similar to those for pop-up menus, as you can see in Table 7-1.

Table 7-1 Commands for horizontal bar and pop-up menus

Operation	Horizontal Bar Menu	Pop-up Menu
Naming the menu	DEFINE MENU	DEFINE POPUP
Naming the prompts	DEFINE PAD	DEFINE BAR
Defining the actions	ON MENU	ON BAR
	ON PAD	ON POPUP
	ON SELECTION MENU	ON SELECTION BAR
	ON SELECTION PAD	ON SELECTION POPUP
	ON EXIT MENU	ON EXIT BAR
	ON EXIT PAD	ON EXIT POPUP
Opening the menu	ACTIVATE MENU	ACTIVATE POPUP
Showing an inactive menu	SHOW MENU	SHOW POPUP
Closing the menu	DEACTIVATE MENU	DEACTIVATE POPUP
Releasing the menu definitions	CLEAR MENUS or RELEASE MENUS	CLEAR POPUPS or RELEASE POPUPS
Returning the menu name	MENU()	POPUP()
Returning the internal item	PAD()	BAR()
Returning the external item name of the last selection	PROMPT()	PROMPT()
Returning the external item name of the active or specified internal item	PADPROMT()	BARPROMPT()
Returning the number of items		BARCOUNT()

The following procedure defines the horizontal bar menu shown in Figure 7-2, a variation of the main pop-up menu used throughout the Business application.

The DEFINE MENU and DEFINE PAD commands define the bar menu and its items. The AT coordinates in this example position the menu items on the second row of the screen. dBASE IV automatically adds a blank space to the left and right of the item name. The order in which you DEFINE the PADs in your code determines the order in which the highlight accesses them.

The ON SELECTION PAD commands ACTIVATE a pop-up menu or DO a procedure for each item in the bar menu. (Assume that you have already defined these popups and procedures in your code.)

```
PROCEDURE Menu_def
   DEFINE MENU barmenu
   DEFINE PAD add_del OF barmenu PROMPT "Add/Delete" AT 2,1;
      MESSAGE "Add, change, or delete records in current database"
   DEFINE PAD movefind OF barmenu PROMPT "Move/Find" AT 2,14;
      MESSAGE "Go to selected record"
   DEFINE PAD listprin OF barmenu PROMPT "List/Print" AT 2,26;
      MESSAGE "List data to screen, file, or printer"
   DEFINE PAD cnt_indx OF barmenu PROMPT "Count/Index" AT 2,39;
      MESSAGE "Count records or index the database file"
   DEFINE PAD help OF barmenu PROMPT "Help" AT 2,53;
      MESSAGE "Get help on using this program"
   DEFINE PAD exit OF barmenu PROMPT "Exit" AT 2,61;
      MESSAGE "Exit this program"

  ON SELECTION PAD add_del OF barmenu ACTIVATE POPUP add_del
  ON SELECTION PAD movefind OF barmenu ACTIVATE POPUP movefind
  ON SELECTION PAD listprin OF barmenu ACTIVATE POPUP listprin
  ON SELECTION PAD cnt_indx OF barmenu ACTIVATE POPUP cnt_indx
  ON SELECTION PAD help OF barmenu DO Helper
  ON SELECTION PAD exit OF barmenu DO Exit_now
  ACTIVATE MENU barmenu PAD add_del
RETURN
```

TIP

PAD() returns the internal name (assigned in the DEFINE PAD statement) of the menu item most recently selected by the user. You can use it to verify a questionable choice of a menu item:

```
@...SAY "Are you sure you want to" + PAD() + "(Y/N)?";
    GET choice
READ
```

If the user presses **Esc** *to exit the menu, PAD() returns a null string.*

Pull-Down Menus

A pull-down menu is a pop-up menu that pulls down from a horizontal bar menu, as used in the Control Center. Figure 7-3 shows the bar menu from Figure 7-2 with the first pull-down menu open.

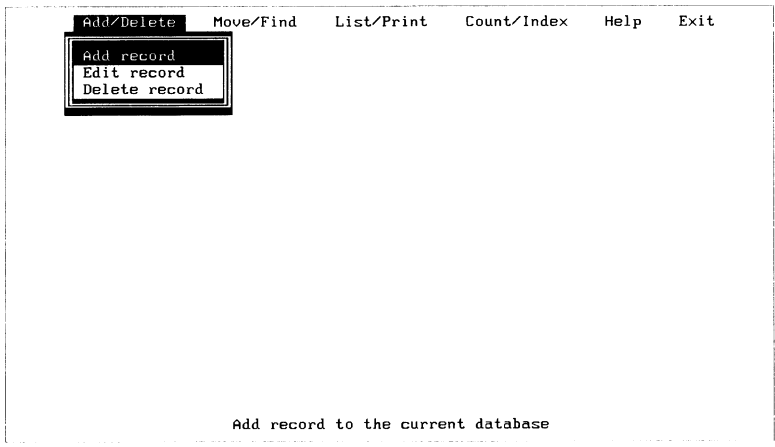


Figure 7-3 Pull-down menu

Pull-down menus differ from pop-up menus in one essential respect. Simply moving the highlight to an item on the bar menu without pressing ↵ opens the attached pull-down. In a pop-up menu (or a horizontal bar menu without attached pull-downs), the user must *select* an item to display the submenu.

The following code fragment replicates the last part of the previous routine, except that the first four ON SELECTION PAD commands have been changed to ON PADs. This causes the corresponding pull-downs to appear automatically when the user moves the highlight to the corresponding bar menu item.

```
ON PAD add_del OF barmenu ACTIVATE POPUP add_del
ON PAD movefind OF barmenu ACTIVATE POPUP movefind
ON PAD listprin OF barmenu ACTIVATE POPUP listprin
ON PAD cnt_indx OF barmenu ACTIVATE POPUP cnt_indx
ON SELECTION PAD help OF barmenu DO Helper
ON SELECTION PAD exit OF barmenu DO Exit_now
```

Lists

To dBASE IV, a *list* is a special kind of pop-up menu that contains variable items rather than fixed ones. You can create three kinds of lists:

- File lists, containing the names of the files on disk or in the current catalog
- Field lists, containing the names of the active fields defined by SET FIELDS
- Value lists, containing the contents of a field in each record

Use DEFINE POPUP with a PROMPT argument to create a list such as the following file list:

```
DEFINE POPUP files FROM 2,20 TO 20,50 PROMPT FILES LIKE *.DBF
```

You don't need to define the items in a list explicitly, as you did for pop-up menu items. Once you've specified the type of list you want, dBASE IV provides the items. Since a list is considered a type of pop-up menu, you display a list with ACTIVATE POPUP.

Main Menus

A *main menu* is the top-level menu of an application. It calls all other menus in the program, and users enter and exit the program through the main menu. The Business application uses the main menu shown in Figure 7-4.

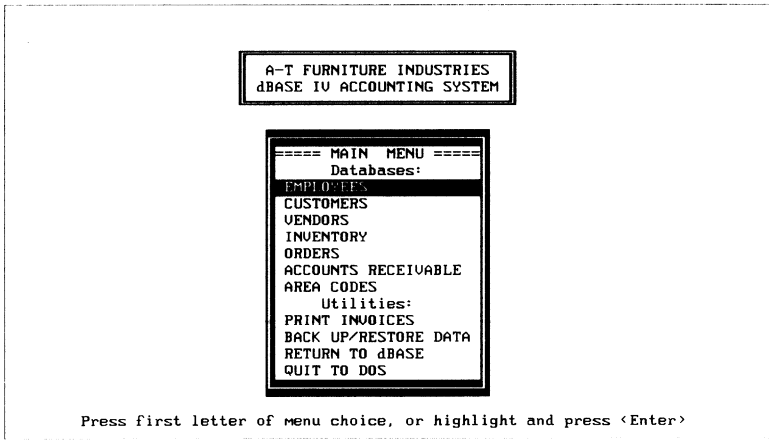


Figure 7-4 Business.prg main menu

The code for this main menu appears below. The DEFINE BAR statements in Main_def define the menu options. Then the CASE statements in Main execute the appropriate procedure for each menu choice. The sample code contains the complete code for both procedures and the many procedures called by Main.

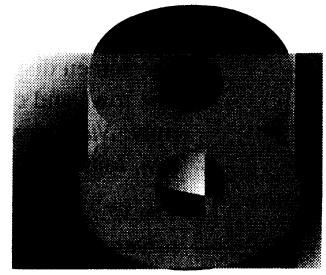
```
DO Main_def
DO Title
ACTIVATE POPUP mainmenu

PROCEDURE Title
   CLEAR
   @ 2,24 TO 5,53 DOUBLE COLOR &blue.
   @ 2,24 FILL TO 5,53 COLOR &blue.
   SET COLOR TO &red.
   @ 3,27 SAY "A-T FURNITURE INDUSTRIES"
   @ 4,26 SAY "dBASE IV ACCOUNTING SYSTEM"
   SET COLOR TO & c_normal.
RETURN

PROCEDURE Main_def
   DEFINE POPUP mainmenu FROM 7,27 TO 22,50;
         MESSAGE;
         "Press first letter of menu choice, or highlight and press <Enter>"
   DEFINE BAR  1 OF mainmenu PROMPT "===== MAIN MENU =====" SKIP
   DEFINE BAR  2 OF mainmenu PROMPT "       Databases:" SKIP
   DEFINE BAR  3 OF mainmenu PROMPT " EMPLOYEES"
   DEFINE BAR  4 OF mainmenu PROMPT " CUSTOMERS"
   DEFINE BAR  5 OF mainmenu PROMPT " VENDORS"
   DEFINE BAR  6 OF mainmenu PROMPT " INVENTORY"
   DEFINE BAR  7 OF mainmenu PROMPT " ORDERS"
   DEFINE BAR  8 OF mainmenu PROMPT " ACCOUNTS RECEIVABLE"
   DEFINE BAR  9 OF mainmenu PROMPT " AREA CODES"
   DEFINE BAR 10 OF mainmenu PROMPT "     Utilities:" SKIP
   DEFINE BAR 11 OF mainmenu PROMPT " PRINT INVOICES"
   DEFINE BAR 12 OF mainmenu PROMPT " BACK UP/RESTORE DATA"
   DEFINE BAR 13 OF mainmenu PROMPT " RETURN TO dBASE"
   DEFINE BAR 14 OF mainmenu PROMPT " QUIT dBASE"
   ON SELECTION POPUP mainmenu DO Main
RETURN

PROCEDURE Main
   DO CASE
      CASE BAR() = 3
         DO Employee
      CASE BAR() = 4
         DO Cust
      CASE BAR() = 5
         DO Vendors
         .
         . <remaining CASE statements>
         .
   ENDCASE
RETURN
```

Traditional Main Menu Code

If you're maintaining a dBASE III PLUS program, you might encounter this traditional code for a main menu. To select an item, the user presses the indicated number or letter rather than the first letter of the item name.

```
DO WHILE .T.
 SET COLOR TO & c_standard
 CLEAR
 @ 0,0
 TEXT
                            A-T FURNITURE INDUSTRIES
                            dBASE IV ACCOUNTING SYSTEM

                                    MAIN MENU

                            Databases:
                            1.   EMPLOYEES
                            2.   CUSTOMERS
                            3.   VENDORS
                            4.   INVENTORY
                            5.   ORDERS
                            6.   ACCOUNTS RECEIVABLE
                            7.   AREA CODES

                            Utilities:
                            P.   PRINT INVOICES
                            B.   BACK UP/RESTORE DATA
                            R.   RETURN TO dBASE
                            Q.   QUIT dBASE
 ENDTEXT

 choice = " "
 @ 22,29 SAY "CHOICE: " GET choice PICTURE "!";
    VALID choice $ "1234567PBRQ";
    MESSAGE "Please choose a database or utility"
 READ

 DO CASE
    CASE choice = "Q"
       QUIT
    CASE choice = "1"
       DO Employee
    CASE choice = "2"
       DO Cust
    .
    . <remaining CASE statements>
    .
 ENDCASE
ENDDO
CLEAR
```

Input: Getting Data from the User



To get information from someone, you ask questions and receive answers. Application programs also ask questions and get answers, using the screen and keyboard to communicate. They format the data entered by the user for display, evaluate it to prevent data entry errors, and enter the validated data into the database.

What This Chapter Covers

This chapter describes methods for getting information from the user with dBASE IV application programs. The specific topics covered in this chapter are:

- Screen coordinates and relative addressing
- Getting data with screen forms
- Getting data from an array
- Formatting data entry screens
- Customizing Browse tables
- Formatting data
- Checking data for errors
- Processing keypresses
- Providing help for the user
- Entering data into and deleting data from the database

Screen Coordinates and Relative Addressing

The screen display is a grid of evenly spaced horizontal rows and vertical columns. The rows and columns are numbered from the top of the screen down and from left to right. *Numbering starts with 0, rather than 1.* For example, *1,3* refers to the second row, fourth column.

The @...SAY...GET command positions characters on the screen by providing their *screen coordinates.* A screen coordinate consists of the row number followed by the column number. For example, @ *1,5 SAY <expC>* positions the character string on the second row and the sixth column.

dBASE IV uses the ROW() and COL() functions for *relative addressing*, that is, to a position relative to the current coordinates. ROW() and COL() return the current screen row and column coordinates. *ROW() + 1* means one row below the current row. You can substitute the $ symbol for either ROW() or COL().

```
@ ROW(), COL() SAY <exp1>          && Or: @ $,$ ...
@ ROW()+1, COL()+3 SAY <exp2>      && Or: @ $+1, $+3 ...
```

When using relative addressing, be careful not to exceed the screen boundaries.

Getting Data with Screen Forms

Screen forms arrange prompts and blanks on the screen to simulate paper forms already familiar to users. Use the dBASE IV forms design screen to prototype screen forms without having to provide exact screen coordinates. You can access it from the Control Center, or with the command CREATE SCREEN <filename> at the dot prompt. Chapter 9 of *Using dBASE IV* explains how to build screen forms with the forms generator.

The forms generator produces three files: a forms design (.scr) file containing the internal code used by the generator, a format (.fmt) file containing dBASE code, and a format object (.fmo) file containing compiled dBASE code. You can change the code in the .fmt file directly, using the program editor.

If you change the .fmt file directly and subsequently make and save changes on the forms design screen, your changes to the .fmt file will be overwritten. To save development time, build the form on the forms design screen, then fine-tune it in the program editor.

Looking at the Code from a Screen Form

After prototyping your screen form in the forms generator, you can edit the code with MODIFY COMMAND filename.fmt. For example, compare the Vendor screen form shown in Figure 8-1 with the generated code following the figure.

The format file has three parts: initialization code, processing code, and exit code. The *initialization* code sets up the form environment. The *processing* code determines how the form appears on the screen. The *exit* code restores the original environment.

In the processing section, the GETs store data entered by the user into fields in the database file, or into memory variables defined previously. The order of the GETs in the format file determines the order in which the cursor advances through the fields on the form.

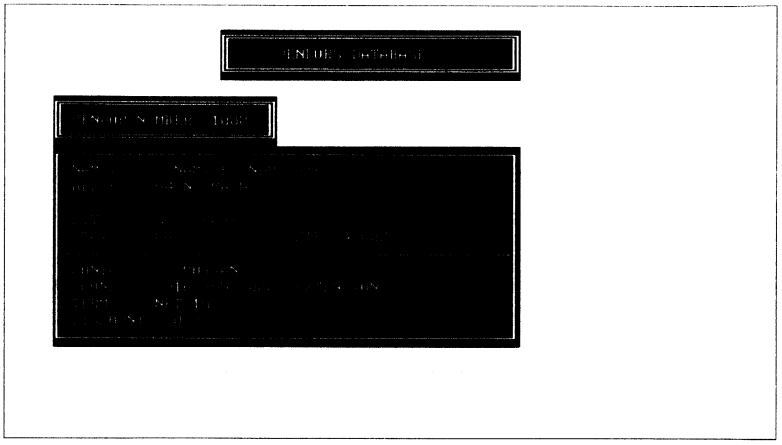


Figure 8-1 Vendor screen form

The screen procedure in Vendors.prg contains custom code comparable to the processing code shown here (see the sample code).

```
*-- Format file initialization code ---------------------------

IF SET("TALK")="ON"
   SET TALK OFF
   lc_talk="ON"
ELSE
   lc_talk="OFF"
ENDIF

*-- This form was created in MONO mode
SET DISPLAY TO MONO
lc_status=SET("STATUS")
*-- SET STATUS was OFF when you went into the Forms Designer.
IF lc_status = "ON"
   SET STATUS OFF
ENDIF

*-- @ SAY GETS Processing -------------------------------------

*-- Format Page: 1
@ 1,22 TO 3,53
@ 1,28 SAY "VENDOR NUMBER:"
@ 6,21 GET vendor_id PICTURE "9999";
     MESSAGE "Enter a four digit vendor or Esc to quit"
@ 8,4 TO 19,53
@ 9,6 SAY "NAME:"
@ 9,15 GET vendor PICTURE "@! XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
@ 10,6 SAY "ADDRESS:"
@ 10,15 GET address1 PICTURE "@! XXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
@ 11,15 GET address2 PICTURE "@! XXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
@ 12,6 SAY "CITY:"
@ 12,15 GET city PICTURE "!XXXXXXXXXXXXXXXXXX"
@ 13,6 SAY "STATE:"
@ 13,15 GET state PICTURE "!!"
@ 13,30 SAY "ZIP:"
@ 13,35 GET zip PICTURE "XXXXXXXXXX"
@ 14,5 TO 14,52 COLOR r/w
@ 15,6 SAY "CONTACT:"
@ 15,15 GET contact PICTURE "@! XXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
@ 16,6 SAY "PHONE:"
@ 16,15 GET phone PICTURE "(999)999-9999"
@ 16,30 SAY "EXTENSION:"
@ 16,41 GET phone_ext PICTURE "9999"
@ 17,6 SAY "TERMS:"
@ 17,15 GET terms PICTURE "@! XXXXXXXXXX"
@ 18,6 SAY "DISCOUNT:"
@ 18,16 GET discount PICTURE "99";
   MESSAGE "Enter a discount rate (max. 99)"
@ 18,19 SAY "%"

*-- Format file exit code -------------------------------------

*-- SET STATUS was OFF when you went into the Forms Designer.
IF lc_status = "ON"  && Entered form with status on
   SET STATUS ON     && Turn STATUS "ON" on the way out
ENDIF

IF lc_talk="ON"
   SET TALK ON
ENDIF

RELEASE lc_talk,lc_fields,lc_status
*-- EOP: VENDORS.FMT
```

You can create forms of up to 10 pages in the forms generator. The forms generator inserts a page break automatically after each full screen. To define a short page, place a READ command in the .fmt file wherever you want a page break to appear.

Opening and Closing Screen Format Files

You can activate a screen form in a program in one of the following ways:

- Create the form on the forms design screen. In your code, SET FORMAT TO <.fmt filename>. (This creates a compiled .fmo file when you DO the program.)
- Create the form on the forms design screen. Edit a copy of the .fmt file in the program editor and copy the resulting code into your program.
- Write a custom forms procedure in your program code. Use the command DO <.prg filename> to activate the form.

The first method calls an external format file and thus saves some lines of code in your program. The last two methods incorporate the form code directly into your program so it needn't call an external program file. These methods ensure that the form code is always available and make your program more efficient.

When you use SET FORMAT TO <.fmt filename>, the full-screen commands APPEND, EDIT, CHANGE, and READ display the current record in the form for data entry purposes (see the Entering Data into the Database section). CLOSE FORMAT, SET FORMAT TO with no argument, and USE with no argument close the current format file. CLEAR ALL and CLOSE DATABASES close all open format files.

NOTE
You can also use ACCEPT, INPUT, and WAIT to display a question and receive the user's response in a single step. However, these commands have limited screen positioning, color, and data validation capabilities. For this reason, programmers generally prefer to use the more powerful @...SAY...GET, ?, and ?? commands. See Language Reference *for more information on ACCEPT, INPUT, and WAIT.*

Getting Data from an Array

Arrays are useful for storing information in memory. Your program can COPY data entered by users to a memory variable array, and later read in the data from the array using APPEND FROM ARRAY or REPLACE FROM ARRAY. (See Chapter 2 for more information on memory variable arrays.)

Formatting Data Entry Screens

Your application will look more polished if you format the data entry screens. This section discusses how to format the SAYs and GETs themselves. See the Formatting Data section for information on how to format the information that the user enters.

Setting Intensity and Delimiters

You can control the intensity of GETs, the characters used to delimit them, and the color of both SAYs and GETs.

With SET INTENSITY ON, SAYs display in standard mode and GETs in enhanced mode (inverse video). SET INTENSITY OFF uses standard mode for both. You can also use SET DELIMITERS to specify characters surrounding the GETs.

Control the colors of the SAYs and GETs with SET COLOR OF, SET COLOR TO, or the COLOR option of the @ command. Chapter 6 discusses color in more detail.

This example displays the SAYs in blue against a white background and the GETs in yellow against a red background, with the GETs enhanced and delimited by curly braces:

```
SET COLOR TO B/W,GR+/R
SET INTENSITY ON
SET DELIMITERS TO "{}"
SET DELIMITERS ON
```

Displaying a Memo Field in a Window

The WINDOW option of the @...SAY...GET command lets you display a memo field in a window. If you include an OPEN WINDOW clause, the contents of the memo field are visible. Without this clause, the window is closed and the field displays with the standard **memo** marker.

The following example, based on Orders.prg, defines a screen form with an open memo window:

```
USE Orders ORDER Order
DEFINE WINDOW memo_windo FROM 17,12 TO 22,75 PANEL
* <@...SAY...GETs outside the memo window>
@ 17, 4 SAY "NOTES: " GET Notes OPEN WINDOW memo_windo
* <rest of procedure>
RETURN
```

Customizing Browse Tables

The BROWSE command options provide flexibility in formatting a Browse table. Refer to *Language Reference* for a complete description. Table 8-1 gives a brief overview.

Table 8-1 BROWSE command options

Option	Purpose
COMPRESS	Compress table vertically to allow two more rows
FIELDS	Specify which fields display, and in what order
FORMAT	Activate current screen format file*
FREEZE	Confine the cursor to a specific column
LOCK	Keep left-hand columns on the screen when panning
NOAPPEND	Prevent addition of records
NOCLEAR	Keep table on screen when user exits Browse
NODELETE	Prevent deletion of records
NOEDIT	Prevent editing of records
NOFOLLOW	Prevent cursor from following edited key field
NOINIT	Re-use previously defined Browse table
NOMENU	Prevent access to the Browse menu bar
NOORGANIZE	Prevent access to the Organize menu
WIDTH	Specify width for columns
WINDOW	Display Browse table in a window

* Any attributes defined in the current format file, such as PICTURE and VALID, are effective in the Browse table as well as the form.

Figure 8-2 shows a Browse table in a window. Code for the Browse table follows the figure.

Figure 8-2 Sample Browse table

```
USE Vendors ORDER Vendor_id
DEFINE WINDOW browse_win FROM 1,5 TO 20,70
CLEAR
BROWSE FIELDS Vendor, Phone, Contact FORMAT NOAPPEND;
       NOEDIT NODELETE NOMENU COMPRESS WINDOW browse_win
```

Different command options make selected fields or an entire database file read-only. Use the BROWSE FIELDS /R option to make individual fields read-only. With both BROWSE and EDIT, the NOAPPEND, NOEDIT, NODELETE, and NOMENU options make the defined field list read-only. The easiest way to make an entire database file read-only is with USE...NOUPDATE.

Formatting Data

Data in its raw state is unformatted. There are no commas or dollar signs in numeric fields, and date fields all appear in a default format. The @...SAY...GET and ?/?? commands have special formatting options called PICTUREs and FUNCTIONs. These options, for example, enable you to:

- Enter formatting characters for the user, such as the parentheses and hyphen in American phone numbers
- Display numbers in the currency format of various countries
- Display dates in American or international format

- Force uppercase display
- Limit entry in a character field to numbers or letters

PICTURE templates format individual characters, while FUNCTIONs format the entire variable. To combine PICTUREs and FUNCTIONs, use the syntax *PICTURE "@<function> <template>"* as shown in Table 8-2.

Table 8-2 shows some useful PICTURE and FUNCTION clauses. These options are described in more detail in *Language Reference.*

Table 8-2 Commonly used PICTUREs and FUNCTIONs

Command	Example
@...SAY "Phone: " GET phone PICTURE "(999)999-9999"	(818)555-1234
@...SAY "SSN: " GET ssn PICTURE "999-99-9999"	441-42-4359
@...GET address PICTURE; "XXXXXXXXXXXXXXXXXXXXXXXXX"	1234 Birth St
@...GET mdate PICTURE "@D"	09/12/87*
@...SAY "Choose A-G or X to exit" GET choice PICTURE "!"	C
@...SAY "Is this correct? (Y/N)" GET answer PICTURE "Y"	Y
@...GET mcodename PICTURE "@R X X X X X X"	S H A D O W**
@...SAY "Enter last name:"; GET mlast PICTURE "@S7 !AAAAAAAAAAAAAAAAAAA"	Beethov***
@...SAY "Day of week" GET day; PICTURE "@M Mon,Tues,Wed,Thu,Fri"	Mon
? subtotal AT 10 PICTURE "9,999,999.99"	1,234,567.89
? mamount AT 10 FUNCTION "$"	$123,456.78
? mamount AT 10 FUNCTION "XC$"	$1,234.56 DB $3,456.78 CR
? mdate AT 10 FUNCTION "E"	23/09/87

* This format does not validate the user's entry. To do that, you must define the GET memory variable as a date type: mdate = {}.

** The R function tells dBASE IV to display the non-template characters (here, spaces), but not to store them in the GET variable.

*** The @S7 PICTURE function causes the entry to scroll in a 7-character window. Thus, the *en* of *Beethoven* is initially out of view.

In the following example, the ! FUNCTION displays the last name in uppercase letters, while the ! PICTURE template gives the first name an initial capital letter. The A PICTURE template allows the user to enter only letters in the *firstname* variable.

The *ssn* (social security number) PICTURE template enters hyphens for the user. The rate variable is formatted by a combined function and template. The $ PICTURE function produces currency format, while the template symbols limit the number of digits and enter the decimal point for the user.

Finally, *pay_hist* is an enumerated data option. The **Excelnt** choice appears by default.

```
@...SAY "Last name: " GET lastname FUNCTION "!"
@...SAY "First name: " GET firstname PICTURE "!AAAAAAAAAAAAA"
@...SAY "Social security no: " GET ssn PICTURE "999-99-9999"
@...SAY "Billing rate: " GET rate PICTURE "@$ 999.99"
@...SAY "Payment history: " GET pay_hist;
        PICTURE "@M Excelnt,Good,Fair,Poor";
        MESSAGE "Press SPACEBAR to scroll, ENTER to select"
```

TIP

You can STORE a PICTURE clause in a memory variable and then use the variable in an @...SAY command:

```
money = "@$ 999,999.99"
@ 10,10 SAY debit PICTURE money
@ 11,10 SAY credit PICTURE money
```

Using this method instead of macro substitution reduces your program's execution time.

Checking Data for Errors

Your program should evaluate data entered by the user before writing it to the database file. This helps to ensure the integrity of the database. Using data validation routines, your program can check data:

- Against a range
- Against conditions
- Against a list
- Against a database file
- For duplication

Checking Against a Range

The @...SAY...GET RANGE option sets lower and upper boundaries on numbers and dates entered by users.

```
date_var = {}       && Initializes a null date variable
date_low = {04/01/89}
date_high = {04/30/89}
@...GET date_var RANGE date_low,date_high
```

You can restrict the lower or the upper boundary individually by omitting one of the parameters. (Don't omit the comma.) If the user's entry is out of range, dBASE IV shows the proper range on the status line and instructs the user to press **Spacebar** to continue. The REQUIRED keyword applies to both RANGE and VALID. If you specify REQUIRED, dBASE IV checks RANGE and VALID each time you enter a record. If you do not specify REQUIRED, dBASE IV checks the RANGE and VALID parameters only when you change a record.

Checking Against Conditions

Two options of the @...SAY...GET command, VALID and WHEN, let you specify conditions for data entry.

Using @...SAY...GET VALID

The VALID option of the @...SAY...GET command lets you reject entries that don't meet the conditions you specify. You will find many uses for this post-processing capability. For example, you can use it to check for forbidden characters.

Suppose you want to obtain a Yes/No/Quit response to the prompt **Begin printing?**. The VALID condition and the ! PICTURE function in the following module ensure that the character entered into *answer* is Y, N, or Q. The program rejects any other character and displays the ERROR message.

```
answer = " "
@ 10,10 SAY "Begin printing? (Y/N/Q)" GET answer PICTURE "!";
        VALID answer $ "YNQ";
        ERROR "Enter Y or N, or Q to quit"
READ
```

The previous example identified the *allowed* characters. If you want to allow many characters and exclude only a few, it is sometimes more efficient to identify the *forbidden* characters. For example, suppose part numbers beginning with the letters X, Y, or Z have been discontinued. The following code prevents users from entering one of these part numbers:

```
mpart_id = SPACE(10)
discont = "XYZ"
CLEAR
@ 10,10 SAY "Enter part number: " GET mpart_id PICTURE "!-###-####";
        VALID .NOT. LEFT(mpart_id,1) $ discont;
        ERROR "Discontinued part number"
READ
```

To exclude spaces in the part numbers, you would modify the VALID clause to read *VALID .NOT." " $ mpart_id.*

In the next routine, the VALID expression limits the user's entry only if the quantity ordered is greater than zero and less than or equal to the quantity on hand. Otherwise, an error message appears.

```
USE Orders ORDER Part_id
USE Goods ORDER Part_id IN 2
mpart_id = SPACE(4)
mquantity = 0
@ 5,10 SAY "Enter part number: " GET mpart_id PICTURE "!999"
@ 6,10 SAY "Enter order quantity: " GET mquantity FUNCTION "9";
           VALID mquantity > 0 .AND. mquantity <= Goods->Qty_onhand;
           ERROR "Insufficient stock on hand"
READ
```

Examples in the Checking Against a List, Checking Against a Database File, and Checking for Duplication sections show the use of VALID with a user-defined function.

Using @...SAY...GET WHEN

The more global WHEN option controls whether users may access a field at all. The following routine forces users to fill in a part number before entering the part name. The WHEN condition evaluates the contents of the Part_id field to ensure that it is not empty. If it is, the cursor skips over the **Part Name**: option when the user tries to edit it

```
@ 10,10 SAY "Enter part number: " GET Part_id
@ 12,10 SAY "Part name: " GET Part_name WHEN "" <> TRIM(Part_id)
READ
```

Checking Against a List

Your program can check user entries against an approved list. In the following routine, the main program defines two lists, one for items slightly marked down and another for items being sold at cost. Then it prompts the user for a part number.

The VALID clause activates the Lookupid() user-defined function (see Chapter 4). This UDF identifies the first character of the user's entry, and then checks the appropriate list or file. If it doesn't find the part number, it runs the Not_found procedure.

Not_found defines a message that the part number is not in the required list. It obtains the name of the list as a parameter passed from the CASE statements in the Lookupid() UDF. Then it displays this message in one window and the appropriate parts list in another. Assume that the windows have already been defined.

```
markdown = "M100, M200, M250, M350, M425, M500, M600, M750, M900"
at_cost = "C100, C155, C175, C195, C200, C205, C220, C260, C300, C355"
mpart_id = SPACE(4)
USE Goods ORDER Part_id
@ 5,10 SAY "Enter part number: " GET mpart_id PICTURE "!999";
          VALID Lookupid(mpart_id);
          MESSAGE "Enter part number, for example A250";
          ERROR "Incorrect part number. Press spacebar and revise."
READ

FUNCTION Lookupid
   PARAMETERS entry
   is_here = .T.
   DO CASE
      CASE entry = "M" .AND. .NOT. entry $ markdown
         DO Not_found WITH "markdown"
      CASE entry = "C" .AND. .NOT. entry $ at_cost
         DO Not_found WITH "at_cost"
   ENDCASE
RETURN(is_here)

PROCEDURE Not_found
   PARAMETERS this_list
   message = "Part number not in " + this_list + " list"
   is_here = .F.
   answer = ""
   ACTIVATE WINDOW alert_win
   @ 1,0 SAY message
   WAIT "Press S to stop or spacebar to see parts list";
        TO answer
   DEACTIVATE WINDOW alert_win
   ACTIVATE WINDOW look_win
   SCAN ALL WHILE UPPER(answer) <> "S"
      @ 1,1 SAY "Part No.     Part Name"    && Assumes HEADING OFF
      LIST Part_id + SPACE(3) + Part_name NEXT 10
      WAIT "Press spacebar to continue or S to stop" TO answer
      CLEAR
   ENDSCAN
   DEACTIVATE WINDOW look_win
RETURN
```

Checking Against a Database File

You can check user entries against a database file. The routine below checks whether the part number entered by the user is in the Goods database file. A more generalized version of this example is in Orders.prg and the Lookupid() user-defined function in Library.prg. If the SEEK() for the part number fails, the Lookupp() UDF opens a window and SCANs the database file for items having the same first three characters in the part number. It LISTs the part number, part name, and description for these items five at a time, until the user presses **R** to return to the **Option** menu.

```
USE Orders ORDER Order
USE Goods ORDER Part_id IN 2
DEFINE WINDOW look FROM 6,5 TO 16,65 PANEL COLOR &c_list.
mpart_id = SPACE(4)
@ 10,4 SAY "Part #: " GET mpart_id FUNCTION "!";
            VALID Lookupp(mpart_id) ERROR;
              "Invalid part # - please revise"
READ

FUNCTION Lookupp
   PARAMETERS partid
   IF .NOT. SEEK(partid,"Goods")
      ACTIVATE WINDOW look
      DO Warnbell
      answer = " "
      SELECT Goods
      SET HEADING OFF
      SCAN FOR LIKE(SUBSTR(partid,1,3)+"*",Part_id) WHILE .NOT. answer $ "rR"
         @ 0,0 SAY "-------------- INVALID PART NUMBER ------------"
         @ 1,0 SAY "    This is a list of similar part numbers"
         LIST OFF Part_id,SUBSTR(Part_name,1,20),SUBSTR(Descript,1,22);
               FOR LIKE(SUBSTR(partid,1,3)+"*",Part_id) NEXT 5
         WAIT "Press spacebar to continue or R to return to OPTION MENU" TO answer
         CLEAR
      ENDSCAN
      SET HEADING ON
      DEACTIVATE WINDOW look
      SELECT 1
   ENDIF
RETURN not_valid = .NOT. FOUND()
```

Checking for Duplication

Your program may need to prevent users from entering duplicate records into a database file. One way to check for duplicates is demonstrated by the following procedure, New_emp. The DO WHILE loop repeats until the user either enters nothing, or enters an employee ID number that is not already on file. If nothing is entered, the procedure RETURNs to its caller. If a new, unique ID number is entered, a blank record is added to the file, the Emp_id field is REPLACEd with the new ID number, and the record is EDITed.

```
PROCEDURE New_emp
USE Employee ORDER Emp_id
CLEAR
DO WHILE .T.
   memp_id = SPACE(11)
   @ 8, 10 SAY "Employee ID Number for the new employee";
           GET memp_id PICTURE "999-99-9999"
   READ
   IF "" = TRIM(memp_id)
      RETURN
   ENDIF
   IF SEEK(memp_id)
      ?? CHR(7)
      @ 9, 10 SAY "Employee " + memp_id + " is already on file."
   ELSE
      EXIT
   ENDIF
ENDDO
APPEND BLANK
REPLACE Emp_id with memp_id
EDIT NEXT 1
RETURN
```

The New_emp procedure requires that the user supply a unique ID number in order to add an employee to the file. This guarantees that the record will be unique when it is added to the file. However, New_emp does not prevent users from introducing duplicate records by changing the Emp_id field during the EDIT NEXT 1 command. This could be corrected by activating a format file that does not allow editing the Emp_id field, or by using @...SAY...GET commands to collect the remaining fields.

The next example, from Goods.prg and Library.prg, checks for duplicates when records are added or edited. It uses the VALID clause of the @...GET command with a user-defined function which causes the VALID to fail if the user enters an existing part number.

The first IF statement in the UDF prevents an unnecessary search for duplicates if the database file or the key field of the new or edited record is empty. Then the UDF saves the current record position for later use.

Next, the UDF SEEKs the new or edited key expression, which has been stored in a variable and passed to the parameter *key*. The first CASE statement sets *rec_is_dup* to .T. if a record other than the current one has the same key expression. This would be the case if the user entered a duplicate key while editing a record. The second CASE statement sets *rec_is_dup* to .T. if any record has the same key. This would be the case if the user entered a duplicate key while adding data.

If the record has a duplicate key, the UDF shows the record in a window and prompts the user to enter a unique key expression. Then it restores the record pointer to the original record position and terminates. The UDF returns a false (.F.) to the VALID clause if the record contains a duplicate key expression, causing the VALID to fail.

```
USE Goods ORDER Part_id
mpart_id = SPACE(4)
@ 6,4 SAY "Part #: " GET mpart_id FUNCTION "!" VALID Duplicat(mpart_id);
      ERROR "Duplicate part number, please re-enter";
      MESSAGE "Enter a part number, or Esc to quit"

FUNCTION Duplicat
   PARAMETERS key
   rec_is_dup = .F.
   IF RECCOUNT() = 0 .OR. "" = TRIM(key)
      RETURN rec_is_dup
   ENDIF
   record_num = RECNO()
   SEEK TRIM(key)
   DO CASE
      CASE PROMPT() = " Edit record"
         rec_is_dup = record_num <> RECNO() .AND. FOUND()
      CASE PROMPT() = " Add record"
         rec_is_dup = FOUND()
   ENDCASE
   IF rec_is_dup
      ACTIVATE WINDOW duplicat
      CLEAR
      DO Warnbell
      ?  "------------------ DUPLICATE RECORD -----------------"
      ?  "                     Duplicates not allowed"
      ?  " " + part_id + " " + part_name
      ? "This is the EXISTING record in the database; re-enter part number"
      WAIT "     Press spacebar to continue..."
      DEACTIVATE WINDOW duplicat
   ENDIF
   GO record_num
RETURN .NOT. rec_is_dup
```

Processing Keypresses

The ON KEY and ON ESCAPE commands, and the INKEY(), LASTKEY(), and READKEY() functions, allow your application to do selective processing depending on the keys a user presses.

Using ON KEY and ON ESCAPE

ON KEY executes a command when the user presses a key. Use the LABEL keyword to specify a particular keypress; otherwise, any key will be trapped. ON ESCAPE traps the **Esc** key only. ON KEY causes an immediate trigger in EDIT, BROWSE, READ, and user-defined menus. Otherwise the trap occurs after the current command executes.

The following command lets users access a help procedure by pressing **F1**. The LABEL keyword is required in this case.

```
ON KEY LABEL F1 DO Helper
```

Using INKEY(), LASTKEY(), and READKEY()

Table 8-3 summarizes the differences among INKEY(), LASTKEY(), and READKEY(). The entries for INKEY() and READKEY() in Chapter 4 of *Language Reference* contain tables of the codes returned for each key.

Table 8-3 Differences among INKEY(), LASTKEY(), and READKEY()

Function	Returns	Use
INKEY()	ASCII code of current keypress (first key in typeahead buffer)	With menus and prompts or to pause program
LASTKEY()	ASCII code of key used to exit full-screen command	With full-screen commands or @...GETs
READKEY()	dBASE code of key pressed to exit full-screen command	With full-screen commands or @...GETs

INKEY(0) pauses the program until the user presses a key. INKEY(n) pauses the program the designated number of seconds. READKEY() returns one of two possible codes: a number from 0 to 36 if no data was changed prior to exit, and that number increased by 256 if data was changed.

Notice the relationship between READKEY() and LASTKEY(). READKEY() determines whether the user saved data or discarded changes when exiting a full-screen command. LASTKEY() then determines precisely what key the user pressed to exit.

The first routine below lets the user press **F1 Help** to access a help screen or ↵ to edit data. Assume the existence of a help procedure called Helper. INKEY(0) pauses the program until the user presses a key.

```
CLEAR
@ 10,10 SAY "Press F1 HELP for help or RETURN to enter data"
keypress = INKEY(0)
DO CASE
   CASE keypress = 28
      DO Helper
   CASE keypress = 13
      * <data entry commands>
ENDCASE
```

The next routine uses a Load_fld procedure to copy the data from record 5 into memory variables. Another procedure, Edit_rec, lets the user edit the data. If the user presses **Ctrl-End** (READKEY() = 270) or ↵ (READKEY() = 271), Sav_data REPLACEs the data in the database file.

```
USE Goods
GO 5
DO Load_fld
DO Edit_rec
IF READKEY() > 255
   DO Sav_data
ENDIF
```

Providing Help for the User

User help in a dBASE IV application can vary from simple to sophisticated. Options of the @...SAY...GET command display one-line help prompts. You can also create a help text file that users can call from the program. Finally, you can design a context-sensitive help system that provides help relevant to the tasks users are doing.

Providing Help with @...SAY...GET

Options of the @...SAY...GET command can accomplish many simple user help functions in your program. The PICTURE and FUNCTION options force user entries into a required format (see the Formatting Data section earlier in this chapter). The RANGE option automatically displays an error message if the user's entry is out of range. The ERROR clause of the VALID option displays a message of up to 80 characters if the user's entry fails the VALID (see the Using @...SAY...GET VALID section earlier in this chapter).

Providing Help Text

To provide extensive help text, create a help program file using the TEXT...ENDTEXT construct. If the help file is used by only one program, include it as a procedure within that program. However, if several programs use the same help text, create a separate Help.prg file. Have each program DO this external help file as needed. This approach saves many lines of code by not repeating the help procedure throughout your application.

The basic format for a help file is:

```
CLEAR
@ 0,0
TEXT
   . && Help text. Format the text exactly as you want it
   . && to appear on the screen: you must supply
   . && indentation, underlining, and so on.
ENDTEXT
WAIT "End of help file. Press any key to continue."
CLEAR
RETURN
```

Helper.prg is a typical help program (see the sample code) that provides help for all of the menus in the Business application.

Providing Context-Sensitive Help

The core of a dBASE IV context-sensitive help routine is the VARREAD() function. VARREAD() returns the name in uppercase letters of the field or memory variable currently being altered. Use VARREAD() with ON KEY to provide help for all GETs in a form, as shown in the following example using the Cust database file.

```
ON KEY LABEL F1 DO Get_help
CLEAR
@ 10,10 SAY "Contact: " GET Contact
@ 12,10 SAY "Category: " GET Category
@ 14,10 SAY "State: " GET State
READ
ON KEY LABEL F1

PROCEDURE Get_help
   null = INKEY()
   DEFINE WINDOW get_win FROM 16,10 TO 20,60
   ACTIVATE WINDOW get_win
   DO CASE
      CASE VARREAD() = "CATEGORY"
      TEXT
          The category can be one of the following:
          ARCHITECT, CONSULTANT, CONTRACTOR, LEGAL.
          Enter in uppercase letters, spelled as shown.
      ENDTEXT
      OTHERWISE
         ?? "Sorry, there is no additional information"
         ?  "about ", VARREAD()
   ENDCASE
   WAIT "Press any key to exit Help" TO null
   DEACTIVATE WINDOW get_win
   RELEASE WINDOW get_win
RETURN
```

Entering Data into the Database

You use the same commands for data entry in your programs as you use at the dot prompt: APPEND, EDIT, CHANGE, READ, and REPLACE. This section shows two basic approaches for entering data into the database.

Using APPEND, EDIT, and CHANGE

APPEND, EDIT, and CHANGE display the database in form view. A single record appears according to a designated screen format file. To use these commands in a program to enter data, just create the screen format file, activate it with SET FORMAT, and then issue the desired editing command.

To let the user *add* one record at a time, display a data entry form to capture and validate the data (see the Getting Data with Screen Forms and Checking Data for Errors sections earlier in this chapter). Then APPEND BLANK to display a blank record, and have the user EDIT that record.

```
SET FORMAT TO Invoice
APPEND BLANK
EDIT NEXT 1
```

Here's how to permit *editing* of a single record. (Assume that the user enters a customer ID at the prompt.)

```
mcust_id = SPACE(6)
DO WHILE "" <> TRIM(mcustid)
   @ 10,10 SAY "Enter customer ID: " GET mcust_id
   IF SEEK(mcust_id)
      SET FORMAT TO <.fmt filename>
      EDIT NEXT 1
      SET FORMAT TO
   ELSE
      ? "Customer not found"
   ENDIF
ENDDO
```

Use a FILTER condition to let users EDIT or CHANGE certain records without granting access to other records. The routine below lets the user edit records for a selected vendor. First, the @...SAY...GET and READ commands capture the user's entry of a vendor number. If the user presses **Esc** (ASCII code 27), the routine EXITs at this point. Otherwise, the routine FILTERs the Goods database file by the specified vendor number. Finally, if any records match the filter condition, the EDIT command shows the first matching record.

```
USE Goods ORDER Vendor_id
DO WHILE .T.
   mvendor_id = SPACE(4)
   CLEAR
   @ 10,10 SAY "Enter vendor number or press Esc to exit";
           GET mvendor_id PICTURE "!999"
   READ
   IF LASTKEY() = 27
      EXIT
   ENDIF
   SET FILTER TO Vendor_id = mvendor_id
   GO TOP
   IF .NOT. EOF()
      EDIT
   ELSE
      ?? CHR(7)
      @ 15,30 SAY "Vendor ID not found. Please try again."
   ENDIF
ENDDO
```

In both these examples, you can use CHANGE in place of EDIT. EDIT and CHANGE have the same options as BROWSE (see Table 8-1).

Variation #1 — Carrying Data Forward.

The SET CARRY command lets you carry forward the contents of one or more fields in a database file when APPENDing data. SET CARRY affects screen format files and the full-screen commands APPEND, BROWSE, EDIT, and CHANGE. Fields are not carried forward with the APPEND BLANK command.

You can save the user unnecessary keystrokes by carrying forward specific fields. In the following example, the State field will be carried forward until the program SETs CARRY OFF.

```
USE Cust ORDER Cust_id
SET CARRY TO State
SET FORMAT TO <.fmt filename>
APPEND
```

Issuing SET CARRY TO automatically SETs CARRY ON.

Variation #2 — Entering Default Data

With the DEFAULT option of the @...SAY...GET command, you can save the user keystrokes by entering default data into the database file during an APPEND. A typical use for this option is to enter today's date in a date field:

```
@ 10,10 SAY "Enter transaction date: " GET Date_trans DEFAULT DATE()
```

You might also use this option to enter a default state (DEFAULT "CA") or order quantity (DEFAULT 1).

The DEFAULT option works in APPEND, INSERT, BROWSE, and EDIT with database fields only, and only when APPENDing records. If your program carries forward a field that has a DEFAULT, the carry-forward value takes precedence over the DEFAULT whenever SET CARRY is ON.

Using READ and REPLACE

For maximum database integrity, activate the GETs using the READ command (rather than APPEND or EDIT) and REPLACE the data into the database file.

This method stores the information that the user enters in memory variables. After verifying that the data is correct, the program REPLACEs the old data with the new or revised data. The user never touches the data in the database file.

NOTE
You should use the REPLACE method if you intend to run your application on a network. Working with memory variables rather than the records themselves minimizes the amount of time a record is inaccessible due to an RLOCK().

The Add_new procedure in Library.prg handles the addition of new records for all of the sub-applications. First it calls Init_fld, which initializes the memory variables that hold data entered by the user. Then it calls Get_data, which displays the fields on the screen form for the sub-application. The READ command captures the data entered by the user.

If the user saves no data or tries to enter data with a duplicate key field, the procedure terminates. (The Checking for Duplication section earlier in this chapter shows how the program prevents duplicate data.)

If the data is valid, the procedure calls Sav_data. This procedure opens a window and asks whether to save the data to disk. If the user answers **Y**, the procedure first APPENDs a BLANK record if the user is adding records. Then it calls Repl_fld, which REPLACEs the fields in the database file with the contents of the corresponding memory variables. Each sub-application has its own Rep_fld procedure; the one from Goods.prg is shown here.

```
PROCEDURE Add_new
   DO Init_fld
   DO Get_data
   READ
   IF "" = TRIM(&key.) .OR. READKEY() < 256
      RETURN
   ELSE
      IF rec_is_dup
         rec_is_dup = .F.
         RETURN
      ELSE
         DO Sav_data
         GO record_num
      ENDIF
   ENDIF
RETURN

PROCEDURE Sav_data
   choice = "Y"
   ACTIVATE WINDOW alert
   @ 0,1 SAY "———— SAVE DATA ————"
   @ 2,1 SAY "Save this data to disk? (Y/N)" GET choice PICTURE "Y"
   READ
   DEACTIVATE WINDOW alert
   IF choice = "Y"
      IF PROMPT() = "Add record"
         APPEND BLANK
         record_num = RECNO()
      ELSE
        GO record_num
      ENDIF
      DO Repl_fld
   ELSE
      GO record_num
   ENDIF
RETURN

PROCEDURE Repl_fld
   REPLACE Part_id WITH m->part_id, Part_name WITH m->part_name,;
           Descript WITH m->descript, Qty_onhand WITH m->qty_onhand,;
           Cost WITH m->cost, Price WITH m->price,;
           Qty_2order WITH m->qty_2order, Vendor_id WITH m->vendor_id,;
           Lead_time WITH m->lead_time, Comments WITH m->comments
RETURN
```

Use a similar approach to edit existing records (see the Edit procedure in Library.prg in the sample code).

TIP

Placing the READ at the end of a group of @...SAY...GETs or after the SET FORMAT TO command causes all the @...SAY...GETs to display at once. The user can go back with the ↑ key to correct errors. However, you can also display @...SAY...GETs one at a time by putting a READ after each prompt.

Deleting Data from the Database

Deleting records works the same in dBASE programming as it does at the dot prompt. Position the record pointer to the desired record number and then DELETE the record.

The Eraser procedure in Library.prg processes deleted records for all the sub-applications. First it opens a window and asks whether the record should be erased. If the user responds **Y**, the procedure marks the record for deletion, SKIPs the record pointer to the next record, and sets the variable *erased* to true.

For improved program efficiency, the PACK command is located in the Sub_ret procedure in Library.prg. That way, the program only PACKs the database file when the user exits the sub-application with the option **Quit to Main Menu**.

```
PROCEDURE Eraser
   answer = " "
   ACTIVATE WINDOW alert
   @ 0,0 SAY "——— DELETE DATA RECORD ———"
   @ 2,1 SAY "Erase this data record? (Y/N)" GET answer PICTURE "Y"
   READ
   DEACTIVATE WINDOW alert
   IF answer = "Y"
      DELETE
      SKIP
      .
      .
      .
      erased = .T.
   ENDIF
RETURN
```

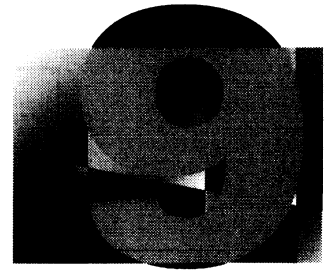
Variations — DELETE ALL, RECALL, and SET DELETED ON

You can DELETE ALL records matching a specified condition, such as all customers from a particular state. To do this, modify the previous example to GET the user's entry of a state and then *DELETE ALL FOR State = mstate.*

You can RECALL a DELETEd record as long as you haven't PACKed the database file. Don't forget to issue the PACK command to save disk space, once you're sure you want to permanently erase DELETEd records.

If you want commands such as LIST and LOCATE to skip DELETEd records, SET DELETED ON. RECALL ALL has no effect with SET DELETED ON.

Processing: Ordering the Database File



Users enter data into a database file in *natural* order — the order in which they receive it. However, they usually want to work with the data in alphabetical, numeric, or date order. dBASE IV indexing commands and functions control the order in which data appears.

What This Chapter Covers

This chapter tells you how to control the order of data in a programming environment. The topics covered are:

- About index files
- Creating and modifying index files
- Using index files
- Determining the current index status
- Estimating the index file size

About Index Files

When you INDEX a database file, dBASE IV creates an index file containing the contents of the key expression plus a pointer to the corresponding record in the database file (see Figure 9-1).

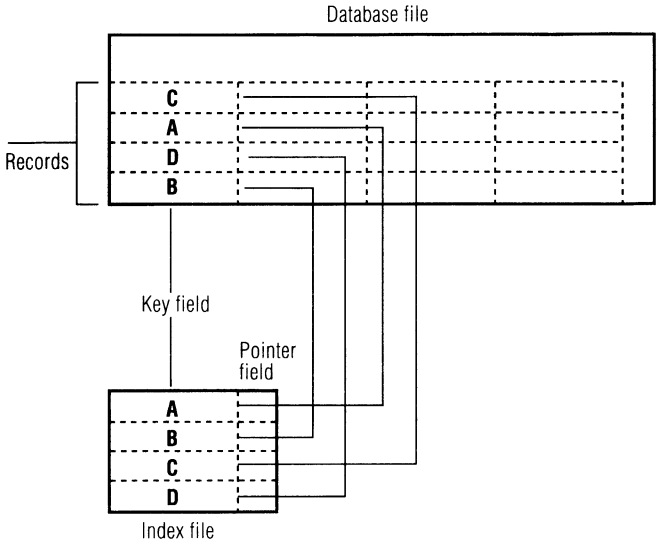


Figure 9-1 Indexing a database file

Index files help you produce data output in the desired order, conduct more efficient searches, and relate database files.

dBASE IV can create two types of index files. *Index (.ndx)* files are for occasional use (for instance, for a particular report). *Multiple index (.mdx)* files are for routine or production use. Each multiple index file can include up to 47 index *tags*, each representing a separate index order.

A special multiple index file or *production index* is linked to the database file through the file header and has the same root name as the database file. It opens automatically when you USE the database file. This means the index tags it contains are automatically kept current. In addition to the production index, you can open up to 10 .mdx and .ndx files at once.

Creating and Modifying Index Files

Use the INDEX command to create index files:

```
USE  Vendors
INDEX  ON  Vendor_id  TAG  Vendor_id
```

Tag names follow the same naming conventions as memory variable names (see Chapter 2). dBASE IV automatically creates a production index if none exists, unless you specify another .mdx file with the clause *OF <.mdx filename>*.

You can index a database file on a character, numeric, or date field or a combination of fields. The following command indexes the Cust database file on the Cust_id field and creates a Cust_id tag in the production index file:

```
USE Cust
INDEX ON Cust_id TAG Cust_id
```

Similarly, you can index the Acct_rec database file on the numeric field Oldbalance:

```
USE Acct_rec
INDEX ON Oldbalance TAG Oldbalance
```

You can index a database file on more than one field:

```
USE Employee
INDEX ON Lastname + Firstname TAG Names
```

Where fields are of different types, you must convert them to a single type. For instance, the STR() function in the next example converts the numeric Salary field to character data for concatenation with Department:

```
USE Employee
INDEX ON Department + STR(Salary,8,2) TAG Dept
```

Similarly, you can index on character and date fields as follows:

```
USE Orders
INDEX ON Cust_id + DTOS(Date_trans) + PO_number TAG Order
```

Controlling Duplicate Keys

If the database file contains more than one record with the same key value, INDEXing the file with the UNIQUE option puts only the first record with a duplicate key in the index file. Commands that access records, such as LIST, won't find additional records with the same key. UNIQUE indexes are flagged in the output of DISPLAY/LIST STATUS. See *Language Reference* for examples using UNIQUE.

While the UNIQUE option *conceals* duplicate records, it does not prevent users from *entering* them. Any duplicate records added by the user will seem to disappear after they are entered. Proper data validation completely prevents entry of duplicate data (see the Checking for Duplication section in Chapter 8).

Conditional Indexing

If you regularly work with a subset of the records in a database file, you can create a *conditional index* by including a FOR clause in the INDEX command. A conditional index contains entries only for the records that satisfy the FOR clause condition.

For example, in an accounts payable database file you may be concerned with unpaid invoices most of the time. Add a FOR clause to the INDEX command to select only unpaid invoices:

```
INDEX ON Inv_no TAG Unpaid FOR .NOT. Paid
```

When you access the database file through the index, paid invoices are suppressed. Although you can accomplish the same thing with *SET FILTER TO .NOT. Paid,* the conditional index can be much faster.

Indexing in Descending Order

The DESCENDING option of the INDEX command indexes an expression in reverse order. It is available with .mdx tags but not .ndx files. This option affects commands that move the record pointer, such as BROWSE, EDIT, LIST, and SKIP.

The DESCENDING option affects all the fields in an index key expression. This example indexes the Orders database file in reverse order by transaction date and client identification number:

```
USE Orders
INDEX ON DTOS(Date_trans) + Cust_id TAG Down DESCENDING
```

You can also index a numeric or date field in descending order with an ascending character field. The command below subtracts the value in the Number field from 10,000, a number larger than the user can enter into that field. Because this difference decreases as the value of Number increases, INDEXing on it arranges the field in descending order. To combine the descending numeric index with an ascending character index, the routine concatenates the STR() of the difference to Char_fld.

```
INDEX ON STR(10000 - Number,4,0) + Char_fld TAG <tag name>
```

Follow the same principle to index a date field in descending order with an ascending character field:

```
INDEX ON STR({12/31/2000} - Date_fld,8) + Char_fld TAG <tag name>
```

Modifying Index Files

You can modify multiple index files by adding or deleting individual index tags. Adding an index tag is similar to creating the multiple index file. This example adds a Zip_code tag to the Vendors production index:

```
USE  Vendors
INDEX  ON  Zip  TAG  Zip_code
```

Use DELETE TAG to delete index tags from multiple index files. Name the tag you want to delete and, if it's not contained in the default production index, name the multiple index file. If you delete all the tags from the production index, dBASE IV deletes the production index file and updates the database file header.

Using Index Files

This section explains how to use index files in your application program — how to open them, close them, and control the data display order.

Opening Index Files

Most of your indexes will be tags in the production .mdx file. To open them, just USE the file in the desired ORDER. The database file header identifies the production index, which opens automatically.

```
USE  <file>  ORDER  <tag>
```

To open an .mdx file other than the production index, provide the name of the .mdx file in an OF clause. The following example opens the Vendors file with its production index in work area 1. It then opens the Goods database file with a production and an alternate multiple index file in the next available work area with the SELECT() function. It ORDERs the Goods display by the part number tag in the second multiple index file.

```
USE  Vendors
USE  Goods  IN  SELECT()  INDEX  Mult2  ORDER  Part_id  OF  Mult2
```

Closing Index Files

Several commands handle the different situations in which you might want to close an index file. Consult *Language Reference* for more information on the individual commands named here.

- DELETE TAG closes one or more index (.ndx) files without closing all of them. (While DELETE TAG only *closes* an .ndx file, it permanently deletes a multiple index tag.)
- SET INDEX TO with a file list closes all index files except the production index, and then opens the index files named in the list.
- SET INDEX TO with no argument or CLOSE INDEX closes all open index files except the production index.
- To close the production index, CLOSE the database file.

Controlling the Display Order

The *master* index file controls the display and search order. You can specify the master index in several ways:

- USE <filename> ORDER <.mdx tag> makes the tag the master index.
- USE <filename> INDEX <.ndx file list> makes the first-named .ndx file in the file list the master index.
- For a database file already in USE, SET ORDER TO <expN> or <.mdx tag> makes the specified file or tag the master index.
- For a database file already in USE, SET INDEX TO <file list> ORDER <.ndx file> or <.mdx tag> changes the active index file and makes the specified file or tag the master index.
- SET ORDER TO or SET INDEX TO with no argument displays the data in natural (record number) order.

Determining the Current Index Status

The DISPLAY STATUS command and several functions reveal the current index status. Table 9-1 summarizes these briefly. See *Language Reference* for more detailed information.

Table 9-1 Determining index status

Command/Function	Returns
DISPLAY STATUS	Lists active index files
ALIAS()	Alias for current work area
DBF()	.dbf file in the current work area
DESCENDING()	Returns a true (.T.) if controlling index is descending; returns a false (.F.) if an .ndx is the controlling index
FOR()	For condition of controlling index, or a null string if there is no open index
KEY()	Current controlling key
MDX()	Name of controlling .mdx file, or a null string if controlling index is .ndx
NDX()	Name of controlling .ndx file, or a null string if controlling index is .mdx
ORDER()	Name of controlling index in the current work area
TAG()	Name of controlling index
TAGCOUNT()	Total number of indexes in the current work area
TAGNO()	Position of controlling index
UNIQUE()	Returns a true (.T.) if UNIQUE is used with controlling index

The index functions have been enhanced to return the names of the controlling index files when they have no argument specified. If there is no controlling index, these functions will return a null string, a zero, or a logical false (.F.) depending on the data type they reference.

The ALIAS() function will return the alias of the work area in which a file is opened. This defaults to the database filename open in that work area. If there is no open filename, then ALIAS() returns the dBASE IV assigned name, which consists of an underscore character and the work area number (for example, _18). This simplifies manipulation of the up to 40 concurrently open work areas.

The following routine saves the current database filename and index order before it closes all database and index files. Later, it reopens the database file and re-establishes the index ORDER.

```
lc_dbf = DBF()
lc_order = ORDER()
CLOSE databases
.
. <additional commands>
.
USE (lc_dbf)
IF "" <>lc_order
    SET ORDER TO (lc_order)
ENDIF


mfilename = DBF()
m_order = ORDER()
CLOSE DATABASES
.
. <additional commands>
.
USE (mfilename)
IF "" <> m_order
    SET ORDER TO (m_order)
ENDIF
```

Estimating the Index File Size

This section shows how to estimate the size of an .ndx file. (Use a routine like the Dbt_size user-defined function in Chapter 14 to compute the size of an .mdx file.)

The following routine stores the size of the index file, returned from the Ndxsize procedure, to a variable called *indx_size*. It then subtracts the index size from DISKSPACE() to determine whether there is enough space on the disk for the index file.

The Ndxsize procedure estimates the size of an .ndx file to within an accuracy of 1K. It assumes the database file has just been INDEXed or REINDEXed, and that you are not creating a UNIQUE or conditional index.

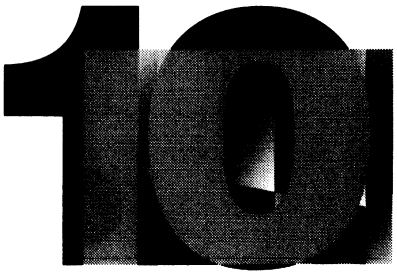
The memory variables are defined as follows:

- *keylength* — Number of characters in the index key expression. This is supplied as *klength* from the main routine. If you'd rather supply the value yourself, enter 8 for numeric or date type keys, and the closest multiple of 4 (rounded up) for character keys.

- *keysperblk* — Number of index keys that fit in one 512K block. The record pointer for each key uses 8 bytes.
- *data_bloks* — Number of blocks needed for all keys.
- *totalbloks* — Total number of blocks needed for data plus the root node of the index (2 blocks).

```
USE Orders
STORE 0 TO klength, indx_size
CLEAR
mfield = SPACE(20)
@ 10,10 SAY "On what field do you want to index? " GET mfield
READ
DO CASE
   CASE TYPE(mfield) = "C"
      klength = LEN(TRIM(&mfield.))
      klength = klength + MOD(klength,4)
   CASE TYPE(mfield) = "D"
      klength = 8
   CASE TYPE(mfield) = "N"
      klength = 8
   OTHERWISE
      ? "Data type error"
      RETURN
ENDCASE
DO Ndxsize WITH klength, indx_size
IF DISKSPACE() - indx_size <= 0
   ?? CHR(7)
   @ 20,10 SAY "Not enough space on disk for index file"
ENDIF

PROCEDURE Ndxsize
   PARAMETERS keylength, ndx_size
   keysperblk = INT(504 / (keylength + 8))
   data_bloks = RECCOUNT() / keysperblk
   totalbloks = data_bloks + 2
   keysperblk = keysperblk + 1
   DO WHILE data_bloks > keysperblk
      data_bloks = INT((data_bloks - 1) / keysperblk) + 1
      totalbloks = totalbloks + data_bloks
   ENDDO
   ndx_size = totalbloks * 512
RETURN
```

Processing: Searching for Data



This chapter covers the programming aspects of searching for data in a database file. You use the same commands to search in a program as you do to search at the dot prompt. However, programs give you more control over when the search starts and stops.

What This Chapter Covers

This chapter covers the following topics:

- The two search methods
- Searching for single records
- Searching for multiple records
- Querying database files
- Determining item existence
- Search tips

The Two Search Methods

dBASE IV uses two search methods. Both methods follow the index order, but the first method searches node by node, while the second method uses a special search algorithm to quickly reach the specified record (see Figure 10-1).

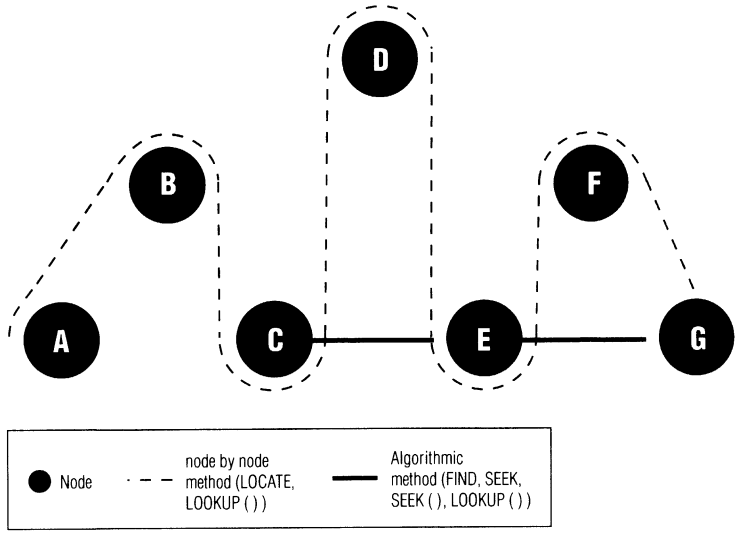


Figure 10-1 Two search methods

Some dBASE IV commands and functions use one method, some the other. LOCATE uses the first method, searching the database file. LOOKUP(), like LOCATE, searches the database file if no index tag is open. FIND, SEEK, and SEEK() use the second method, and thus require an index file. LOOKUP() uses the second method if a tag exists on the search field.

As you review the examples in this chapter, keep in mind that the bottom of the database file is the last logical record, whereas the end-of-file (where EOF() is .T.) is a phantom record just *after* the last logical record. The top of the file is the first logical record, while beginning-of-file (BOF()) is just *before* the first logical record. There is no phantom record at the beginning of the file.

Searching for Single Records

Often a user wants to find a single record to view or edit. Your program can search either an indexed or an unindexed field according to a search criterion entered by the user.

Searching an Indexed Field

The FIND and SEEK commands search indexed database files for records with the specified index key expression. FIND and SEEK search the index file, using a search algorithm to maximize the search speed. Once the key expression is found in the index file, a pointer indicates the corresponding record in the database file.

FIND and SEEK are similar but not identical. SEEK is the more powerful of the two and is used most frequently in programs. FIND is used primarily at the dot prompt, where it can save keystrokes. Table 10-1 summarizes major differences between the two commands.

Table 10-1 Comparison of FIND and SEEK

FIND	SEEK
Takes a literal value: FIND Smith	Takes an expression: SEEK mlastname + mfirstname
Handles character expressions only: FIND Dallas	Handles character, numeric, and date expressions: SEEK DATE() – Date_trans
May omit delimiters: FIND Los AngelesCA*	May not omit delimiters with character data: SEEK "Los AngelesCA"
Use & (macro) function in search with memory variables: FIND &mcity.	Use variable name in search with memory variables: SEEK mcity

*Example assumes that the database file is indexed on City – State.

The following routine shows how to SEEK a single record in a program. First it USEs the Employee database file ORDERed on the tag Names, which has the key expression *Lastname + Firstname*. Then it GETs the user's entry of a name. If the user presses ↵ with no entry (which enters a blank string), the routine terminates. If the user enters a name and a SEEK() on that name is successful, the routine prints the fields shown.

```
USE Employee ORDER Names
mlast = SPACE(15)
mfirst = SPACE(10)
@ 10,10 SAY "Enter last name: " GET mlast
@ 12,10 SAY "Enter first name: " GET mfirst
READ
IF "" <> TRIM(mlast+mfirst)
   IF SEEK(TRIM(mlast+mfirst))
      ? Lastname, Firstname, Department, Specialty
   ENDIF
ENDIF
```

The next example is based on the Orders database file. It stores the customer's identification code in the memory variable *code*. It then SEEKs records with that code number and LISTs customer order information.

```
USE Orders ORDER Cust_id
code = SPACE(6)
@ 10,10 SAY "Enter customer code " GET code PICTURE "!99999"
READ
IF SEEK(code)
    LIST Customer, Phone, Contact, Date_last WHILE Cust_id = code
ELSE
    ? "Customer ID not found"
ENDIF
```

NOTE
FIND and SEEK match expressions starting at the first character, and continue searching only until a match occurs. Thus, dBASE IV might find Smith when you're looking for Smithe. SET EXACT ON guarantees an exact match. Use SET NEAR to keep the record pointer from going to the end of the file if a search fails. See Chapter 3 of Language Reference *for more information on these commands.*

Using SET KEY on Indexed Database Files

The SET KEY command allows you to search many different sets of records while maintaining a small number of indexes. It establishes a range of values to search between. Each work area can be assigned a different SET KEY range.

This command works with the SET FILTER command to set up complex search criteria, or it can be used with complex indexes. See *Language Reference* for SET KEY syntax and examples.

Searching an Unindexed Field

Use LOCATE to search an unindexed field. A LOCATE is slower than a SEEK or FIND. To make LOCATE as fast as possible:

- Put large database files in natural rather than indexed order (SET ORDER TO) before doing a LOCATE. Restore the index order later with SET ORDER TO TAG.
- Use the <scope> clause, such as LOCATE NEXT 5 FOR Vendor_id = "1000", to start the search at the current record rather than the top of the file.

The following routine LOCATEs a particular vendor by name. Then it BROWSEs the data in Vendor_id order starting with that vendor.

```
USE Vendors ORDER Vendor_id
.
.
SET ORDER TO
LOCATE FOR Vendor = "Fastware Furniture Mfg."
SET ORDER TO TAG Vendor_id
BROWSE
```

Searching for Multiple Records

SCAN...ENDSCAN is a programming construct that processes all or selected records in a database file. It works with both indexed and unindexed files.

The following example displays records with a vendor code of 2100 in a previously defined window, *goods_win.* It displays ten records at a time as long as the user presses any key to continue, until the end of the file.

```
USE Goods ORDER Vendor_id
mvend_id = "2100"
SEEK mvend_id
ACTIVATE WINDOW goods_win
CLEAR
SCAN WHILE Vendor_id = mvend_id
   LIST OFF NEXT 10 Part_id, Part_name, Cost WHILE Vendor_id = mvend_id
   WAIT
   CLEAR
ENDSCAN
DEACTIVATE WINDOW goods_win
```

SCAN...ENDSCAN is particularly useful when you want to process selected records. This routine scans an entire unindexed file for accounts that were due a week ago and have an outstanding balance. It then runs a separate procedure that prints a letter for each matching record.

```
SCAN FOR Due_date = DATE() - 7 .AND. Amt_due > 0
    DO Letter WITH Lastname, Firstname, Amt_due
ENDSCAN
```

The next version scans the same file indexed on Due_date. The WHILE condition defines the scope, the group of records you want to SCAN. The FOR condition then selects a subset, the actual records you want within the larger group.

```
SEEK DATE() - 7
SCAN WHILE Due_date = DATE() - 7 FOR Amt_due > 0
    DO Letter WITH Lastname, Firstname, Amt_due
ENDSCAN
```

Variation — Alternate Search Code

Previous versions of dBASE used FIND, SEEK, or LOCATE in a DO WHILE loop to scan a file. The example just shown would have been coded:

```
SEEK DATE() - 7
DO WHILE Due_date = DATE() - 7 .AND. .NOT. EOF()
    IF Amt_due > 0
        DO Letter WITH Lastname, Firstname, Amt_due
    ENDIF
    SKIP
ENDDO
```

Querying Database Files

You can query a database file for fields and records that meet certain conditions. Formulate queries on the menu system's queries design screen, and SET VIEW TO the resulting .qbe file in your program code. The Working with Views section in Chapter 11 shows a typical query and the code that generates it. (Refer to *Using dBASE IV* for more information on the queries design screen.)

You can also build queries directly from dBASE expressions. This section shows some examples.

Simple Queries

A simple query uses one condition. Table 10-2 shows some simple queries based on the Employee database file.

Table 10-2 Simple queries

To find	Query expression
Employees from Sales department	USE Employee ORDER Dept SEEK "SALES" LIST Emp_id, Salary, Date_hired; WHILE Department = "SALES"
Exempt employees	USE Employee LIST Lastname, Firstname,; Specialty FOR Exempt
Employees hired before June 1, 1986	USE Employee ORDER Date_hired LIST Lastname, Firstname; WHILE Date_hired < {06/01/86}
Employees with labor grade other than 2	USE Employee LIST Lastname, Firstname, Laborgrade; FOR Laborgrade <> 2

Complex Queries

A complex query uses more than one condition to qualify the database file. Table 10-3 shows some examples based on the Employee database file

Table 10-3 Complex queries

To find	Query expression
Employees from Nevada or Arizona with MBA degrees	USE Employee LIST Emp_id, City, State; FOR (State = "NV" .OR. State = "AZ"); .AND. Degree = "MBA"
Exempt, full-time employees	USE Employee LIST Lastname, Firstname, Specialty; FOR Exempt .AND. Full_time
Employees hired between June 1, 1983 and December 31, 1983, inclusive	USE Employee ORDER Date_hired LIST Lastname, Firstname; FOR Date_hired >= {06/01/83}; .AND. Date_hired <= {12/31/83}
Employees with labor grade other than 2 who earn less than $20,000	USE Employee LIST Lastname, Firstname, Laborgrade; FOR Laborgrade <> 2; .AND. Salary < 20000
Employees with labor grade above 5, without a bachelor's degree	USE Employee LIST Lastname, Firstname, Degree; FOR Laborgrade > 5 .AND.; .NOT. LIKE("B*",Degree)

Determining Item Existence

Use the SEEK() and LOOKUP() functions to check whether an item exists. For example, if your program doesn't allow duplicate part numbers, these functions can check the user's entry against existing part numbers. In an order entry system, they can make sure an item number exists before your program processes the order.

SEEK() returns a .T. if the search expression is found and moves the record pointer to the record containing the string. LOOKUP() moves the record pointer in an unselected database file to a record with a key expression matching the one in the current file.

This first example uses SEEK() to print vendor names and part numbers for a vendor code of 1000:

```
USE Goods ORDER Vendor_id
mvend_id = "1000"
IF SEEK(mvend_id, Goods)
    LIST OFF Vendor, Part_id WHILE Vendor_id = mvend_id TO PRINTER
ENDIF
```

The following LOOKUP() code is based on the Findvend procedure in Library.prg. Assume that Vendors is active in some work area. Also assume that the user has previously entered a vendor number at a screen prompt, and that this number was passed as the parameter *vendr* to Findvend.

The LOOKUP() function looks for the vendor number in the Vendor_id field of Vendors. If it can't find the number, FOUND() is .F. and the routine sounds a bell and prints a message. Otherwise, LOOKUP() returns the vendor's name (in the Vendor field) to *v_name*. The routine then displays the vendor's name and phone number.

```
v_name = LOOKUP(Vendors->Vendor,TRIM(vendr),Vendors->Vendor_id)
IF .NOT. FOUND()
    DO Warnbell
    ? "Vendor ID: " + TRIM(vendr) AT 2
    ? "was NOT FOUND in Vendors database" AT 2
ELSE
    ? "VENDOR is: " + TRIM(v_name) AT 2
    ? "Phone: " + Vendors->phone AT 2
    ? "for ID: " + vendr AT 16
ENDIF
```

Search Tips

This section provides some tips for conducting successful searches.

1. Character fields often end in varying numbers of spaces. As a rule of thumb, if the last part of an expression consists of character data, TRIM() it:

   ```
   SEEK TRIM(part_name)
   ```

2. When you concatenate character fields, dBASE IV inserts spaces between the fields to fill up the field width. To ensure a successful search, initialize memory variables with the defined field width:

   ```
   mfirstname = SPACE(LEN(Firstname))
   mlastname = SPACE(LEN(Lastname))
   @ 10,10 SAY "Enter first name: " GET mfirstname
   @ 12,10 SAY "Enter last name: " GET mlastname
   READ
   SEEK TRIM(mlastname + mfirstname)
   ```

3. To search on fields of mixed data types, convert all fields to one type, usually character. (See Chapter 3 for more information on data types.)

```
USE Orders
INDEX ON DTOS(Date_trans) + Cust_id TAG Cust_date
SEEK DTOS(DATE())
```

4. Your program should account for the possibility that the record being sought is not in the file. You can test for a failed search in three ways:

   - Test for EOF(): dBASE IV positions the record pointer at the end of the file when a search fails unless SET NEAR is ON.
   - Test for .NOT. FOUND().
   - Test for .NOT. SEEK() (see the second example below).

If you SET NEAR ON before conducting the search, the program will stop at near misses when a FIND, SEEK, or SEEK() fails. The record pointer stops on the record directly following the nonexistent record in the index order. You can test for this with EOF() and FOUND(): both will return a false (.F.).

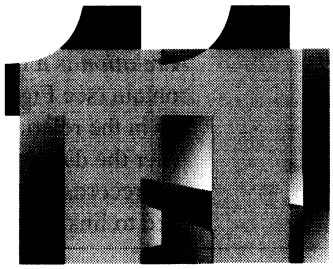
The following routine lets users guess at a vendor code. It searches Vendors in Vendor_id order for the vendor code just after the guess. Then it SKIPs back five records before the guess and LISTs 10 records.

```
USE Vendors ORDER Vendor_id
@ 10,10 SAY "Enter vendor id: " GET Vendor_id PICTURE "9999";
          MESSAGE "Inexact entries OK. Program shows closest values."
READ
SET NEAR ON
SEEK Vendor_id
IF .NOT. FOUND()
   SKIP -5
ENDIF
LIST NEXT 10
```

The Lookupid() user-defined function from Library.prg validates data entered into any of the fields of customer ID, parts ID, vendor ID, and employee ID by checking for their existence in their database files.

The UDF continues the SCAN and LISTs possible vendor numbers until the user types R to return to the main menu.

Processing: Relating and Restricting Data



This chapter describes how you can refine the database environment to improve the power and efficiency of a program. The routines discussed here allow you to link files by a common field, specify which fields and records are active, and capture the environment in a view for use in a program.

What This Chapter Covers

This chapter covers the following topics:

- Relating database files
- Processing selected fields
- Processing selected records
- Working with views

See the *Language Reference* manual for a discussion of the 40 work areas and alias naming conventions. The SELECT() function facilitates work area management. The CATALOG() function provides access to the catalog file, which is in an invisible, dynamically allotted buffer.

Relating Database Files

When you design a database system, you create separate database files to avoid redundancy. However, you might want data from the separate files to appear in a single screen listing or printed report. The SET RELATION command lets you relate database files so that they behave almost like a single file.

The queries design screen lets you build simple relations, where the files are linked on a single field and the child file is indexed on a single field (see *Using dBASE IV).* To establish more complex relations, use the programming methods shown in this chapter.

About Relations

A *relation* is a link between two or more database files on a key field that they both contain (see Figure 11-1). Relations are called *joins* in database theory. The primary file in the relation is known as the *parent,* while the secondary files are the *children.* After the database files are linked, moving the record pointer in the parent file moves it to records with the same index key expression in each child. Relations are generally used to link many records in a child to one record in a parent.

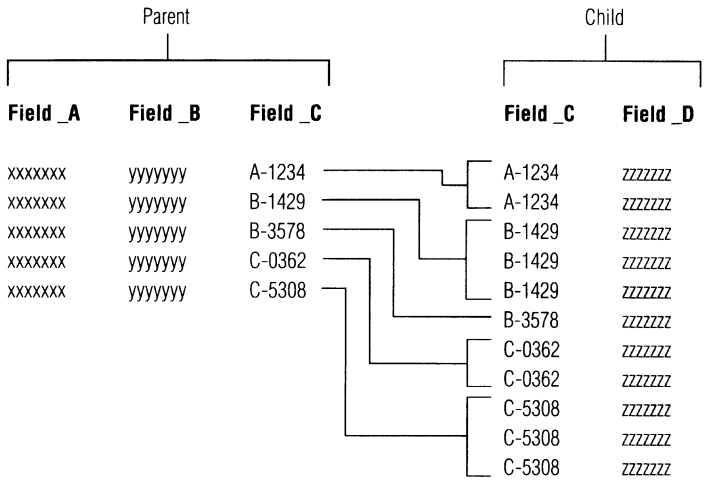


Figure 11-1 Sample relation

Before establishing a relation, decide which database file is the parent and which are the children. For example, suppose you plan to issue a report showing which vendors supply which articles of furniture. You'll need information from both Vendors and Goods for the report; however, the topic of the report is vendors. Therefore, Vendors should be the parent file in this relation, and Goods the child.

The rest of this section shows how to establish different types of relations within a program.

NOTE
Always move the record pointer after a SET RELATION. You can use any command that moves the record pointer, such as GO RECNO(), GO TOP, or SKIP.

One-to-One Relations

A *one-to-one* relation links one parent file to a single child. This type of relation is used primarily for very large database file structures that exceed the maximum number of fields (255). For example, you could handle a 500-question marketing survey by relating two files, each containing half of the questions. Figure 11-2 shows such a one-to-one relation.

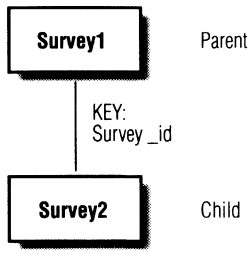


Figure 11-2 One-to-one relation

The code below sets up the relation in Figure 11-2.

```
SELECT 1
USE Survey1
USE Survey2 ORDER Survey_id IN 2
SET RELATION TO Survey_id INTO Survey2
GO TOP
```

One-to-Many Relations

In a *one-to-many* relation, the child file contains *several* records with the same index key as a given record in the parent. In the example shown in Figure 11-3, each vendor supplies several articles of furniture.

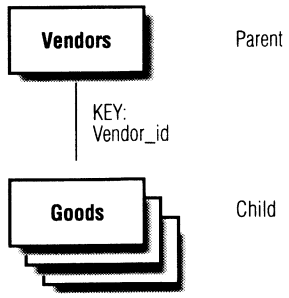


Figure 11-3 One-to-many relation

The following code sets up the relation shown in Figure 11-3. The SET SKIP command ensures that all articles of furniture for each vendor appear when the data is displayed with a command such as LIST or BROWSE. Without SET SKIP in the code, only the first article of furniture would appear.

```
SELECT 1
USE Vendors
USE Goods ORDER Vendor_id IN 2
SET RELATION TO Vendor_id INTO Goods
SET SKIP TO Goods
GO TOP
```

You can relate files in series to build a *relation chain.* Figure 11-4 shows the relation chain Client->Transact->Stock, which could form the basis of an order form. In this example, a customer can order several items, but only one vendor is entered for each item ordered.

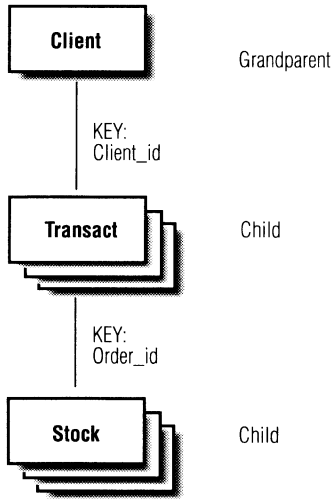


Figure 11-4 Relation chain

The code below establishes the relation in Figure 11-4.

```
CLOSE DATABASES
SELECT 1
USE CLIENT.DBF ORDER CLIENT_ID
USE TRANSACT.DBF IN 3 ORDER CLIENT_ID
USE STOCK.DBF IN 2 ORDER ORDER_ID
SET RELATION TO A->CLIENT_ID INTO C
SELECT 3
SET RELATION TO C->ORDER_ID INTO B
SELECT 1
SET SKIP TO C,B
GO TOP
SET FIELDS TO A->CLIENT_ID /R,C->ORDER_ID /R,B->PART_ID,B->QTY
```

Multiple Child Relations

A *multiple child* relation links one parent file to two or more children (see Figure 11-5).

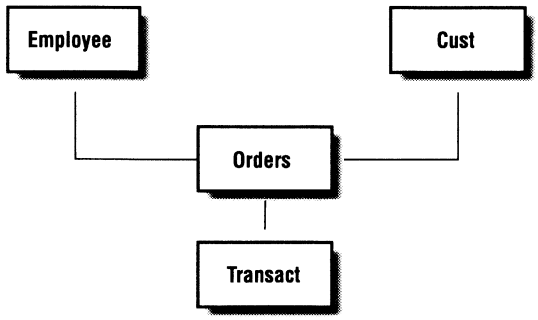


Figure 11-5 Multiple child relation

The following routine relates one parent file with three children. Note that there can be several transactions but only one employee and one customer per order.

```
CLOSE  DATABASES
SELECT 1
USE  ORDERS.DBF  ORDER  PO_NUMBER
USE  CUST.DBF  IN  2  ORDER  CUST_ID
USE  EMPLOYEE.DBF  IN  3  ORDER  EMP_ID
USE  GOODS.DBF  IN  4  ORDER  PART_ID
SET  RELATION  TO  A->CUST_ID  INTO  B
SELECT  2
SET  RELATION  TO  A->EMP_ID  INTO  C
SELECT  3
SET  RELATION  TO  A->PART_ID  INTO  D
SELECT  1
SET  SKIP  TO  B,C,D
GO  TOP
SET  FIELDS  TO  A->PO_NUMBER,B->CUST_ID  /R,C->EMP_ID  /R
```

Using Relations

Once you set up a relation, you can use it in programs for data input and output. The following routine builds a report that processes data from several related database files. First it USEs Orders, Goods, and Vendors in work areas 1, 2, and 3, respectively, and relates the three files in a chain. Then it defines the report date, title, and column headings. Finally, it prints part numbers and names from Goods and vendor numbers from Vendors, grouped by customer number from Orders. The total number of orders prints at the end of the report.

```
mcount = 0
SELECT 1
USE Orders ORDER Order
USE Goods ORDER Part_id IN 2
USE Vendors ORDER Vendor_id IN 3
SET RELATION TO Part_id INTO Goods
SELECT Goods
SET RELATION TO Vendor_id INTO Vendors
SELECT Orders
GO TOP
?? DATE() AT 2
?
? "ORDERS GROUPED BY CUSTOMER NUMBER" AT 20
?
? "Part ID    Part Name    Vendor ID" AT 30
? "=======    =========    ==========" AT 30
DO WHILE NOT EOF()
   mcust_id = Cust_id
   ? mcust_id AT 2
   SCAN WHILE mcust_id = Cust_id
      ?? Goods->Part_id AT 30, Goods->Part_name AT 40
      ?? Vendors->Vendor_id AT 52
      ?
      mcount = mcount + 1
   ENDSCAN
ENDDO
?
? "Total no. of orders: " AT 2, mcount PICTURE "99"
```

The next example, based on procedures in Orders.prg, shows how relations are used with a screen form. First the routine USEs Orders, Goods, Cust, and Employee and RELATEs Orders to the other three database files. Then it GETs the customer number, part number, and employee number from the user and SAYs the corresponding customer name, part name, and employee name.

Only selected GETs and SAYs are shown here. See Orders.prg in the sample code for the entire procedure. Figure 11-6 shows the screen.

```
SELECT 1
USE Orders ORDER Order
USE Goods ORDER Part_id IN 2
USE Cust ORDER Cust_id IN 3
USE Employee ORDER Emp_id IN 4
SET RELATION TO Part_id INTO Goods, Cust_id INTO Cust, Emp_id INTO Employee
GO TOP
DO Screen

PROCEDURE Screen
    .       .
    .       .
   @  2,20 SAY "ORDER TRANSACTIONS DATABASE"
   @  6, 4 SAY "CUSTOMER ID: " GET m->cust_id PICTURE "!99999";
           VALID Lookupc(m->cust_id) ERROR "Invalid customer I.D. - revise."
                 .         .
                 .         .
   @ 10, 4 SAY "PART #:      " GET m->part_id FUNCTION "!";
           VALID Lookupp(m->part_id) ERROR "Invalid part # - revise."
                 .         .
                 .         .
   @ 14, 4 SAY "EMPLOYEE # :" GET m->emp_id PICTURE "999-99-9999";
           VALID Lookupe(m->emp_id) ERROR "Invalid employee number # - revise."
               .         .
               .         .
```

```
   @  6,26 SAY Cust->Customer COLOR &c_yelowhit.
   @ 11,18 SAY Goods->Part_name COLOR &c_yelowhit.
                  .         .
                  .         .
   @ 14,30 SAY TRIM(Employee->Firstname) + " " + Employee->Lastname;
      COLOR &c_yelowhit.
                  .         .
                  .         .
RETURN
```

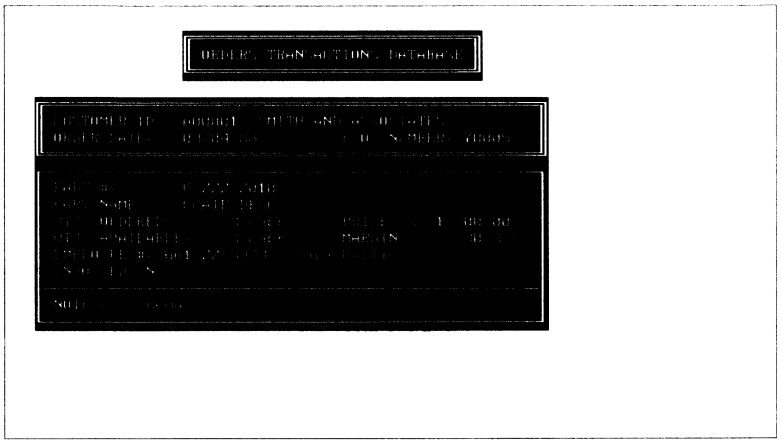


Figure 11-6 Screen form based on a relation

Processing Selected Fields

The SET FIELDS command lets you specify which fields you want dBASE IV to process and in what order. (This is called a *projection* in database theory.) SET FIELDS affects any command that allows you to specify a field list, including BROWSE, EDIT, and LIST. Unlike previous versions of the dBASE product, dBASE IV maintains a single field list for all work areas.

SET FIELDS is really two commands: SET FIELDS TO <field list> and SET FIELDS on/OFF. When you SET FIELDS TO a field list, you automatically SET FIELDS ON. If you no longer want the field list to be in effect, SET FIELDS OFF.

SET FIELDS TO is additive, so each new instance of the command adds fields to the existing list. The only way to delete fields from the list is to delete the field list (SET FIELDS TO with no argument, or CLEAR FIELDS).

The following code specifies an active field list from Vendors:

```
USE Vendors
SET FIELDS TO Vendor, Address1, City, State, Zip
BROWSE
SET FIELDS OFF
```

To include fields from other work areas, open the files in their respective work areas but SET FIELDS TO from the current work area. Precede the names of fields from other work areas with the proper ALIAS. Note the use of a calculated field in the following field list:

```
USE Goods ORDER Part_id
USE Vendors ORDER Vendor_id IN 2
SET RELATION TO Vendor_id INTO Vendors
SET FIELDS TO Part_id, Part_name, Qty_onhand, Cost,;
              total_cost = Qty_onhand*Cost, Vendors->Vendor
BROWSE
SET FIELDS OFF
```

You can use the * and ? wildcard characters with dBASE IV to represent a field list. The following example builds a field list of Part_id, Part_name, Price, and Qty_onhand from the Goods database file:

```
USE Goods
SET FIELDS TO ALL LIKE P*
SET FIELDS TO Qty_onhand
BROWSE
SET FIELDS OFF
```

Processing Selected Records

Sometimes you only want to see certain data on the screen or in a report. For example, you might only want records for a particular vendor, from a particular state, or for a particular range of dates.

Filtering a database file creates a temporary subset of the complete file. (A filter is called a *selection* in database theory.) Use SET FILTER to include records that meet a filter condition, and SET DELETED to exclude records marked for deletion.

Including Matching Records

SET FILTER lets you specify a condition that a record must meet to be included in the filtered database file. SET FILTER filters records *in,* not *out.* Some typical filter expressions are:

- SET FILTER TO State = "CA"
- SET FILTER TO State = "CA" .AND. Zip < "91400"

- SET FILTER TO Date_order = DATE()
- SET FILTER TO Date_trans > mdate .AND. Date_trans < mdate + 30

NOTE
Always move the record pointer after a SET FILTER. You can use any command that moves the record pointer, such as GO RECNO(), GO TOP, or SKIP.

The following code selects employees with more than five years of experience who are earning less than $25,000 per year. The GO TOP positions the record pointer at the first matching record. The last line removes the filter so that it does not affect subsequent operations.

```
USE Employee ORDER Years
SET FILTER TO Yrs_exper > 5 .AND. Salary < 25000
GO TOP
IF .NOT. EOF()
   LIST OFF Lastname, Firstname, Yrs_exper, Salary
ELSE
   ? "No records meet the filter condition"
ENDIF
SET FILTER TO
```

The filter condition in the next routine contains both a logical and a character field. See the Relating Database Files section earlier in this chapter for more information on SET RELATION.

```
USE Orders
USE Goods ORDER Part_id IN 2
SET RELATION TO Part_id INTO Goods
SET FILTER TO Invoiced .AND. Goods->Part_name = "BOOKCASE"
```

Variation — Filtering Large Files

Working with a very large file is often more efficient if you use SET KEY TO RANGE x,y to limit the search range to only the desired records.

```
SET KEY TO RANGE Yrs_exper > 5
```

Excluding Deleted Records

If SET DELETED is ON in your program, records marked for deletion are excluded from many operations (see *Language Reference* for more detail).

The following routine lists information only from records not marked for deletion:

```
USE Employee ORDER Employee
SET DELETED ON
LIST Lastname, Firstname, Phone
SET DELETED OFF
```

You can also SET DELETED ON to COPY only active records TO a new database file.

Working with Views

You can define a view file that assembles one or more database files and their ancillary files (forms, reports, and indexes). Opening the view file opens all the component files and establishes the necessary relations.

Creating a View

You can create two types of views: .qbe views (defined on the queries design screen) and .vue views (defined at the dot prompt). With both types of views, you can open one or more database files, relate files, define a field list, and specify filter conditions. In addition, a .vue view can open a format file and index files or tags. Each type of view has its own advantages:

- You can use .qbe views to specify complex relations, calculated fields, and aggregate operators without programming.
- You can generate dBASE code from .qbe views.
- You can save the current environment to a .vue view using CREATE VIEW <filename> FROM ENVIRONMENT.
- You can save a .vue view in which a relation is based on a calculated field (such as Lastname – Firstname).

Using dBASE IV describes how to define views on the queries design screen (accessed through the menu system). Figure 11-7 shows a typical query definition, and is followed by the code generated for that query. The query opens the Employee and Orders database files in two work areas and links them on the Emp_id field. Then it builds a view containing the fields Emp_id, Lastname, and Department from Employee.dbf and PO_number from Orders.dbf.

NOTE
When you TYPE a .qbe file, you see only the dBASE code at the top of the file. The file also contains internal code for the query processor, but you cannot TYPE this code. There is a **Ctrl-Z** *character (end-of-file) between the two parts of the file.*

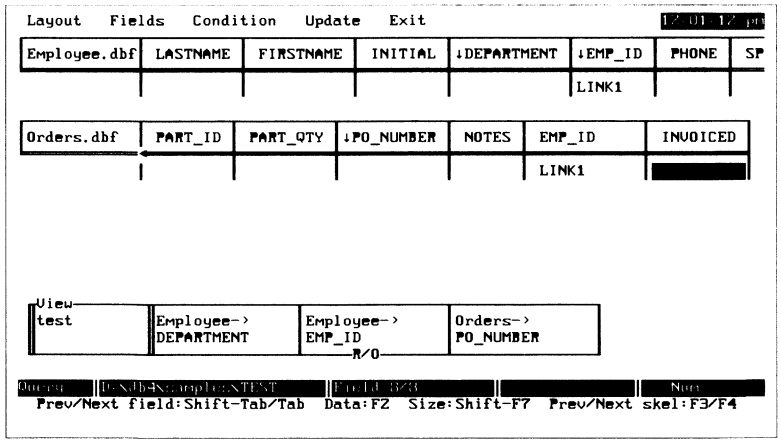


Figure 11-7 Query definition

```
* dBASE IV .QBE file
CLOSE DATABASES
SELECT 1
USE EMPLOYEE.DBF
IF TAGSLOT("EMP_ID") = 0
USE EMPLOYEE.DBF EXCLUSIVE
INDEX ON EMP_ID TAG EMP_ID
ENDIF
CLOSE DATABASES
SELECT 2
USE ORDERS.DBF
USE EMPLOYEE.DBF IN 1 ORDER EMP_ID
SET EXACT ON
SET RELATION TO B->EMP_ID INTO A
SET FILTER TO FOUND(1)
SET SKIP TO A
GO TOP
SET FIELDS TO A->DEPARTMENT,A->EMP_ID /R,B->PO_NUMBER
```

The queries generator produces two files: a queries design (.qbe) file containing dBASE code plus internal code used by the generator, and a query object (.qbo) file containing compiled dBASE code.

WARNING
If you edit .qbe code directly and subsequently make and save changes on the queries design screen, your changes to the .qbe file will be overwritten.

Queries and SET EXACT

In order for queries to run as expected, the query processor SETs EXACT ON. If you are incorporating query code directly into a program, include the following lines in your program to restore the original EXACT status:

```
* Capture the current EXACT status and SET EXACT ON
mexact = SET("EXACT")
SET EXACT ON
.
* <remaining code>
.
* Restore the original EXACT status
SET EXACT &mexact.
```

You create views at the dot prompt by typing the component commands and then entering CREATE VIEW <filename> FROM ENVIRONMENT.

You can convert .vue views to .qbe views using the command MODIFY QUERY <.vue filename>. However, only single-field relations will be represented on the queries design screen. You will have to set up the links again for more complex relations.

Using a View in a Program

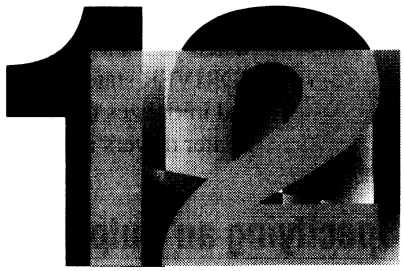
Activating a view opens the files in the view. You activate a view in a program in one of the following ways:

- Create the view on the queries design screen. In your code, SET VIEW TO <.qbe view>. This outputs a compiled .qbo file when you DO the program.
- Create the view at the dot prompt. In your code, SET VIEW TO <.vue view>.
- Create the view in your program code directly by putting the commands that define the view into a procedure file. DO the procedure when you want to activate the view.
- Create the view on the queries design screen. Edit a copy of the resulting .qbe file in MODIFY COMMAND and put the resulting code in your program.

The first two methods call external view files and thus save some lines of code in your program. However, your program will run more slowly because the operating system has to access a separate disk file. The last two methods incorporate the view code directly into your program, without having to call an external program file. These methods ensure that the view code is always available and make your program more efficient.

Rather than closing a view directly when you no longer want it to be in effect, just close the files in the view with CLOSE ALL, CLOSE DATABASES, or CLEAR ALL.

Output: General Considerations



A major purpose of your program is to enable users to produce information in a desired format when they need it. The information generated from a database system is called *data output.* This chapter discusses general output issues such as specifying an output device and using printer control codes. See Chapter 13 for information on how to output data in formatted reports.

What This Chapter Covers

This chapter covers the following topics:

- Customizing dBASE IV for printing
- Specifying an output device
- Special print effects
- Printing tips

Customizing dBASE IV for Printing

You can customize dBASE IV for a particular printer configuration by including certain commands in the Config.db file. This section briefly mentions the relevant commands. See *Getting Started with dBASE IV* for more details.

The Config.db PRINTER command installs up to four printer drivers. Optionally, it also specifies a name to appear in the menu system and a printer destination or device for each printer driver.

The Config.db PRINTER FONT command specifies up to five user-defined fonts per printer driver. These are in addition to fonts for boldface, italics, underlining, superscript, and subscript, which are supported automatically for installed printer drivers. The STYLE option of the ?/?? command prints text in these special fonts.

The Config.db PDRIVER command activates a particular printer driver (.pr2) file. When you start dBASE IV, it assumes a GENERIC default printer driver. If there is a PDRIVER statement in Config.db upon startup, it activates the named printer driver and initializes the _pdriver system variable with the same driver. You can also switch printer drivers in a program with a *_pdriver = <driver filename>* statement.

Specifying an Output Device

Your program can output to the screen, to a printer, or to a file. The output device you choose depends not only on what the user plans to do with the information, but also on characteristics of the information itself.

General Considerations

Some general questions will help you determine what output device to use.

How wide is the output? If it is over 80 characters wide, don't display the data on the screen. Print on wide paper or use compressed mode with regular width paper.

How long is the output? If it is over 24 lines long, part of it may scroll out of view in a screen display. Is this acceptable to users? If not, your program should print the file or let users pause the display.

For lengthy output that might be printed on slower printers, your program should let users send the report to a file for printing later.

Streaming Output

Most dBASE IV output commands, including DIR, DISPLAY, LABEL FORM, LIST, REPORT FORM, TEXT...ENDTEXT, EJECT PAGE, and TYPE, produce a stream of characters known as a *data stream* or *streaming output.* (The @ commands and EJECT do not produce streaming output.) Streaming output is just a sequence of characters. Output commands operate directly on this stream rather than on the device to which the data is being sent.

Streaming output goes to all available devices, while non-streaming output goes only to the specified device. For example, suppose CONSOLE is ON and an ALTERNATE file is open:

- If you SET DEVICE TO PRINTER, output from the @...SAY command goes *only* to the printer.
- If you SET PRINTER ON, output from the ?/?? command goes to the printer, screen, and the alternate file.

Streaming output starts at the current position on the screen, printer, or in the output file. Instructions in the output, such as the AT option of the ?/?? command, determine the actual output position.

Note that EJECT PAGE is a streaming output command while EJECT is not. When you EJECT PAGE, dBASE IV resets _plineno to 0, advances _pageno to the next page number, and executes the page handler program (ON PAGE). When an @...SAY causes an eject or you issue an EJECT, however, these changes do not occur.

The Page Breaks section in Chapter 13 shows a page handler program.

Specifying the Output Device

How you specify an output device depends on whether the output is streaming or non-streaming. Table 12-1 shows the sets of commands used to send output to the printer, screen, or a disk file.

Table 12-1 Commands for specifying the output device

Output format	Output device		
	Printer	**Screen**	**File**
Streaming output	SET PRINTER TO [SPOOLER] <queue name> or SET PRINTER TO LOCAL then SET PRINTER ON	SET DISPLAY TO <type> SET CONSOLE ON SET PRINTER OFF	SET PRINTER TO FILE <filename> SET PRINTER ON or SET ALTERNATE TO FILE <filename> SET ALTERNATE ON
Non-streaming output	SET DEVICE TO PRINTER	SET DISPLAY TO <type> SET DEVICE TO SCREEN	SET DEVICE TO FILE <filename>

DISPLAY, LABEL FORM, LIST, REPORT FORM, and TYPE also have TO PRINTER and TO FILE clauses that can be used instead of SET PRINTER ON.

SET PRINTER TO allows you to direct output to a different printer. For example, SET PRINTER TO <DOS device> allows you to direct output to a specific printer such as LPT1 or LPT2.

When you send output to a file, dBASE IV generates a disk file. SET ALTERNATE TO FILE and SET DEVICE TO FILE create ASCII text files with the default extension .txt. These files contain output from ?/?? commands and @...SAYs but not @...GETs. SET PRINTER TO FILE creates a .prt file containing printer control codes unless you define *_pdriver = "ASCII.pr2"*, in which case it generates an ASCII .txt file containing no printer control codes.

Providing a Destination Menu

You can give users more control over the output environment by letting them specify an output device. The following procedure would accompany code for a Destination menu like the one shown in Figure 12-1. This procedure and Bar_def, the menu definition code, are from Library.prg.

The first three CASEs in the procedure direct output to the printer, a disk file, and the screen, respectively. The last CASE exits the menu. Note that it stores the current page settings (_plength and _rmargin) before configuring the page for screen output. After outputting to the screen, the procedure restores the original dimensions.

Figure 12-1 Destination menu

```
PROCEDURE Barpop_d
   SET COLOR TO &c_popup.
   DO CASE
      CASE BAR() = 2          && Output to printer
         DO Prt_menu
         SET PRINTER ON
         SET CONSOLE OFF
         DO Printout
         SET PRINTER OFF
         SET CONSOLE ON
      CASE BAR() = 3          && Output to file
         answer = SPACE(8)
         ACTIVATE WINDOW alert
         @ 0,0 SAY "---------- SEND REPORT TO FILE ----------"
         @ 2,1 SAY "Enter filename for report: " GET answer;
               VALID "" <> TRIM(answer);
               MESSAGE "Enter a filename of up to eight characters"

         READ
         DEACTIVATE WINDOW alert
         SET ALTERNATE TO (answer)
         SET ALTERNATE ON
         SET CONSOLE OFF
         DO Printout
         SET ALTERNATE OFF
         CLOSE ALTERNATE
         SET CONSOLE ON
      CASE BAR() = 4          && Output to screen
         SET COLOR TO &c_standard.
         CLEAR
         plength = _plength
         rmargin = _rmargin
         _plength = 25
         _rmargin = 80
         DO Printout
         CLEAR
         _plength = plength
         _rmargin = rmargin
         GO record_num
      CASE BAR() = 5          && Exit to OPTION MENU
         DEACTIVATE POPUP
   ENDCASE
   DEACTIVATE POPUP
RETURN
```

Special Print Effects

You can produce special print effects such as lines and boxes, different typefaces, and graphic characters. Special effects draw the user's attention to particular items in the output and guide the eye along the printed page.

Table 12-2 summarizes some of the special print effects you can create with dBASE IV. (See the comment box below for an explanation of printer control codes.) Unless otherwise indicated, the examples shown in this section work with an IBM Proprinter XL. If you're using another printer, consult your printer manual for the proper codes. Your printer must have the necessary features for the examples to work.

Table 12-2 Special print effects

Desired effect	Example
Print a line	?? REPLICATE(CHR(205),45) AT 5
Print a box	DEFINE BOX FROM 5 TO 50 HEIGHT 9 AT LINE 3
Print graphic characters	?? CHR(252)
Print in color*	magenta = "{ESC}A" ??? magenta
Change typeface Printer font defined in Config.db	In Config.db: PRINTER 1 = "Ibmgp.pr2" PRINTER 1 FONT 1 = {ESC}E,{ESC}F Then in program: ? "Amount owed: $" + STR(balance) STYLE "1"
Not defined in Config.db	??? "{ESC}E" ? "Amount owed: $" + STR(balance) ??? "{ESC}F"
Change type size Standard Non-standard	 _ppitch = "ELITE" ??? "{ESC}E"
Combine effects Using STYLE option Using ???**	 ? "Payment 45 days overdue!" STYLE "BU" ??? "{ESC}G{ESC}0"

* Escape code for a Fujitsu DPL24D color printer.

** On an IBM Graphics Printer, this escape code sets double-strike on and sets the line spacing to 1/8 inch.

Note the different commands for producing lines and boxes on the screen and printer:

@...TO *draws* lines in non-streaming output. Use ? REPLICATE() to *print* horizontal lines in streaming output.

@...TO also *draws* boxes on the screen in non-streaming output. Use DEFINE BOX to *print* boxes in streaming output. The _box system variable must be set to true (.T.), its default value, for boxes to print.

@...CLEAR TO clears lines and boxes from the screen. Set _box to .F. if you don't want boxes to print.

Figure 12-2 shows a printout that uses some of these special print capabilities.

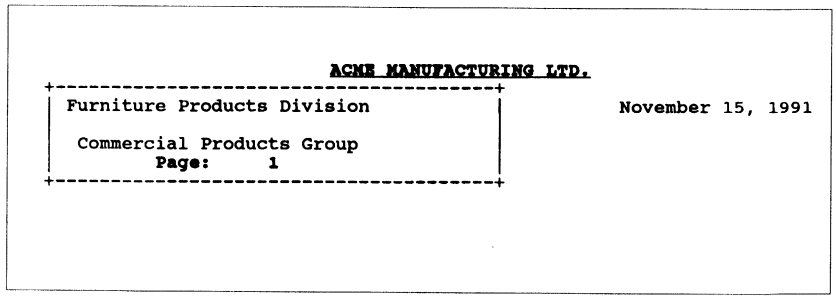


Figure 12-2 Special print effects

Following is the code for this printout. First it sets _alignment to "CENTER" and DEFINEs the BOX. (Note that _wrap must be .T. for _alignment to work.) Then it prints the title, "ACME MANUFACTURING LTD." centered on an 80-column page in boldface type and underlined. After setting _alignment to "LEFT", it prints the rest of the text. The box prints automatically because _box is set to .T. (true).

```
SET CENTURY ON
SET TALK OFF
_plineno = 0
_alignment = "CENTER"
_wrap = .T.
_rmargin = 80
DEFINE BOX FROM 5 TO 50 HEIGHT 6 AT LINE 4
SET PRINTER ON
_box = .T.
?
? "ACME MANUFACTURING LTD." STYLE "BU"
?
?
_alignment = "LEFT"
? "Furniture Products Division" AT 7
?? MDY(DATE()) AT 60
?
? "Commercial Products Group" AT 8
? "Page: " + STR(_pageno,5,0) AT 15 STYLE "B"
?
?
?
SET PRINTER OFF
_wrap = .F.
_box  = .F.
```

Printer Control Codes

In dBASE IV, the ??? command bypasses the installed printer drivers and sends data (usually printer control codes) directly to the printer. You can use printer control codes to print styled text, such as boldface and italics, or otherwise control the printer. Many printer control codes consist of an Escape character (ASCII code 27) followed by other characters.

The following example switches to landscape mode and back on a Hewlett-Packard LaserJet. Note that the Escape character is represented by {ESC}. dBASE IV provides a number of such *control character specifiers* for use with the ??? command (see Chapter 2 of *Language Reference).*

```
SET PRINTER ON
??? "{ESC}&l1O"       && Letter "l", number 1, uppercase letter "O"
DO Widerept
??? "{ESC}&l0O"       && Letter "l", number 0, uppercase letter "O"
SET PRINTER OFF
```

To avoid typing complex printer control codes, initialize memory variables with the code strings. The following memory variable definitions for the HP LaserJet cartridge switch from Courier to Helvetica bold, from Helvetica to Times Roman, and from Times Roman to the default Courier typeface:

```
helvbold = "{ESC}(0U{ESC}(s1p14.4v3B"
tmsroman = "{ESC}(0U{ESC}(s10v0B"
reset_def = "{ESC}(8U"
```

Printing Tips

Here are some tips for handling common printing problems:

- If the last line won't print, the print buffer may be waiting for an end-of-line carriage return (ASCII 13). To supply this, set _peject to "BOTH" or "AFTER" in a print job, or end the output with an EJECT PAGE or ? command.
- If an unwanted form feed occurs at the beginning of a print job while printing with REPORT FORM, use the NOEJECT option or set _peject to "NONE" or "AFTER" to suppress the form feed.

- If an unwanted initial form feed occurs while printing a file formatted with @...SAYs, the previous printer or screen coordinate that you specified may be greater than the current one. Here's what to do:

  1. Terminate the last print operation with an EJECT command.
  2. Put an EJECT command prior to the SET DEVICE TO SCREEN command in the faulty printing routine.

- Use the command ??? CHR(0) or ??? {null} to send a null character if your printer requires one as part of a string of control characters.

- Remember to check the printer status before printing:

```
IF PRINTSTATUS()
   .
   . <commands to print the report>
   .
ELSE
   WAIT "Printer is not ready"
ENDIF
```

Output: Getting Data from the System

After validating and processing data entered by the user, your program is ready to output data from the database system. Many application programs use custom reports to output data in an attractive and useful format. This chapter looks at the code underlying some common types of reports. (See Chapter 12 for general printing information, such as how to install and select a print device.)

What This Chapter Covers

This chapter covers the following topics:

- Preparing data for output
- Outputting unformatted data
- Tabular reports
- Handling print jobs
- Printing memo fields and long expression results
- Other types of reports
- Running reports

Preparing Data for Output

To prepare data for output, you need to order the data and select fields and records to go in the report.

Ordering the Database File

Data output can appear in natural order (the order in which the data was entered), or in index order. Users rarely want to see the data in natural order, because they cannot easily locate records if the file is not organized. Index order outputs data in alphabetical, numeric, or date order, or a combination. Chapter 9 discusses indexing procedures.

Filtering Output

A database can contain thousands of records. Your program lets users view a subset of these records on the screen or in a printed report by *filtering* the records. Here are three comparable ways to filter data and list records:

```
LIST OFF City, State, Zip FOR State = "CA"

SET FILTER TO State = "CA"
GO TOP
LIST OFF City, State, Zip

SCAN FOR State = "CA"
   ? City AT 0, State AT 15, Zip AT 21
ENDSCAN
```

However, filtering might prove too slow if your application uses large database files. The faster method shown below first ORDERs the database file on the key field Oldbalance. Then it SEEKs the smallest possible balance (one cent). The command SET NEAR ON causes the record pointer to stop at the value closest to .01 if .01 is not found, rather than going to the end of the file. Finally, the routine SCANs the database file and prints data as long as Oldbalance is greater than zero.

The indexing method is faster because it groups all records matching the filter condition together. dBASE IV only has to search for the first record in the group; any other matching records follow directly.

```
USE Acct_rec ORDER Oldbalance
SET NEAR ON
SEEK .01
? "CustID   Date      Last bill Last paid   Balance"
SCAN WHILE Oldbalance > 0
    ? Cust_id, Dat_lstbil, Amt_lstbil, Amt_lst_pd, Oldbalance
ENDSCAN
```

The Processing Selected Records section in Chapter 11 discusses filtering in more detail.

Outputting Unformatted Data

The DIR, DISPLAY, LIST, TEXT...ENDTEXT, and TYPE commands output minimally formatted data. For example, LIST displays data in columns with field names above each column and the record number at the beginning of each row. You can suppress the column headings with SET HEADING OFF and the record number with the OFF option.

For example, the following procedure from Library.prg LISTs records ten at a time until the user types R to return to the calling menu. Since the SCAN moves the record pointer, the routine stores the record number before doing the SCAN and repositions the pointer afterwards.

```
PROCEDURE List_rec
   record_num = RECNO()
   ACTIVATE WINDOW lister
   answer = " "
   CLEAR
   @ 0,1 SAY "------- LIST RECORDS ----------";
         COLOR &c_red.
   SCAN WHILE .NOT. answer $ "rR"
      LIST OFF NEXT 10 &list_flds.
      WAIT "Press spacebar to continue or R to return to Option menu" TO answer
      CLEAR
   ENDSCAN
   DEACTIVATE WINDOW lister
   GO record_num
RETURN
```

Tabular Reports

This section uses Emp_rept, the custom tabular report shown in Figure 13-1, to demonstrate typical data output code. With the exception of the *Variations,* most of the code examples are based on this report. See the Other Types of Reports section for information on how to code other report layouts and the Running Formatted Reports section for how to print or display a report.

As indicated in the figure, separate elements of the report correspond to distinct procedures in Emp_rept. For each element, the three columns on the right identify the element name, the procedures that produce it, and the section in this chapter that describes it. Use these references to find out how particular report elements are coded. For example, if you want to learn how the department summary data is coded, look up the Det_calc and Brk_data procedures in the sample code or refer to the Summary Data section in this chapter.

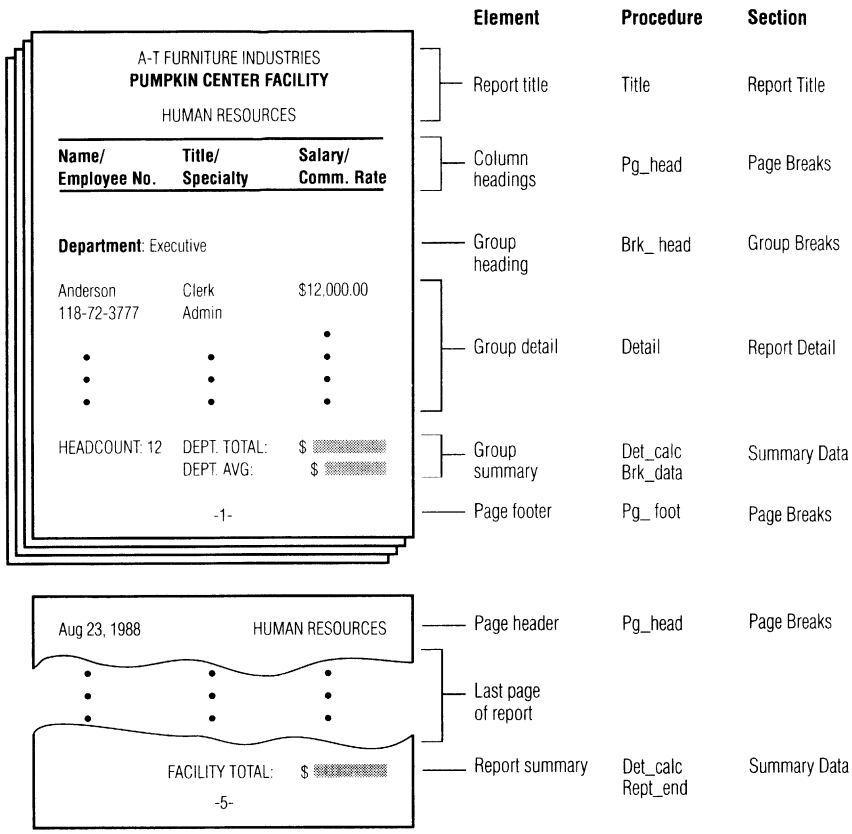


Figure 13-1 Printout of Emp_rept

As an alternative to coding reports on your own, you can create them on the reports design screen (see *Using dBASE IV*). The reports generator produces a report design (.frm) file containing the internal code used by the generator, and a generated report program (.frg) file containing dBASE code. You can modify the code in the .frg file directly.

WARNING
If you change the code in an .frg file and subsequently make and save changes on the reports design screen, your changes to the .frg file will be overwritten.

Environmental Setup

The following code sets up the working environment for Emp_rept. First it opens and orders the Employee database file (the Dept index tag orders by department, last name, and first name). Then it configures the **Esc** key to exit the report (by setting print_flag to .F.) and SETs CENTURY ON. The routine then initializes memory variables used throughout the program. These variables are explained in Table 13-1. Finally, it stores the system variable settings that will be changed during the program.

```
USE Employee ORDER Dept
ON ESCAPE print_flag = .F.
SET CENTURY ON

rpt_level = 1
print_flag = .T.
on_pg_line = 0
STORE 0 TO pageno, salary_sum, salary_gt, number_emp, yrsexp_sum,
yrsexp_avg
group_brk = ""
p_title = "Human Resources Report"

mplength = _plength
mploffset = _ploffset
mlmargin = _lmargin
mrmargin = _rmargin
mppitch = _ppitch
mwrap = _wrap
mpeject = _peject
```

Table 13-1 Memory variables used in Emp_rept

Memory variable	What it stores
rpt_level	Level 0 = text (such as header or footer), level 1 = first group detail, level 2 = second group detail, and so on
print_flag	Continue printing if .T., stop if .F.
on_pg_line	The line at which the ON PAGE procedure executes
pageno	Current page number
salary_sum	Department subtotal for salary
salary_gt	Salary column grand total
number_emp	Department subtotal for number of employees
yrsexp_sum	Department subtotal for years of experience
yrsexp_avg	Department average for years of experience
group_brk	Department name for current group
p_title	Page header text

(continued)

Table 13-1 Memory variables used in Emp_rept *(continued)*

Memory variable	What it stores
mplength	Current page length
mploffset	Current page left offset
mlmargin	Current left margin
mrmargin	Current right margin
mppitch	Current printer pitch
mwrap	Current word wrap setting
mpeject	Current page eject setting

Many reports combine information from more than one database file. The following code from Invoices.prg in the sample code sets up the database environment for a multi-file report using four database files:

```
SELECT 1
USE Orders ORDER Order IN SELECT()
USE Cust ORDER Cust_id IN SELECT()
USE Acct_rec ORDER Cust_id IN SELECT()
USE Goods ORDER Part_id IN SELECT()
SET RELATION TO Cust_id INTO Cust, Cust_id INTO Acct_rec, Part_id INTO
Goods
GO TOP
```

Chapter 11 discusses relations and views in more detail.

Page Layout

The dBASE IV system memory variables and the ON PAGE command control the *page layout* — the arrangement of text on the page (see Figure 13-2).

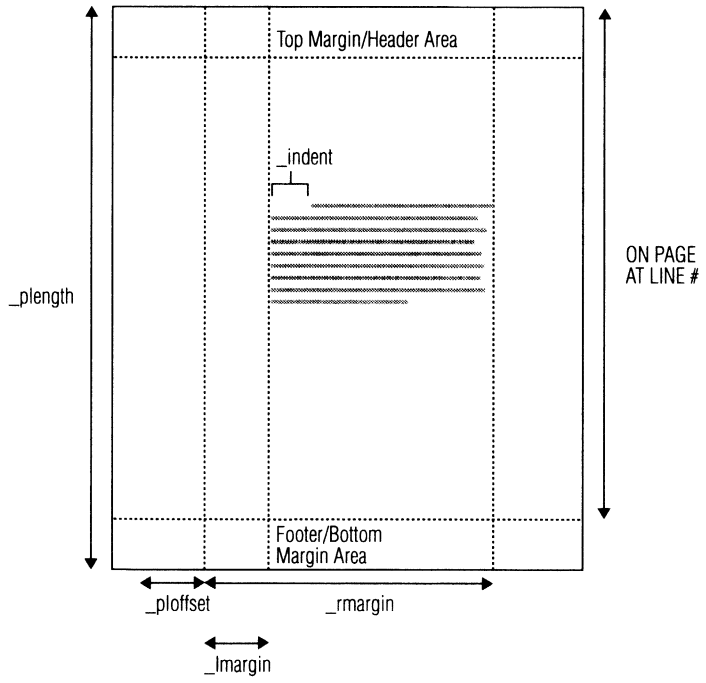


Figure 13-2 Controlling the page layout

The following Emp_rept setup code sets the page length to 66 lines, the left offset and left margin to 0, and the right margin to 80. These are the default settings.

The code then defines the line number where page breaks occur *(on_pg_line)* as the page length minus five lines. This variable is later used with the ON PAGE command to execute page breaks on the correct line. Note that the code does not set top and bottom margins. They are implicit in the header and footer routines (see the Page Breaks section later in this chapter).

```
_plength = 66
_ploffset = 0
_lmargin = 0
_rmargin = 80
on_pg_line = _plength - 5
```

TIP

If you plan to use the same system variable settings in many reports, enter the definitions at the dot prompt and save them to a memory file:

```
. SAVE TO Page_fmt ALL LIKE _*
```

Then RESTORE them in the page layout procedure of each report:

```
RESTORE FROM Page_fmt ADDITIVE
```

The most carefully thought-out page settings are useless if the paper is not properly positioned in the printer. Run a test program like the following to find out the home position on your printer:

```
answer = "T"
SET PRINTER ON
DO WHILE answer = "T"
    ? "Testing ..." AT 1
    WAIT "Adjust paper, then press any key to stop or T";
         "to continue testing" TO answer
    EJECT PAGE
ENDDO
SET PRINTER OFF
```

Then adjust the paper to the top-of-form.

Main Report Program

The main program of Emp_rept consists of a series of calls to the many procedures in the report. Thus, the main program provides an overview of the logic of Emp_rept. Most of the procedures called in this routine are shown in the following sections.

First, the program SETs PRINTER ON. Then the PRINTJOB construct defines a print job including all commands up to the closing ENDPRINTJOB command (see comment box). Each time the user prints the report, these commands execute.

Within the print job, the routine prints the report title and executes Pg_head and Brk_head, which print the page header and group heading for the first group. Then the GO TOP positions the record pointer at the first record to be printed, and the SCAN...ENDSCAN processes records until it reaches the end of the file or the user presses **Esc** (which sets *print_flag* to .F.).

Within the SCAN, the routine first checks whether a break on department has occurred. If so, the Brk_data procedure outputs summary data for the department that has just finished printing. Then Reinit reinitializes variables and Brk_head prints the group heading of the next department. After the ENDIF command, Detail prints the data for the next department, Det_calc calculates the summary data for that department, and the SCAN ends.

The final code segment processes the end of the report. It prints summary data for the last group (if *print_flag* is still .T.) and then prints summary data for the entire facility.

The ON PAGE command instructs dBASE IV to print the page footer (procedure Pg_foot) when the printer finishes printing the line number defined by *on_pg_line*. The EJECT PAGE command automatically prints blank lines until this line is reached.

```
SET PRINTER ON
   ON PAGE AT LINE on_pg_line DO Pg_foot

PRINTJOB
   DO Title
   DO Pg_head
   DO Brk_head
   GO TOP
   SCAN WHILE print_flag
      IF group_brk <> Department
         rpt_level = 1
         DO Brk_data
         DO Reinit
         DO Brk_head
      ELSE
         rpt_level = 0
      ENDIF
      DO Detail
      DO Det_calc
   ENDSCAN
   IF print_flag
      DO Brk_data
   ENDIF
   DO Rpt_end
   EJECT PAGE
ENDPRINTJOB
```

Handling Print Jobs

The dBASE IV PRINTJOB...ENDPRINTJOB construct is a convenient tool for handling print jobs. The initial PRINTJOB statement instructs dBASE IV to send starting print codes if any (defined by _pscode) to the printer and eject a page if _peject is set to BEFORE or BOTH. The closing ENDPRINTJOB statement sends any _pecode control code to the printer, ejects a page if _peject is set to AFTER or BOTH, and loops back to the PRINTJOB statement for the number of copies defined by _pcopies.

To ensure that all records print for multiple copies, move the record pointer *within* the PRINTJOB...ENDPRINTJOB construct to the first record to be printed. You can use a GO TOP command or a SCAN with no WHILE condition.

Be sure to set up the print environment (see the Environmental Setup section earlier in this chapter) *before* the PRINTJOB construct in your code. This ensures that the environmental system variables are in effect throughout the entire print job. You might compare them to the instructions that you give a printing service when you turn in a print job. After the PRINTJOB construct, restore the original print environment (see the Environmental Cleanup section later in this chapter).

Report Title

Here is the routine that prints the Emp_rept report title. The _alignment system variable centers the lines between the left and right margins (_wrap must be .T.). The "B" and "BU" in the ? commands output the text in boldface type and underline it.

```
PROCEDURE Title
   ?
   _wrap = .T.
   _alignment = "CENTER"
   ? "A-T FURNITURE INDUSTRIES" STYLE "B"
   ? "PUMPKIN CENTER FACILITY" STYLE "BU"
   ?
   ? "HUMAN RESOURCES REPORT - QUARTERLY MANAGEMENT MEETING"
   ?
   ? MDY(DATE())
   _wrap = .F.
   _alignment = "LEFT"
RETURN
```

Page Breaks

In dBASE IV report programming, page breaks are processed by the ON PAGE command. It calls a page break routine known as a *page handler* after the output in the line specified in the ON PAGE command has printed. Below is the code that handles page breaks in Emp_rept. (The ON PAGE command is shown here for clarity. It actually appears in the program setup code.)

When the output finishes printing the line number specified in the ON PAGE command *(on_pg_line),* the Pg_break procedure executes. It calls, in turn, footer, header, and group break procedures. If the current page number is greater than or equal to the ending print page requested by the user, the report terminates.

The footer procedure, Pg_foot, defines pageno as the TRIMmed STRing of the _pageno system variable placed between hyphens. It prints the page number centered at the bottom of the page, three lines below the last line of the report. After restoring _wrap and _alignment to their default values, it EJECTs the PAGE.

The header procedure, Pg_head, prints the date and report title (*p_title*) at the top of each page of the report except the first. It right-justifies the title by subtracting its length from the right margin. Then it prints the column headings for the report, boxed and in boldface type.

The Brk_head procedure is described in the Group Breaks section.

```
ON PAGE AT LINE on_pg_line DO Pg_break

PROCEDURE Pg_break
   DO Pg_foot
   IF _pageno >= _pepage
      GO BOTTOM
      SKIP
      rpt_level = 0
      RETURN
   ENDIF
   DO Pg_head
   DO Brk head
RETURN

PROCEDURE Pg_foot
   pageno = "-" + LTRIM(STR(_pageno,3,0)) + "-"
   _wrap = .T.
   _alignment = "CENTER"
   ?
   ?
   ? pageno
   _wrap = .F.
   _alignment = "LEFT"
   EJECT PAGE
RETURN

PROCEDURE Pg_head
   ?
   IF _pageno <> 1
      ?
      ? MDY(DATE()) AT 0, p_title AT (_rmargin - LEN(p_title))
   ENDIF
   ?
   ?
   DEFINE BOX FROM 0 TO 79 HEIGHT 4
   ? "Name/" STYLE "B" AT 1, "Title/" STYLE "B" AT 30;
     "Salary/" STYLE "B" AT 48, "Exper." STYLE "B" AT 62;
     "Degree" STYLE "B" AT 73
   ? "Employee No." STYLE "B" AT 1, "Specialty" STYLE "B" AT 30;
     "Comm. Rate" STYLE "B" AT 48
   ?
RETURN
```

The rest of this section shows variations in the handling of page breaks, headers, and footers.

Variation #1 — Breaking the Page on a Memo Field

Page breaks can occur at an awkward point in a memo or long character field. Your page handler code should include instructions to prevent one or two lines at the top or bottom of a page. The following code prevents less than three lines from printing at the bottom of the page:

```
IF _plineno + 3 > on_pg_line
   EJECT PAGE
ENDIF
```

Variation #2 — Starting Page Numbers on Page 2

Many people don't want the page number to show on the first page of a report. This footer routine suppresses the page number display on page 1, but prints the rest of the footer:

```
PROCEDURE Footer
   ? "Prepared by Corporate Human Resources Department" AT 18
   IF _pageno <> 1
       ? "Page ", _pageno PICTURE "999" AT 37
   ENDIF
RETURN
```

Variation #3 — Different Headers and Footers on Facing Pages

Your application may require different headers or footers on facing pages. The following example assumes that you've written two header procedure files. The MOD() function distinguishes odd-numbered from even-numbered pages.

```
ON PAGE AT 3 DO Dif_head

PROCEDURE Dif_head
   IF MOD(_pageno,2) = 0
      DO Evenhead
   ELSE
      DO Odd_head
   ENDIF
RETURN
```

Variation #4 — Different Header or Footer Depending on Condition

Some applications require different headers or footers in different circumstances. The following routine prints different messages on a billing statement depending on the amount due.

```
PROCEDURE Dif_foot
   DO CASE
      CASE Oldbalance > 1000
         ? "Very large balance due from previous invoice. " +;
           "Please call number shown on your statement. "
      CASE Oldbalance > 100
         ? "Large balance due from previous invoice. Please pay promptly."
      OTHERWISE
         ? "Balance due from previous invoice."
   ENDCASE
RETURN
```

Variation #5 — Page x of y

You might want to print the page number in the format *Page 3 of 10*. The example shown here assumes that each record occupies only one line. (The calculation becomes more complicated if the report contains variable-length fields such as memo fields.)

First, the routine calculates *detail*, the number of lines available for report detail. (Assume that you've previously initialized *headlines* and *footlines* with the number of lines required by your header and footer procedures.) Then the Grup_cnt procedure counts the number of lines required by group summaries and headings and the number of detail lines.

The routine divides the number of lines in the report by the number of detail lines per page to compute *pages*, the total number of pages. Finally, it displays the current page number and the total number of pages in the format *Page x of y*.

```
lines = 0
detail = _plength - (headlines + footlines)
DO Grup_cnt
pages = lines/detail
? "Page " + LTRIM(STR(_pageno,3,0)) + " of " + LTRIM(STR(pages,3,0))

PROCEDURE Grup_cnt
   DO WHILE .NOT. EOF()
      m->grp_break = Department
      lines = lines + <n>      && <n> is no. of lines for group head and summary
      SKIP
      SCAN WHILE grp_brk = Department
         lines = lines + <n>   && <n> is no. of detail lines per record
      ENDSCAN
   ENDDO
   GO TOP
RETURN
```

A slower but simpler method to determine the last page of the report is:

```
lastpage = 0
SET CONSOLE OFF
SET PRINTER OFF
DO <custom report>   && Or REPORT FORM <report>
lastpage = LTRIM(STR(_pageno,3,0))
SET CONSOLE ON
SET PRINTER ON
```

Report Detail

The report detail is the data that a report prints for each selected row in the database file or view.

The Detail procedure shown below outputs this data for Emp_rept. First, the IF...ENDIF ejects a page if there are five or fewer lines left on the page. Then the procedure prints the report detail.

PICTUREs and PICTURE functions format the data. (The initial capital letters of Lastname and Firstname were formatted at the data entry stage, so they are not formatted here.) Note the "@T" PICTURE function, which trims the Lastname field so that there is no gap before the comma separating it from Firstname. The AT option aligns the fields under the column headings defined earlier (see the Page Breaks section earlier in this chapter).

```
PROCEDURE Detail
   IF _plineno + 5 > on_pg_line
      EJECT PAGE
   ENDIF
   ?
   ?? Lastname PICTURE "@T XXXXXXXXXXXXXXX" AT 0, ","
   ?? Firstname PICTURE "XXXXXXXXXX"
   ?? Title PICTURE "XXXXXXXXXXXXXXX" AT 30
   ?? " $", Salary PICTURE "99,999.99" AT 48
   ?? Yrs_exper PICTURE "99.9" AT 63
   ?? Degree PICTURE "XXX" AT 73
   ?  Emp_id AT 0, Specialty AT 30
   IF Rate <> 0
      ?? Rate PICTURE "99.9" AT 48, " %"
   ENDIF
   ?
RETURN
```

Printing Memo Fields and Long Expression Results

dBASE IV provides capabilities for printing memo fields and lengthy expression results. The "H", "V", "I", "B" and "J" FUNCTIONs used with the ?/?? command control stretching and alignment of long fields.

Use the "H" or *horizontal stretch* FUNCTION to wrap a memo field within the left and right margins set by _lmargin and _rmargin. Be sure _wrap is .T. or dBASE IV will ignore the FUNCTION:

```
_wrap = .T.
?? Notes FUNCTION "H"
```

The "V<n>" or *vertical stretch* FUNCTION wraps an expression result within the column width defined by the numeric argument. Again, _wrap must be true. The following commands print Notes at column 13 and wrap it in a column width of 25 characters.

```
_wrap = .T.
?? Notes FUNCTION "V25" AT 13
```

TIP

Use FUNCTION "V0" to overstrike text. The value of _pcolno (the print column) does not increment as the expression prints, so the next expression you print overlays it.

Use the "I" FUNCTION to center a field within a column:

```
_wrap = .T.
?? Notes FUNCTION "IV25" AT 13
```

Similarly, left- and right-align the memo field using the "B" and "J" FUNCTIONs, respectively.

You can print a memo field in a box. In the following code, the first two statements define the width of the memo field to the lesser of (_rmargin – _lmargin) or 35. Then the routine opens a print job and prints fields from the database file. When it reaches the memo field, it executes the Box_memo procedure.

Box_memo DEFINEs a BOX two characters wider and taller than the memo field, to allow for the box border. (MEMLINES() returns the number of lines in the memo field.) Then the procedure prints the memo field starting one column to the right of the left edge of the box. The _box = .T. statement is required to make the box print.

```
mem_width = MIN(_rmargin - _lmargin, 35)
SET MEMOWIDTH TO mem_width
PRINTJOB
   SCAN
      * <?/?? fields before memo field>
      DO Box_memo
      * <?/?? fields after memo field>
   ENDSCAN
ENDPRINTJOB

PROCEDURE Box_memo
   DEFINE BOX FROM 14 TO (14 + mem_width + 2) HEIGHT MEMLINES(Notes) + 2
   _box = .T.
   ?
   ?? Notes AT 15
   ?
   _box = .F.
RETURN
```

Group Breaks

Many reports are grouped on a field or expression. Emp_rept, for example, is grouped on department. The program prints a department name followed by data for that department and department summary information. Then the cycle starts over with the next department.

The following excerpt from the main report program shown earlier forces the group breaks. As long as *group_brk* contains the current department, the IF condition fails and the routine skips to the ENDIF. The Detail procedure prints the current record and Det_calc increments the values of the variables holding summary data (see the Report Detail and Summary Data sections). Because the code is nested in a SCAN loop, all records for that department print.

When the department changes, however, the IF condition becomes true. Now Brk_data prints summary information for the department that has just finished printing (see the Summary Data section). Reinit reinitializes the summary variables and Brk_head prints the group heading for the *next* department (see below). Then the records for that department print.

```
SCAN WHILE print_flag
   IF group_brk <> Department
      rpt_level = 1
      DO Brk_data
      DO Reinit
      DO Brk_head
   ELSE
      rpt_level = 0
   ENDIF
   DO Detail
   DO Det_calc
ENDSCAN
```

Below is the code for Brk_head, the procedure that prints the group headings. It starts with an IF construct to check whether enough lines remain on the page to begin a new group. This construct adds eight lines to the current line number (_plineno) to allow for the department name and two two-line records, with blank lines in between. If the resulting value is larger than *on_pg_line,* the page ejects.

Then the procedure prints the group heading shown in Figure 13-1 earlier in the chapter. The STYLE "BU" clause prints the heading underlined and in boldface type. Note that the underlining does not extend to the colon (:) character. If a group continues on to a following page, the final IF...ENDIF construct prints the string *(cont'd)* after the group heading at the top of the page.

```
PROCEDURE Brk_head
   group_brk = Department
   IF _plineno + 8 > on_pg_line
      EJECT PAGE
   ELSE
      ?
      ?  "Department" STYLE "BU" AT 0, ": " STYLE "B"
      ?? Department PICTURE "@T XXXXXXXXXXXXXXX"
      IF rpt_level = 0
         ?? "(Cont'd)"
      ENDIF
      ?
      rpt_level = 0
   ENDIF
RETURN
```

The rest of this section shows different ways to group data output.

Variation #1 — Alphabetical Grouping

If your program prints a list of names, it should group the list alphabetically. Use the LEFT() function to identify the first letter of each last name, and print a new group heading each time the first letter changes.

```
PROCEDURE Alfagrup
   USE Names ORDER Name
   SCAN
      STORE LEFT(Lastname,1) TO first_ltr
      ? "Last names beginning with " + first_ltr + ":"
      ?
      SCAN WHILE LEFT(Lastname,1) = first_ltr
         ? Lastname, Firstname, Phone
      ENDSCAN
      ?
   ENDSCAN
RETURN
```

Variation #2 — Numeric Grouping

Your program can also group data in numeric increments. This example shows the data in a credit file grouped according to how much the borrower owes. It breaks the listing in $500 increments.

```
PROCEDURE Num_grup
   USE Credit ORDER Amt_due
   m->amt_due = 0
   SCAN
      m->amt_due = m->amt_due + 500
      ? "Owes up to $", m->amt_due PICTURE "9,999,999.99"
      SCAN WHILE Amt_due <= m->amt_due
         ? Lastname, Firstname, Amt_due
      ENDSCAN
      ?
   ENDSCAN
RETURN
```

Variation #3 — Grouping by Identification Number

You might want to group data according to a unique identification number. This example groups inventory data by vendor number. For each group, it prints the vendor number and information on that vendor's inventory.

```
SCAN
   vend_id = Vendor_id
   ? "Vendor No:" AT 0 STYLE "B", vend_id
   ?
   SCAN WHILE vend_id = Vendor_id
      inv_cost = ROUND(Qty_onhand * Cost,2)
      ?  Part_id AT 5
      ?? Part_name AT 18
      ?? Qty_onhand PICTURE "9999" AT 50
      ?? Cost PICTURE "99,999.99" AT 58
      ?? inv_cost PICTURE "99,999.99" AT 68
   ENDSCAN
   ?
ENDSCAN
```

Variation #4 — Two-level Grouping

Two-level grouping groups data within other groups. The following code groups first by department and then by title. Department is the *major group* and title is the *minor group.*

```
USE Employee
INDEX ON Department + Title TO Depttitl
SET SPACE ON
GO TOP
SCAN
   mdept = Department
   ? "Department:" AT 0, mdept
   SCAN WHILE Department = mdept
      mtitle = Title
      ? "Title:" AT 5, mtitle
      SCAN WHILE Title = mtitle
         ? Firstname FUNCTION "T" AT 10, Lastname
      ENDSCAN
      ?
   ENDSCAN
   ?
ENDSCAN
USE
DELETE FILE Depttitl.ndx
RETURN
```

Summary Data

Summary information in a report can consist of group subtotals, column totals, a grand total, averages, and other kinds of calculated information. Emp_rept calculates summary information for each department and an overall salary total for the facility. The procedures shown here are all called from the main report program.

As each employee record prints, Det_calc updates the variables holding summary information. When the records for a department have finished printing, Brk_data prints the summary data that has been stored in these variables.

Brk_data also keeps track of salaries for the whole facility by incrementing *salary_sum* after each department prints. Thus, *salary_gt* contains a running total of department salaries. When the report has finished printing, Rpt_end prints this facility total.

```
PROCEDURE Det_calc
   number_emp = number_emp + 1
   salary_sum = salary_sum + salary
   yrsexp_sum = yrsexp_sum + yrs_exper
   yrsexp_avg = ROUND(yrsexp_sum/number_emp,1)
RETURN

PROCEDURE Brk_data
   ?
   ?? "HEADCOUNT:" STYLE "B" AT 0
   ?? number_emp PICTURE "999"
   ?? "TOTAL:" STYLE "B" AT 36
   ?? "  $" AT 44, salary_sum PICTURE "999,999.99"
   ?? " ", "AVG:" STYLE "B" AT 59
   ??  yrsexp_avg PICTURE "99.9", " yrs"
   ?  "AVG:" STYLE "B" AT 36
   ?? "  $" AT 44, ROUND(salary_sum/number_emp,2) PICTURE "999,999.99"
   ?
   salary_gt = salary_gt + salary_sum
RETURN

PROCEDURE Rpt_end
   ?
   ?? "FACILITY TOTAL:" STYLE "B" AT 27
   ?? "  $" AT 42, salary_gt PICTURE "9,999,999.99"
RETURN
```

Environmental Cleanup

A report program should restore the environment to its original state before terminating. If the report program is only executed from within a larger application, that application can restore most settings. The report program need only restore settings unique to it.

The following cleanup code is located at the end of the main program in Emp_rept. Compare it with the environmental setup code shown previously.

```
SET PRINTER OFF
ON ESCAPE
ON PAGE
SET CENTURY OFF
SET ORDER TO TAG Names

_ppitch = mppitch
_wrap = mwrap
_peject = mpeject
_plength = mplength
_ploffset = mploffset
_lmargin = mlmargin
_rmargin = mrmargin
```

Other Types of Reports

You can program reports other than tabular listings in dBASE IV. This section shows how to code form letters and mailing labels.

Both form letters and mailing labels consist of variable data against a background of fixed text. A form letter inserts variable information such as the name and address of the recipient into the text of the letter. In mailing labels, variable name and address fields are inserted between fixed punctuation and spacing.

Form Letters

Using dBASE IV, you can program form letters such as the one shown in Figure 13-3. The variable data in this form letter includes the mailing address, date, account number, and addressee's name in the salutation, and the payment amount, payment date, account balance, amount of recent orders, and due date for the next payment. All other text is constant from letter to letter.

TOM SMITHER
664 LONG STREET
APT. 33
Santa Monica, CA 97090

November 16, 1988

RE: Account No. B12345

Dear Mr. TOM SMITHER:

ACME MANUFACTURING COMPANY has enjoyed an illustrious history of manufacturing fine furniture for customers who appreciate the value that only solid wood and hand finishing can provide. We would like to thank you for your prompt payment history, which includes your recent payment of $11,350.00 on 05/03/1988.

Our records now show a balance on your account of $1,000.00. Your next invoice will reflect this balance plus your orders for this month in the amount of $400.00. Your next payment is due by 12/16/1988.

Sincerely yours,

Howard J. Spurling
Credit Manager

Figure 13-3 Accounting form letter

On the following page is the code for this form letter. First, the routine opens and relates the necessary files. Then it establishes the desired environment with a series of SET commands. The command SEEK .01 finds the first customer owing a previous balance of as little as one cent. (SET NEAR ON causes the SEEK to stop at the balance no larger than this value.)

The heart of the procedure is a SCAN loop that prints the form letter for all customers with an outstanding balance. Variable fields are printed with ?/??, text with TEXT...ENDTEXT. (The TEXT...ENDTEXT construct outputs text exactly as you enter it into your program code.) The routine uses FUNCTION "T" to trim the City and Contact (customer name) fields and the "$" PICTURE function to output a dollar sign before the amount.

```
SELECT 1
USE Acct_rec ORDER Oldbalance
USE Cust ORDER Cust_id IN 2
SET RELATION TO Cust_id INTO Cust
SET SPACE OFF
SET CENTURY ON
SET CONSOLE OFF
SET PRINTER ON
SET NEAR ON
SEEK .01
SCAN WHILE Oldbalance > 0
   ? Cust->Customer
   ? Cust->Address1
   IF "" <> Cust->Address2
      ? Cust->Address2
   ENDIF
   ? Cust->City FUNCTION "T", ", ", Cust->State, "  ", Cust->Zip
   ?
   ? MDY(DATE())
   ?
   ? "RE: Account No. ", Cust_id
   ?
   ? "Dear ", Cust->Contact FUNCTION "T", ":"
   ?
   TEXT
ACME MANUFACTURING COMPANY has enjoyed an illustrious history of
manufacturing fine furniture for customers who appreciate the
value that only solid wood and hand finishing can provide. We
would like to thank you for your prompt payment history, which
includes your recent payment of
   ENDTEXT
   ?? Amt_lst_pd PICTURE "@$ 999,999.99", " on ", Dat_lstbil, "."
   ?
   ?  "Our records now show a balance on your account of "
   ?? LTRIM (TRANSFORM(Oldbalance, "@$ 999,999.99")), "."
   ?  "Your next invoice will reflect this balance plus your orders for "
   ?  "this month in the amount of "
   ?? LTRIM (TRANSFORM(Amt_of_cur, "@$ 999,999.99")), "."
   ?? Amt_of_cur PICTURE "@$ 999,999.99", ". "
   ?? "Your next payment is"
   ?  "due by "
   ??  DATE()+30, "."
   ?  "Sincerely yours,"
   ?
   ?  "Howard J. Spurling"
   ?  "Credit Manager"
   EJECT PAGE
ENDSCAN
SET CENTURY OFF
SET CONSOLE ON
SET PRINTER OFF
SET NEAR OFF
```

Mailing Labels

A mailing labels report positions name, address, city, state, and zip code fields on sheets of special label paper (see Figure 13-4). You can create labels in the menu system. The command CREATE/MODIFY LABEL accesses the labels design screen from the dot prompt.

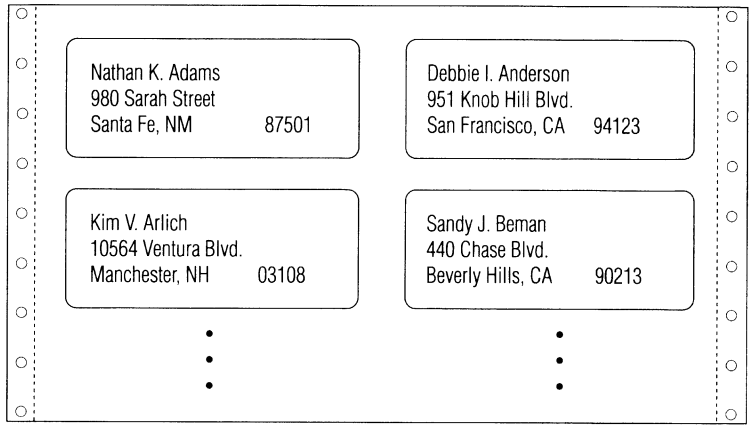


Figure 13-4 Sheet of mailing labels

Following is the routine for the labels shown in the figure. First, it DECLAREs an array to hold data for each of the seven fields in the label. Then it opens a print job and starts processing records. It loads the array with the fields from the first record and advances the record pointer. If more records are found, it sets the variable *isfound* to .T..

Then the routine prints the two columns of labels line by line in zig-zag fashion — the left name, the right name, the left address, the right address, and so on through each row of each label. It gets the data for the left-hand labels from the array and the data for the right-hand labels from the database file. The "@T" PICTURE function trims the Firstname and City fields, so that no gaps appear in the label.

Using an array makes this program run more efficiently because the record pointer needn't jump back and forth between records to get data.

```
_wrap = .F.
DECLARE lbl_array[1,7]
isfound = .F.

PRINTJOB
   DO WHILE FOUND() .AND. .NOT. EOF()
      isfound = .F.
      lbl_array[1,1] = Firstname
      lbl_array[1,2] = Initial
      lbl_array[1,3] = Lastname
      lbl_array[1,4] = Address1
      lbl_array[1,5] = City
      lbl_array[1,6] = State
      lbl_array[1,7] = Zip
      CONTINUE
      IF FOUND() .AND. .NOT. EOF()
         isfound = .T.
      ENDIF

      *- Column 1 first name, initial, last name
      ?? lbl_array[1,1] PICTURE "@T XXXXXXXXXX" AT 0, " ",;
         lbl_array[1,2] PICTURE "X", ". ",;
         lbl_array[1,3] PICTURE "XXXXXXXXXXXXXXX",
      *- Column 2 first name, initial, last name
      IF isfound
         ?? Firstname PICTURE "@T XXXXXXXXXX" AT 35, " ",;
            Initial   PICTURE "X", ". ",;
            Lastname  PICTURE "XXXXXXXXXXXXXXX"
      ENDIF
      ?
      *- Column 1 address
      ?? lbl_array[1,4] PICTURE "XXXXXXXXXXXXXXXXXXX" AT 0,
      *- Column 2 address
      IF isfound
         ?? Address1 PICTURE "XXXXXXXXXXXXXXXXXXX" AT 35
      ENDIF
      ?
      *- Column 1 city, state, zip
      ?? lbl_array[1,5] PICTURE "@T XXXXXXXXXXXXX" AT 0, ", ",;
         lbl_array[1,6] PICTURE "XX",;
         lbl_array[1,7] PICTURE "XXXXXXXXX" AT 25,
      *- Column 2 city, state, zip
      IF isfound
         ?? City  PICTURE "@T XXXXXXXXXXXXX" AT 35, ", ",;
            State PICTURE "XX",;
            Zip   PICTURE "XXXXXXXXX" AT 60
      ENDIF
      ?
      ?
      ?
      ?
      CONTINUE
   ENDDO
ENDPRINTJOB
```

Running Formatted Reports

Two commands output dBASE IV formatted reports. REPORT FORM runs reports created in the menu system, while DO runs custom report programs that you code yourself. You can use the dBASE IV menu-building commands to provide users a print menu, so they can select print attributes when they run a report.

Printing Reports

Use REPORT FORM to run reports created in the menu system. The following command prints an accounting report showing all June records. The NOEJECT option prevents formfeeds before and after the printing. It is comparable to *_peject = "NONE"*. To send the report to the screen rather than the printer, omit the TO PRINTER option and SET PRINTER OFF.

```
REPORT FORM Acct_rpt FOR MONTH(Date_order) = 6 NOEJECT TO PRINTER
```

Use the DO command to run custom reports like the ones shown in this chapter. Print settings such as pitch or page length are defined in the report code using system memory variables (see the Environmental Setup and Page Layout sections).

Print Menus

You can let users select print attributes from a print menu. Figure 13-5 shows the print menu used in all the Business application reports.

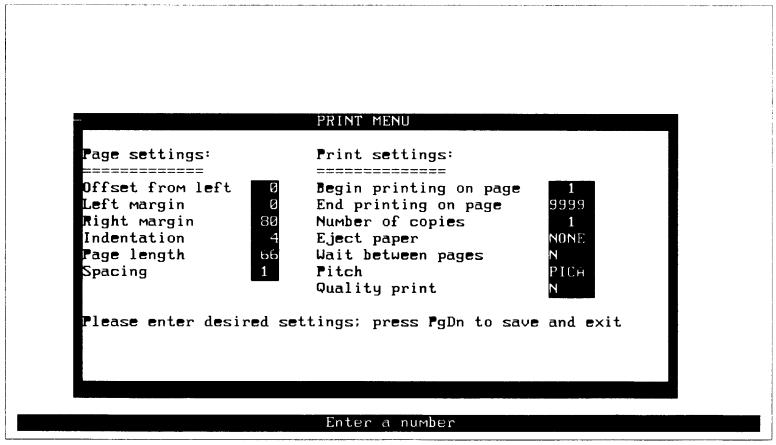
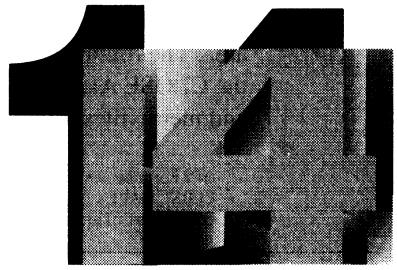


Figure 13-5 The Business print menu

Below is the procedure for the print menu shown in Figure 13-5. First it defines a set of variables to hold the user's print selections. Then it displays the print menu in a window. This menu is constructed with @...SAY...GETs, rather than the dBASE IV menu-building commands, so that users can enter a selection for each option. Finally, the procedure sets the printer system variables to the values entered by the user.

```
PROCEDURE Prt_menu
   msg_num = "Enter a number"
   msg_logic = "Enter Y or N"
   msg_enum = "Press spacebar for other options"
   loffset = 0
   lmargin = 0
   rmargin = 80
   indent = 4
   plength = 66
   STORE 1 TO pspacing, pbpage, pcopies
   pepage = 9999
   peject = "NONE  "
   STORE .F. TO pwait, pquality
   ppitch = "PICA     "
   ACTIVATE WINDOW lister
   CLEAR
   @  0, 0 SAY "---------- PRINT MENU ----------";
          COLOR &c_red.
   @  2, 1 SAY "Page settings:"
   @  3, 1 SAY "=============="
   @  4, 1 SAY "Offset from left      " GET loffset  PICTURE "99" MESSAGE msg_num
   @  5, 1 SAY "Left margin           " GET lmargin  PICTURE "99" MESSAGE msg_num
   @  6, 1 SAY "Right margin          " GET rmargin  PICTURE "99" MESSAGE msg_num
   @  7, 1 SAY "Indentation           " GET indent   PICTURE "99" MESSAGE msg_num
   @  8, 1 SAY "Page length           " GET plength  PICTURE "99" MESSAGE msg_num
   @  9, 1 SAY "Spacing               " GET pspacing PICTURE "9"  RANGE 1,3;
              MESSAGE msg_num
   @  2,26 SAY "Print settings:"
   @  3,26 SAY "==============="
   @  4,26 SAY "Begin printing on page" GET pbpage  PICTURE "999";
    MESSAGE msg_num
   @  5,26 SAY "End printing on page  " GET pepage  PICTURE "9999";
    MESSAGE msg_num
   @  6,26 SAY "Number of copies      " GET pcopies PICTURE "999";
    MESSAGE msg_num
   @  7,26 SAY "Eject paper           " GET peject;
          PICTURE "@M BEFORE, AFTER, BOTH, NONE" MESSAGE msg_enum
   @  8,26 SAY "Wait between pages    " GET pwait PICTURE "Y" MESSAGE msg_logic
   @  9,26 SAY "Pitch                 " GET ppitch;
          PICTURE "@M DEFAULT, PICA, ELITE, CONDENSED" MESSAGE msg_enum
   @ 10,26 SAY "Quality print         " GET pquality PICTURE "Y";
    MESSAGE msg_logic
   @ 12, 1 SAY "Please enter desired settings; press PgDn to save and exit"
   READ
   DEACTIVATE WINDOW lister
   _ploffset = loffset
   _lmargin  = lmargin
   _rmargin  = rmargin
   _indent   = indent
   _plength  = plength
   _pspacing = pspacing
   _pbpage   = pbpage
   _pepage   = pepage
   _pcopies  = pcopies
   _peject   = peject
   _pwait    = pwait
   _ppitch   = ppitch
   _pquality = pquality
   SET COLOR TO &c_standard.
RETURN
```

Data Security and Integrity



Users of your applications should have confidence that their data is secure against loss or corruption, and that it is in a consistent state throughout the application. This chapter reviews some techniques for handling disk files from within a program, and then shows how your programs can ensure the security and integrity of the data they handle.

What This Chapter Covers

This chapter discusses the following topics:

- Handling disk files
- Backing up data
- Managing data transactions
- Protecting data

The last two topics are only summarized here, because they are covered in detail in other manuals. See BEGIN/END TRANSACTION in Chapter 2 of *Language Reference* for information on managing data transactions. Chapter 14 of *Using dBASE IV* discusses protecting confidential files.

Handling Disk Files

You can perform many disk file-handling operations from within a dBASE IV program. This section reviews how a program can identify and find files, delete files, import and export files, and determine file size and available disk space.

Identifying a Database or Index File

The DBF() function returns the name of the database file in USE, if any. The MDX() and NDX() functions return the name of an open index file (see Chapter 9). To identify the open file in an unselected work area, include the alias with these functions. They return a null string if no file is in USE in the specified work area. If you have SET FULLPATH ON, these functions return the full path along with the filename.

The following short routine closes all files except the database file in the current work area. The routine first stores the name of the current database file to *mfilename*. Then the CLOSE ALL command closes all open database files and associated index, format, and memo files. Finally, the routine reopens the database file noted in *mfilename*.

```
mfilename = DBF()
CLOSE ALL
USE (mfilename)
```

Finding Files

Usually an application accesses files directly without the user's knowledge. Sometimes, however, a program must request a filename from the user. In that case, use the FILES option of DEFINE POPUP to display a file directory.

The following module displays a list of database files. First, DEFINE POPUP defines the file list. Note that the *LIKE *.dbf* clause limits the display to database files. Then the routine ACTIVATEs the list. If the user presses ↵ without selecting a file, a message appears and the routine ends. Otherwise, the selected file appears on the status line.

```
DEFINE POPUP filespop FROM 4,30 PROMPT FILES LIKE *.dbf;
                      MESSAGE "Press <Enter> to select a file"
ON SELECTION POPUP filespop DEACTIVATE POPUP
ACTIVATE POPUP filespop
mfile = PROMPT()
IF "" = mfile
   @ 21,30 SAY "No file selected"
   WAIT "Press any key to return to Main Menu"
   RETURN
ENDIF
USE (mfile)
RELEASE POPUP filespop
```

You can use a similar routine to locate files of a different file type. Just change the file skeleton in the DEFINE POPUP statement and replace the USE command with the appropriate command.

Deleting Files

Usually an application deletes files automatically. If you want your program to prompt users for files to delete, define a file list as shown in the previous section. Use the same code, but replace the USE command with:

```
ERASE (mfile)
```

If you want to use wildcards to delete groups of similarly named files, use the DOS ERASE command instead of the dBASE ERASE command:

```
RUN ERASE *.txt
```

Importing and Exporting Files

Complex applications typically store data in a specific format that is foreign to other applications. For example, dBASE IV stores information about a database file in the database file header.

Your programs might have to import files that are in another file format, or export files to another format.

dBASE IV can import from and export to the following file formats:

- Lotus (WKS and WK1)
- Framework II®, Framework III®, and Framework IV™ (.FW2, .FW3, and .FW4)
- RapidFile
- PFS
- dBASE II®

To import a file that is not one of the supported file formats, you must convert the data into a standard data exchange format such as SDF or delimited ASCII. Use the export utilities of the original application to create the ASCII or SDF file. Then import the data into dBASE IV using APPEND FROM with the DELIMITED WITH or SDF option.

See *Language Reference* for more information on IMPORT, EXPORT, APPEND FROM, and COPY TO.

NOTE
dBASE IV can directly read and write to dBASE III PLUS .dbf files. However, once you change data in a memo field, the file is no longer downwardly compatible with dBASE III PLUS. While in dBASE IV, you must COPY the modified file TO a new file using the DBMEMO3 option. The new file will then be accessible to dBASE III PLUS.

Determining File Size and Disk Space

Sometimes the action taken by your program depends upon the size of the database file and the available space on disk. For example, your program should check for sufficient disk space before SORTing or COPYing a database file.

This section provides two user-defined functions (UDFs) that determine the size of database and memo files. Then it shows how to use the information in a routine that determines available disk space.

Computing Database (.dbf) File Size

The following UDF computes the total size of a database (.dbf) file. First it opens the file passed to it from the calling module (see the Determining Available Disk Space section). Then it computes the total record volume by multiplying RECCOUNT() by RECSIZE().

Next, the routine uses a DO WHILE loop to count the fields in the database file. Each pass through the loop increments *field_no* by one. Then the FIELD() function in the WHILE condition returns the name of the field denoted by that number. When there are no more fields, *FIELD(field_no + 1)* returns a null string and the loop ends. The value of *field_no,* which is the number of fields in the file, is assigned to *numfields.*

The last part of the routine uses a formula to compute the size of the database file header. (dBASE IV maintains a *file header* that keeps track of field names, lengths, and types.) Then it calculates the total size of the database file. This consists of the record volume and the header size, plus one byte for an end-of-file marker.

```
FUNCTION  Dbf_size
    PARAMETERS mfile
    USE (mfile)
    rec_vol = RECCOUNT() * RECSIZE()

    field_no = 0
    DO WHILE "" < FIELD(field_no + 1)
       field_no = field_no + 1
    ENDDO
    numfields = field_no

    header = (32 * numfields) + 34
    dbf_size = rec_vol + header + 1
RETURN  dbf_size
```

Computing Memo (.dbt) File Size

The UDF shown here obtains the size of a memo (.dbt) file from the operating system directory listing. (You can use this same method to determine the size of any type of file.) The first part of this UDF creates an empty database file. It opens the database file Client and copies its structure to Temp. (COPY STRUCTURE EXTENDED creates a database file with the fields Field_name, Field_type, Field_len, and Field_dec.) Then it ZAPs the contents of Temp, APPENDs a BLANK record, and REPLACEs its fields with the field that will hold the DIR listing. It closes Temp, CREATEs Dir_text FROM the structure defined in Temp, and opens the newly created database file.

The second part of the module gets the name of the memo file from the parameter *mfile.* It initializes *mfilename* with the memo file specification (for example, *\DBASE\SAMPLES\CLIENT.DBF)* and extracts the right 12 characters of *mfilename.* The DO WHILE...ENDDO then removes any extraneous characters (here, trimming *S\CLIENT.DBF to CLIENT.DBF).* Finally, the routine changes the last character to a T (here, producing *CLIENT.DBT).*

The last part of the UDF RUNs the DIR command for *mfilename,* redirecting the output to Dir.txt. Then it APPENDs the contents of Dir.txt into Dir_text, the file created earlier. Each record in Dir_text now contains one line of the DIR listing. The routine LOCATEs the record containing data on the .dbt file and extracts the size of the text file. It uses the VAL() function to convert the string to a number.

```
FUNCTION Dbt_size
   PARAMETERS mfile
   USE Client
   COPY TO Temp STRUCTURE EXTENDED
   USE Temp
   ZAP
   APPEND BLANK
   REPLACE Field_name WITH "Dir_line", Field_type WITH "C", ;
      Field_len WITH 80
   USE
   CREATE Dir_text FROM Temp
   USE Dir_text

   mfilename = RIGHT(mfile,12)
   DO WHILE "\" $ mfilename
      mfilename = RIGHT(mfilename, LEN(mfilename) - 1)
   ENDDO
   mfilename = UPPER(LEFT(mfilename, LEN(mfilename) - 1) + "T")
   var = "RUN DIR" + mfilename + "> Dir.txt"
   &var
   APPEND FROM Dir.txt TYPE SDF
   GO 5
   LOCATE FOR SUBSTR(Dir_line, 10, 3) = "DBT"
   dbt_size = VAL(SUBSTR(Dir_line,13,9))
RETURN dbt_size
```

Determining Available Disk Space

Once your program has computed the total size of a database file and its associated memo file (see previous two sections), it can determine whether there is sufficient space for a disk operation. The following routine prompts users when there is not enough space on disk to COPY a database file:

```
DO WHILE .T.
   IF DISKSPACE() < (DBF_SIZE(mfile) + DBT_SIZE(mfile)) * 2
      ?? CHR(7)
      choice = "N"
      @ 10,10 SAY "Not enough room on disk to copy this file"
      @ 11,10 SAY "Delete some files and try again? (Y/N)";
              GET choice PICTURE "!" VALID choice $ "YN"
      READ
      IF choice = "Y"
         DO Delefile
      ELSE
         EXIT
      ENDIF
   ELSE
      COPY TO Temp
      EXIT
   ENDIF
ENDDO
```

You can also use the UDFs shown above to determine if there is enough disk space for a file backup (see the Backup Menus section).

Backing Up Data

Backing up data is a necessary precaution in the event of catastrophic data loss, such as a power failure or disk crash. Don't assume that users of your applications will back up database files from their operating system. Your applications should back up data automatically or provide a Backup menu option.

This section explains how to use both dBASE IV and external commands in record and file backup routines. You can modify the routines shown here for use in a variety of applications.

Saving Data to Disk

Whenever your program APPENDs, REPLACEs, or otherwise enters information into a database file, that information is written to disk. However, to maintain maximum performance, dBASE IV only updates the disk file and directory periodically. dBASE IV does a more complete disk update when your program performs a USE, CLOSE DATABASES, or QUIT.

If you want dBASE IV to update the file at other times, SET AUTOSAVE ON. This command instructs dBASE IV to update the disk file and directory after a record update. The update operation occurs after each record update for commands that process one record at a time, such as APPEND, BROWSE, and EDIT. For commands that process groups of records, like REPLACE, SORT, and TOTAL, dBASE IV performs the disk update after the processing is complete.

SET AUTOSAVE ON does result in somewhat slower performance. Therefore, your program should SET AUTOSAVE ON only before I/O intensive operations, such as REPLACE, and then SET AUTOSAVE OFF when the operation is complete.

Copying Files

dBASE IV provides commands for copying both open and closed files. In addition, you can RUN external file copy commands from within a dBASE program.

Copying an Open File

You can COPY data from an open database file to a new file in dBASE IV. The following routine uses the STRUCTURE option of DEFINE POPUP to define a field list for user selection. The DO WHILE loop builds a list of the fields selected by the user. For each selection, it adds the field name followed by a comma to the list. After the last selection, it strips the final comma and performs a COPY FIELDS using the defined field list.

```
filds = ""
DEFINE POPUP fildspop FROM 4,24 TO 14,62 PROMPT STRUCTURE;
       MESSAGE "Press <Enter> to select fields, <Esc> when done"
ON SELECTION POPUP fildspop DEACTIVATE POPUP
DO WHILE .T.
   ACTIVATE POPUP fildspop
   IF "" = PROMPT()
      EXIT
   ENDIF
   IF .NOT. (PROMPT() $ filds)
      filds = filds + TRIM(PROMPT()) + ","
   ENDIF
   @ 18,0 SAY "Fields selected:"
   @ 19,0 SAY filds
ENDDO
RELEASE POPUP fildspop
IF " " <> filds
   filds = LEFT(filds, LEN(filds) - 1)
   COPY FIELDS &filds. TO Newfile
ELSE
   @ 21,30 SAY "No fields selected"
   WAIT "Press any key to return to Main Menu"
   RETURN
ENDIF
```

Copying a Closed File

You can back up closed files using the COPY FILE command. You can also copy and back up files at the operating system level.

Backup Menus

You should recommend a consistent backup routine to your clients. Users should back up very important data daily to another disk (not just to another directory of the same disk). They should also make weekly archival copies of all database files. For less important data, have users back up data every two or three days and archive the data every two weeks.

Your application should provide information to encourage users to back up their work using available backup utilities.

```
DO M_popdef
ACTIVATE POPUP backmenu

PROCEDURE M_popdef
   DEFINE POPUP backmenu FROM 7,20 TO 12,59;
     MESSAGE;
     "Press first letter of menu choice, or highlight then press <Enter>"
   DEFINE BAR 1 OF backmenu PROMPT "======== BACKUP - RESTORE MENU ========" SKIP
   DEFINE BAR 2 OF backmenu PROMPT " Back up your data to floppy disks"
   DEFINE BAR 3 OF backmenu PROMPT " Restore your data from floppy disks"
   DEFINE BAR 4 OF backmenu PROMPT " Exit to the Main Menu"
   ON SELECTION POPUP backmenu DO Back
RETURN

PROCEDURE Back
   DO CASE
      CASE BAR() = 2
         DO Sure
         IF choice = "Y"
            DO Bbackup
         ENDIF
      CASE BAR() = 3
         DO Sure
         IF choice = "Y"
            DO Rrestore
         ENDIF
      CASE BAR() = 4
         RETURN TO MASTER
   ENDCASE
RETURN
```

Assuming your program contains UDFs like DBF_SIZE() and DBT_SIZE() (see the Determining File Size and Disk Space section earlier in this chapter), it can perform a COPY if all the data fits on one disk, and a BACKUP if more than one disk is required:

```
SET DIRECTORY TO DISK2
IF DISKSPACE() > DBF_SIZE(dbfsize) + DBT_SIZE(dbt_size)
   DO Sys_copy
ELSE
   DO Bbackup
ENDIF
```

Managing Data Transactions

File backups let you re-create data from earlier versions. dBASE IV also provides the capability to protect data in the course of a transaction. *Transaction management* is a system for ensuring the integrity of data during operations that add or change records, or mark them for deletion.

dBASE IV provides the construct BEGIN...END TRANSACTION for managing data

transactions. When dBASE IV encounters BEGIN TRANSACTION, it creates a *transaction log file* that records changes to the database file. (In a single-user environment, this file is called Translog.log.) In the event an error occurs, dBASE IV can use this file to restore all database files to their state when the BEGIN TRANSACTION command was issued.

dBASE IV also puts an integrity flag in the database file header indicating that a transaction is in progress and the file is in an inconsistent state. dBASE IV uses this flag in the event of a transaction failure to prevent editing of an inaccurate file.

You can open some files with the USE NOLOG option during a transaction. Such files do not become part of the transaction log and can be closed during a transaction.

You can use this option to open files and look up data, or to create temporary files while a transaction is in progress.

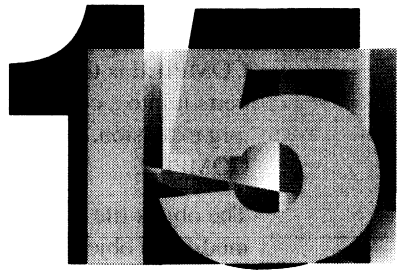
Protecting Data

While hardware failure is a genuine hazard to data, human mishandling of data can be equally disastrous. To keep such problems to a minimum, limit the accessibility of the data to those individuals who need to work with it.

dBASE IV provides an internal utility called PROTECT that lets you create and maintain security in your applications. PROTECT makes a database system more secure by:

- Preventing unauthorized users from logging onto the system
- Controlling which files and fields each user may access
- Encrypting the data in database files

Compiling, Debugging, and Linking



The COMPILE command of dBASE IV produces object code files in binary format from program source files. Object files contain an executable form of the program.

The executable file contains expressions that have been evaluated and saved as a value, and macro (&) operators that have been expanded. The object file can execute rapidly without loading the dBASE macro overlay.

The DBLINK program can link several compiled programs or procedures into a single executable file to further improve execution times.

dBASE IV has a full-screen debugger to help you find and correct any program errors.

What This Chapter Covers

- Compiling
- Accessing the debugger
- The debugger windows
- The debugger commands
- A sample debugging session
- Linking your program modules

Compiling

COMPILE is the dBASE command that reads a dBASE program source file and converts it into a compiled object file. The source code file is in ASCII, and defaults to a .prg extension. You can use another file extension by specifying it with the filename to COMPILE.

The object file is binary, and COMPILE always gives it a .dbo extension. You cannot modify an object file; you must change the source program file and recompile it.

Compile and File Search Path

dBASE IV searches for program files to be compiled using a search order that involves an interaction among SET DIRECTORY and SET PATH settings and the directory you were in when you started dBASE IV.

If you use unambiguous filenames for your program files, dBASE IV bypasses the search order and finds the files directly. An unambiguous filename is a filename that includes the drive letter and the full directory path for that file.

For all ambiguous filenames, the dBASE IV search path for program files follows this order:

- The current drive and directory
- SET DIRECTORY/SET DEFAULT setting
- SET PATH setting

At start up, the current dBASE directory is the current DOS drive or directory. The current directory can be the directory from which you started dBASE IV, or the directory to which you changed using a SET DIRECTORY or SET DEFAULT command. The SET DIRECTORY command can be issued from the Config.db file, in a start-up program, or from the dot prompt.

If dBASE IV cannot locate the program files in the current drive and directory, it will look in the dBASE SET PATH directories.

To make sure that your program files and their matching .dbo files are kept together, SET DIRECTORY to your current working directory. Setting a default directory also saves you from typing the full file specification including the drive and the path for program files.

If you have different versions of your program with the same filenames, you must keep them in separate directories.

You must have different names for .prg and .prs files that are in the current directory. COMPILE will assign the .dbo extension to the compiled versions of all .prg and .prs files. Different versions of program files with the same name will write over each other's .dbo files unless you use a version-specific naming system, or you redirect the output files to separate directories.

Expression Optimization

dBASE IV optimizes expressions during compilation. If you use constants in the source code, the compiler computes and saves the value for the expression in the object code. For example, x = 1 + 3 + 4 is optimized and saved as x = 8.

Comparing two string constants in an expression could cause a problem. For example, "abcde" = "abcd" evaluates to true if SET EXACT is OFF, but evaluates to false if SET EXACT is ON. If you SET EXACT ON during execution of code, the expression will still be false because it was optimized and stored as a logical false during compilation.

Compile-Time Error Messages

If there are any syntax errors, program flow control errors, or logic errors, COMPILE sends error messages to the screen when SET TALK is ON.

If there are any programming errors, COMPILE will terminate without producing a .dbo file.

Compiler Directives

Compiler directives support conditional compilation. These directives can be used only in programs and not from the dot prompt.

Conditional compilation is a method for including some modules of a program and excluding other modules from the compiled object file. For example, if you write an application that will run on both UNIX and DOS, you would have two separate procedures (two sets of code) for defining the main menu. You could keep one central .prg file that contains both procedures. At compile time, by defining an operating system directive, you could compile for UNIX and leave out the DOS specific procedures.

The compiler directives are:

#define <name>

#undef <name>

#ifdef <name>

#ifndef <name>

#else

#endif

#ifdef, #ifndef, #else, and #endif make up a program flow construct. These directives must be used together to control the branching of conditional compilation.

Commonly used compiler directives of other programming languages produce warning messages. These are:

#if

#elif

#pragma

#include

Menus.prg would contain the following lines:

```
#define unixver
COMPILE Menus.prg
```

An example of conditional compilation follows:

```
#ifdef unixver
    DO Menu1
    .
    .
    .
#else
    DO Menu2
    .
    .
    .
#endif
```

Conditional compilation variable names must be character strings (A to Z or a to z) of up to 32 characters. They cannot be evaluated expressions.

Compiler directives can be nested up to 32 levels. A compiler directive defined in a lower level procedure is local.

Procedure Libraries

dBASE IV versions 1.5 and higher support system level procedures from special procedure library files. These libraries are options you can specify in the Config.db file, and dBASE IV searches through them in a specific order for procedures you use in a program file. The search order for a procedure in dBASE IV versions 1.5 and higher is as follows:

1. SYSPROC=Filename if a file is defined as SYSPROC in the Config.db file.
2. The .dbo file currently being executed where a procedure has been called.
3. SET PROCEDURE to file if one is active.
4. The other .dbo files in the calling chain of programs beginning with the most recently opened .dbo file.

5. SET LIBRARY file if one is active.
6. Search the disk for a .dbo file with the same name and open it.
7. Search the disk for a .prg file with the same name and compile it.
8. Search the disk for a .prs file with the same name and compile it.

If you add a line to your Config.db file that is similar to the following:

```
SYSPROC=MYPROCS.DBO
```

the Myprocs.dbo file and the procedures and UDFs it contains will take precedence over everything except native dBASE IV commands.

Similarly, a line in Config.db such as the following:

```
PROCEDURE=Proc_abc.dbo
```

will replace procedures that are defined in a SET LIBRARY command.

And, a setting in Config.db like the following:

```
LIBRARY=Mylib.dbo
```

will define a core set of procedures not found in the SYSPROC library or the PROCEDURE file.

You can use the SET() function with these two keywords to find out the names of the currently active procedure libraries. The syntax is as follows:

```
y = SET("PROCEDURE")
z = SET("LIBRARY")
```

The SET() function returns a null string if a library has not been specified.

You can establish a library of core procedures that are compiled and available to many program modules, and also use SET PROCEDURE locally to override the SET LIBRARY file.

If you include a SET LIBRARY TO <library name> in your program, make sure the library file is available to the various program modules by putting it in the directory specified with SET DIRECTORY TO.

Libraries set from Config.db can be turned off inside a program. To remove a library, use SET LIBRARY TO without an argument. To remove a procedure, use SET PROCEDURE TO without an argument.

To remove the SYSPROC library, you must quit the dBASE IV session and restart dBASE IV with a different Config.db file that does not contain a SYSPROC setting.

dBASE IV Debugger

dBASE IV provides a program debugger to assist you in correcting program errors. You can activate the debugger in three ways:

- With SET TRAP ON, the debugger is activated automatically when a program error is encountered, unless an ON ERROR command is active.
- With SET TRAP ON, the debugger is activated if you press **Esc** while a program is running (assuming SET ESCAPE is ON and no ON ESCAPE statement is currently in effect).
- You can open the debugger directly by entering DEBUG <.prg filename> at the dot prompt.

TIP

If you want to debug a user-defined function, create a program that contains the function name, then enter DEBUG <.prg filename> *at the dot prompt. For example, if you have a function named Myudf(), write a Test.prg containing the following statement:*

```
x=Myudf()
```

Then, at the dot prompt, enter the following:

```
. DEBUG TEST
```

Debugger Windows

The debugger screen is divided into four windows: debug, display, breakpoint, and edit (see Figure 15-1).

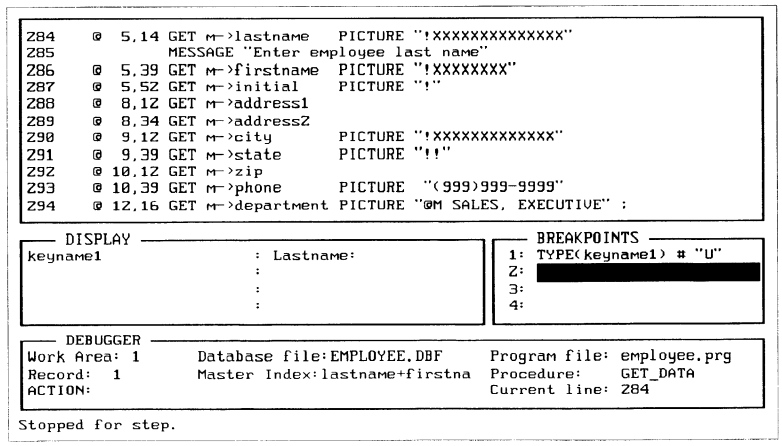


Figure 15-1 Sample debugger screen

The debugger windows have fixed sizes and locations.

The usual dBASE IV editing and navigation keys operate in all debugger windows except in the right side of the display window (see the Display Window section). Press **F9** to toggle between the debugger screen and the application.

Debug Window

The debug window across the bottom of the screen is active when you first activate the debugger. After activating another window, press **Esc** or **Ctrl-End** to get back to the debug window.

This window shows general database file and program information (see Figure 15-1). Note that the line number is relative to the beginning of the file, not to the beginning of the procedure. You issue debugger commands at the **ACTION:** prompt in the debug window (see the Debugger Commands section).

Display Window

To activate the display window, type D at the **ACTION:** prompt. This window shows the status of the program environment. You may enter one or more expressions on the left side of the display window. The debugger evaluates the expressions and displays the results in the right side of the window.

Unlike the other windows in the debugger, the right side of the display window is read-only. It scrolls vertically in conjunction with the left side.

Figure 15-1 shows a sample display window. Note that you need only enter the expression you want evaluated, not the ? command verb. Be sure that your entry is a valid dBASE expression.

Breakpoint Window

Type B at the **ACTION:** prompt to activate the breakpoint window. The debugger halts program execution when it encounters a breakpoint, which can be any logical expression. Type each breakpoint on a separate line in the breakpoint window. You can enter up to ten breakpoints; the debugger assigns a line number to each. You may enter conditions that use variables you haven't yet defined. Just ignore the error message and define the variable later.

The debugger evaluates the breakpoints after each line of code is processed. If a condition evaluates to true, the debugger stops the program and activates the debug window. In addition, the message **Breakpoint** appears on the message line, followed by the line number of the breakpoint that caused the halt.

Edit Window

The edit window displays the actual program commands as they execute. The next line to be executed appears in inverse video. To modify the program displayed in the edit window, type E at the **ACTION:** prompt.

Use the edit window just as you do the program editor. Changes that you save (with **Ctrl-End**) are added to the source code, but they do not execute until you recompile the program.

Debugger Commands

You control the debugger with a few simple commands and keys (see Table 15-1). Type the commands at the **ACTION:** prompt without pressing ↵. The first three commands listed in the table access the various debugger windows (see the Debugger Windows section). More detailed descriptions of the other commands follow.

Table 15-1 Debugger commands and keys

Command or Key	Action
B	Enters the breakpoint window
D	Enters the display window
E	Enters the edit window
R	Runs program until breakpoint or error occurs
L	Begins execution at specified line
[<n>]S	Steps through program <n> steps at a time
[<n>]N	Steps through program <n> steps at a time at current or higher level only
P	Shows trace of calling programs and procedures
U	Suspends program and exits to dot prompt
Q	Quits the debugger and cancels program
↵	Issues a 1S or 1N, whichever is most recent
F1 Help	Displays **Help** menu of debugger commands
F9	Toggles debugger windows on and off
Ctrl-L	Executes the highlighted command
Ctrl-T	Toggles the breakpoint on the selected line
PgUp, PgDn **Ctrl-PgUp, Ctrl-PgDn**	Moves through the source code

Running a Program

To run a program from within the debugger, type R at the **ACTION:** prompt. The program executes until a breakpoint or program error occurs. Use the L command to begin execution at a particular line. When you type L at the **ACTION:** prompt, the debugger prompts you to enter a line number. Enter the desired line number, or press ↵ to begin at the current line.

NOTE
Your program executes more slowly when you run it in the debugger, when you have set breakpoints, or when SET TRAP is ON.

Stepping Through a Program

In order to find a particularly elusive bug, you can step through your program one or more commands at a time. Enter S or press ↵ repeatedly at the **ACTION:** prompt to step through a program one command at a time. (The debugger skips comment lines.)

To speed up the steps, precede the S command with a number. For example, 10S stops every 10 commands. To limit the steps to the current program level or higher, use the N command rather than S. Procedures at a level deeper than the current one execute, but they do not count as separate commands in the trace.

When stepping through a program, the debugger stops at breakpoints and errors as well as at the specified points. After all commands have executed, control returns to the debug window. (Press **Esc** to interrupt.)

Tracing Procedure Calls

Type P at the **ACTION:** prompt to see a list of all calling programs and procedures. The list appears near the bottom of the screen. Figure 15-2 shows a sample program trace.

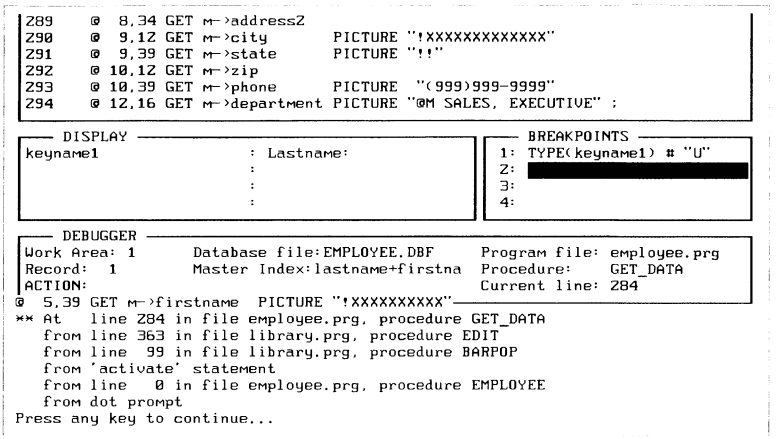


Figure 15-2 Sample program trace

Suspending, Resuming, and Quitting the Debugger

To suspend the debugger and exit to the dot prompt, type U at the **ACTION:** prompt. Once at the dot prompt, you enter commands in the usual manner. Enter RESUME at the dot prompt to re-enter the debugger.

NOTE
If you execute the debugger again while in a suspended debugger session, you will see a prompt box with the message **Debugger already in use. Enter RESUME to return to the Debugger.**

If you want to quit rather than suspend the debugger session before going to the dot prompt, type Q at the **ACTION:** prompt. This cancels both the debugger and the program and returns control to the dot prompt.

Sample Debugging Session

This section contains a sample session using the dBASE IV program debugger. If you want to try the sample session yourself, create the following program, Buggy.prg:

```
* Buggy.prg
* This program has a bug
*
SET TALK OFF
CLEAR
manswer = Yes_no("Can you find the bug in this program?")
? "You answered", IIF(manswer, "YES", "NO")
RETURN
FUNCTION Yes_no
PARAMETERS mmessage
mkeyin = "x"
DO WHILE .NOT. mkeyin $ "YNyn"
    @ 14, 0
    @ 14, 0 SAY mmessage + " (Y/N) "
    ?? CHR(7)
    mkeyin = INKEY(0)
ENDDO
@ 14, 0
RETURN mkeyin $ "Yy"
```

Buggy.prg has a small main procedure that calls the user-defined function Yes_no. The Yes_no UDF displays the message passed as its parameter, and loops until the user enters Y or N. It returns .T. if Y is pressed and .F. if N is pressed.

Buggy.prg has an error in it. To see the error, enter DO Buggy at the dot prompt. Press Y or N to answer the question. You see the error box shown in Figure 15-3, which indicates that *mkeyin* is the wrong data type for the substring comparison operator ($).

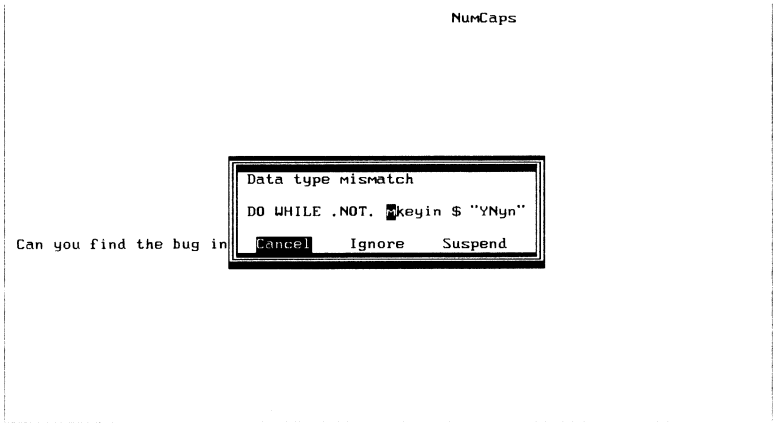


Figure 15-3 Program error box

Select **Cancel** from the error box. You see an error trace, and control returns to the dot prompt.

Data type mismatch is a common program error. According to the error trace, the error was detected at line 13 in the Yes_no procedure. The substring comparison operator expects *mkeyin* to be a character variable. Use the debugger to find out the data type of *mkeyin* when the error occurs.

First, activate the debugger:

```
. DEBUG Buggy
```

Buggy.prg appears in the debugger (see Figure 15-4).

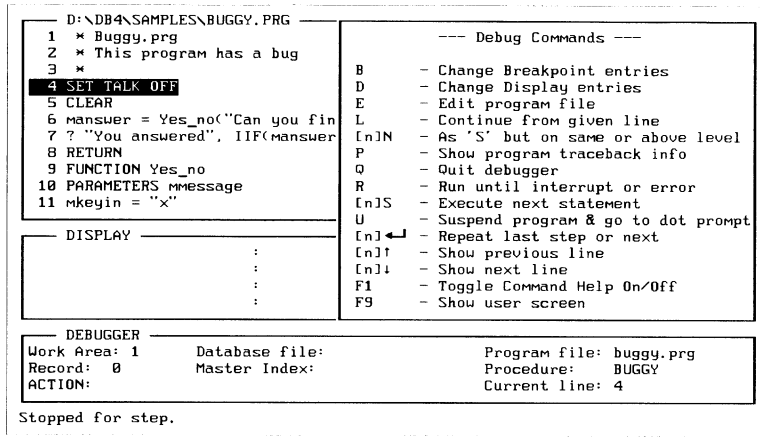


Figure 15-4 Buggy.prg in the program debugger

Now type D to activate the display window, and enter mkeyin followed by ↵. Since *mkeyin* is not yet defined, the message **Variable not found** appears.

Press **Esc** to return to the **ACTION:** prompt, and enter B to activate the breakpoint window. Enter the following line, followed by ↵:

```
TYPE("mkeyin") <> "U"
```

Press **Esc** to return to the **ACTION:** prompt and press R to run Buggy.prg. The debugger screen shown in Figure 15-5 appears.

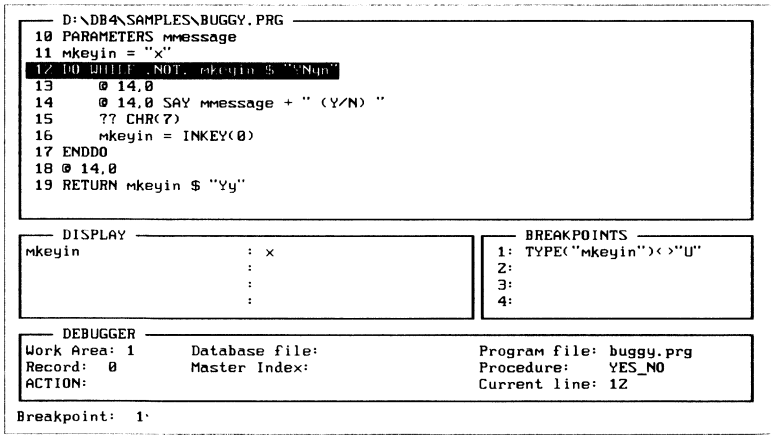


Figure 15-5 Stopping at a breakpoint

The **Breakpoint: 1** message in the lower left-hand corner tells you that the debugger stopped executing the program because the first breakpoint condition was met. An *x* appears in the right side of the display window to show that *mkeyin* has been assigned a value.

Press B to activate the breakpoint window and delete the expression on line 1. Press **Esc** to return to the **ACTION:** prompt.

The steps so far demonstrate one method of getting quickly to the part of a program you want to debug. The TYPE() function was used in a breakpoint expression to execute the program until *mkeyin* was defined. The TYPE() function is useful in the debugger for this purpose. It returns a "U" (unknown) until a variable is assigned a value.

You can place an asterisk (*) in front of a breakpoint to comment it out and use it later.

You can clear out a breakpoint by pressing **Home** to go to the start of the line, and then pressing **Ctrl**-Y to delete it. Next, press ↵ to accept the change.

The PROGRAM() and LINENO() functions are also useful in breakpoint expressions when you want to begin debugging at a specified program statement. For example, you could execute until the program enters the Yes_no function by entering PROGRAM() = "YES_NO" in the breakpoint window. If you want to stop at a specific line in the program, enter an expression like PROGRAM() = "YES_NO" .AND. LINENO() >= 13.

Next, you can step through the program a line at a time, watching the value of *mkeyin*. Press S to execute the next line. This is the line that results in the error, but this time it executes correctly. That means that the error occurs within the DO WHILE loop.

Press R to continue running the program without stopping at each line. The debugger screen disappears and the prompt displays. Enter a Y. The debugger screen in Figure 15-6 appears.

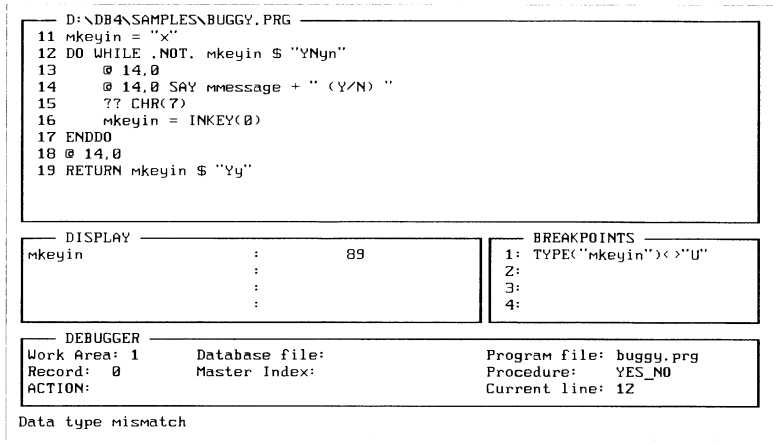


Figure 15-6 Debugger stopped at a program error

The program returned to the debugger because the **Data type mismatch** error occurred. The value of *mkeyin* is 89 (or 121 if you pressed a lowercase y), instead of the letter Y.

The last line to change the value of *mkeyin* was the line containing the INKEY() function. The INKEY() function returns a number, but the substring comparison expects a character string, resulting in the **Data type mismatch** error.

The way to correct the problem is to convert the INKEY() result to character data type. Press E to activate the program editor, and edit the INKEY() line so that it looks like this:

```
mkeyin = CHR(ABS(INKEY(0)))
```

The CHR() function converts a number to a character. Adding CHR() corrects the original problem but also introduces another error. CHR() only converts positive numbers, and INKEY() can return negative numbers, for instance when **F2** is pressed. The ABS() function takes the absolute value of the number returned by INKEY(), making sure that CHR() receives a valid argument.

Save your changes, and press Q to exit the debugger. Enter DO Buggy to verify that the program now executes without error.

Linking

DBLINK is a dBASE program linker. It combines all of the dependent files in a program into one executable program file. This makes all the files you need to run an application available to your main application program.

In a large application, it makes for better organization if you link related files into several executable files rather than one very large .dbo file. This includes .qbo, .lbo, .fro, and .fmo files.

Linked applications run faster, and take up less disk space than do many unlinked .dbo files.

Using DBLINK

The Install program puts the files Dblink.exe and Dblink.res into the dBASE home directory. This directory should be in your DOS path, or you can copy the DBLINK files to your working directory. The two files must be in the same directory to work.

From the DOS prompt, at your working directory where all the compiled object files are, type:

```
DBLINK <input filename> /L /D ↵
```

<input filename> is the name of the main module of your program. DBLINK goes through the main file and identifies all the procedures and files that it contains. Next, it goes through all the subroutines inside the program modules that make up the main program. When DBLINK finds all the main routines and subroutines, it links them into one object file.

The /L option creates a disk file with the main source file name and a .txt extension. All linked module names and all referenced but not linked module names are listed to this file.

The /D option displays progress messages to the screen.

Only direct file references are found by DBLINK. Indirect file references, file expressions with the & macro operator, are not resolved. The linked object file will not contain these references. Therefore, you must replace indirect references to subroutines with direct references before you can compile the program.

Compile all input files, beginning with the main program module, before you link them. Format files and SQL program files must be compiled separately before you link them. Report, form, label, and QBE files must also be compiled.

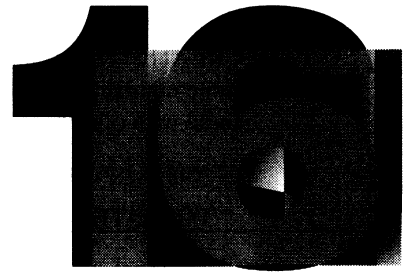
The maximum number of files and procedures that can be linked with DBLINK is 963. Each procedure can contain up to 65,520 bytes of code. The number of procedures that you can access depends on the amount of available RAM in DOS.

Messages

If you enter the DBLINK command without a filename, it terminates with a message showing correct usage.

If DBLINK cannot find the referenced filename, it displays the message: **Cannot open <input filename> File <input filename> does not exist**

Low-level File Functions



dBASE IV provides functions to manipulate operating level binary files. These are C language-type file functions that require expert knowledge of the operating system. These functions work with data stream files.

Long filenames and extensions that are permitted in UNIX are truncated in DOS. Truncation of long names creates identical filenames for files that are not differentiated in their first eight characters. Low-level files with long names created under UNIX write over one another without warning when ported to DOS.

See the specific low-level file function name in the *Language Reference* manual for more information.

What This Chapter Covers

- File opening functions
- File reading functions
- File writing functions
- File pointer functions
- File closing function
- Low-level file error detection

File Opening Functions

There are two functions for opening a low-level file. FCREATE() creates and opens a low-level file, and returns a dBASE assigned handle number. FOPEN() opens an existing file by its filename and returns a dBASE assigned handle number. dBASE assigned file handle numbers are different from operating system level file handles.

FCREATE()

FCREATE() creates and opens a new low-level file and returns the dBASE assigned file handle number. You should save this number to a variable in order to refer to this file handle with other low-level file functions.

The syntax for the FCREATE() function is:

FCREATE(<filename>)

<filename> is any filename permitted by the operating system. You cannot FCREATE() or FOPEN() a file already opened by a dBASE IV command; you will get the dBASE IV error message **File already opened**.

WARNING
FCREATE() creates a new file at the operating system level. If you use the same name as an existing file, DOS and UNIX truncate your file to zero length and all data is lost.

FOPEN()

FOPEN() opens an existing file with the specified privilege level, and returns a file handle number. An attempt to FOPEN() a file that is currently open and was opened with either FCREATE() or FOPEN() will return a new file handle number, and refer to the original file with the original opening attributes. Any attributes supplied by the second user are ignored.

The syntax for FOPEN() is as follows:

FOPEN(<filename>, <"privilege">)

The privilege letter is not case sensitive, must be a quoted string, and can be any of the following:

- "r" — read only
- "w" — write anywhere in the file
- "a" — append to end of file only
- "rw" or "wr" — read and write
- "ar" or "ra" — read and append

Write and append privileges are mutually exclusive and cannot be combined. If you do not specify a privilege level, the default is both read and write.

File Reading Functions

There are two functions for reading the contents of low-level files. FREAD() reads the specified number of bytes from a file and returns them as a character string, including any end-of-line characters. FGETS() reads a string of characters terminated by an end-of-line indicator and returns the character string without the end-of-line marker to dBASE IV.

FGETS()

FGETS() reads the specified number of characters, or reads until it encounters the next end-of-line character, starting from the current file pointer in a low-level file. It returns the string to dBASE IV. The maximum string length is 254. FGETS() positions the pointer after the last character read. The end-of-line character is dropped and not returned.

The syntax for FGETS() is as follows:

FGETS(<filehandle> [,<bytes to read>] [,<eol character(s)>])

The file handle must be assigned by FOPEN() or FCREATE().

The number of bytes to read can be 0 to 254.

The end-of-line character must evaluate to one or two characters.

FGETS() recognizes the DOS, UNIX, or Macintosh end-of-line characters. If you do not specify an end-of-line, FGETS() will search until it encounters any one of the following characters:

- 0AOD (CRLF on DOS)
- 0A (LF on UNIX)
- 0D (CR on Macintosh)

FREAD()

FREAD() reads the specified number of bytes from a low-level file into a character string.

NOTE
The application developer must be aware of the high-low byte order of the computer hardware architecture when porting to some UNIX platforms. DOS, VMS, and many UNIX platforms use the high-low convention.

The syntax for FREAD() is as follows:

FREAD(<filehandle>, <bytes to read>)

The filehandle must be assigned by FOPEN() or FCREATE().

The number of bytes to read is between 0 and 254, inclusive. FREAD() will read until the specified or the maximum bytes are read or it encounters an EOF marker. It does not return the EOF marker.

File Writing Functions

FPUTS() and FWRITE() write to a buffer in RAM that is referenced by the file handle. The FFLUSH() function writes the file out to a disk file and should be used periodically to prevent data loss.

FFLUSH()

FFLUSH() writes the low-level file to disk under the filename assigned to it with FCREATE() or FOPEN() and clears the buffer. It returns a logical true (.T.) if the write operation is successful, or a logical false (.F.) if unsuccessful.

To be certain that FFLUSH() updates the disk file, disable any delayed writing of any disk cache programs.

FPUTS()

FPUTS() writes the specified length character string to a low-level file, including the end-of-line indicator. The syntax of FPUTS() is as follows:

```
FPUTS(<filehandle>,<character string to write>
[, <number of characters to write>] [, <eol character(s)>]
```

FPUTS() returns the total number of characters written, including any end-of-line characters.

If the write is not successful, FPUTS() returns a zero. FERROR() will return the operating system error number.

FWRITE()

FWRITE() writes the specified number of characters to a low-level file.

The syntax for FWRITE() is as follows:

```
FWRITE(< filehandle>,<character string to write>
[, <number of bytes to write>])
```

The file handle must be assigned by FOPEN() or FCREATE().

The entire character string of up to 254 is written to the file, unless a lower number is specified.

File Pointer Functions

These functions move the file pointer or return its location in an open low-level file. They are used to position the file pointer before reading from or writing to low-level files.

FEOF()

FEOF() returns a logical true (.T.) if the pointer is at the end of file, or a false (.F.) if it is not.

The syntax for FEOF() is as follows:

FEOF(<filehandle>)

The filehandle must be assigned by FOPEN() or FCREATE().

FSEEK()

FSEEK() positions the file pointer to any location in a low-level file.

The syntax for FSEEK() is as follows:

FSEEK(<filehandle> , <number of bytes to move> [, <starting position>])

The file handle must be assigned by FOPEN() or FCREATE().

The number of bytes can be between $-(2^{31})+1$ and $(2^{31})-1$.

Negative numbers move the pointer towards the beginning of the file, and positive numbers move the pointer to the end of the file. The starting position is 0 for BOF, 1 for current file pointer, 2 for EOF. FSEEK() returns the final position of the file pointer relative to BOF.

FGETS() and FREAD() move the file pointer one byte past the last character read, or immediately past the end-of-line character.

File Closing Function

FCLOSE() closes a low-level file opened with FCREATE() or FOPEN().

The syntax for FCLOSE() is as follows:

FCLOSE(<filehandle>)

The file handle must be assigned by FOPEN() or FCREATE().

FCLOSE() returns a logical true (.T.) if the file closure is successful, or a logical false (.F.) if it is unsuccessful.

Low-Level File Error Detection

The FERROR() function can return a zero, indicating that no error has occurred, when in fact a low-level file I/O error is intercepted by dBASE IV and does not reach the operating system. To find out whether the zero returned by FERROR() is valid or not, use the dBASE IV ERROR() function to get the dBASE error number for the intercepted error.

Some commonly returned DOS error codes are as follows:

2 File not found

3 Path not found

4 Too many files open

5 Access denied

6 Invalid handle

8 Insufficient memory

Program Examples of Low-Level File I/O

This section has five programming examples of using low-level file I/O functions.

Four different UDF's read or write 16- or 32-byte integers into low-level files.

The last example determines the record size in the file header using the FReadI16() UDF with low-level I/O. Database file header information is in Appendix D of the *Language Reference* manual.

Function FReadI16 reads in two bytes representing a two-byte unsigned integer into a low-level file.

```
FUNCTION FReadI16
PARAMETERS pn_handle
*-------------------------------
* NAME
* FReadI16
*
* DESCRIPTION
* FReadI16 reads in two bytes that would represent a
* two-byte unsigned integer in DOS. FReadI16 reads from
* the current file position. FReadI16 will move the file
* pointer to the next byte after the two bytes that it
* processed. FReadI16 returns a numeric value based on
* the two bytes value.
*
* SYNOPSIS
* FReadI16(<pn_handle>)
*
* PARAMETERS
* pn_handle = file handle to read from
*
* LIMITATIONS
* FReadI16 assumes a valid file handle.
* FReadI16 does not check for end of file.
*
*-------------------------------
PRIVATE ALL LIKE l?_*
ln_result = 0
ln_low = ASC(FREAD(pn_handle,1))
ln_high = ASC(FREAD(pn_handle,1)) * 256
ln_result = ln_high + ln_low
RETURN (ln_result)
*EOF: FReadI16(pn_handle)
```

Function FReadI32 reads in four bytes representing an unsigned integer into a low-level file.

```
FUNCTION FReadI32
PARAMETERS pn_handle
*-------------------------------------
* NAME
* FReadI32
*
* DESCRIPTION
* FReadI32 reads in four bytes that would represent a
* four-byte unsigned integer (long) in DOS. FReadI32 reads from
* the current file position. FReadI32 will move the file
* pointer to the next byte after the four bytes that it
* processed. FReadI32 returns a numeric value based on
* the four bytes value.
*
* SYNOPSIS
* FReadI32(<pn_handle>)
*
* PARAMETERS
* pn_handle = file handle to read from
*
* LIMITATIONS
* FReadI32 assumes a valid file handle.
* FReadI32 does not check for end of file.
*
*-------------------------------------
PRIVATE ALL LIKE l?_*
ln_result = 0
ln_byte1 = ASC(FREAD(pn_handle,1))
ln_byte2 = ASC(FREAD(pn_handle,1)) * 256
ln_byte3 = ASC(FREAD(pn_handle,1)) * 256 * 256
ln_byte4 = ASC(FREAD(pn_handle,1)) * 256 * 256 * 256
ln_result = ln_byte1 + ln_byte2 + ln_byte3 + ln_byte4
RETURN (ln_result)
*EOF: FReadI32(pn_handle)
```

Function FWritI16 writes an unsigned integer 16 value.

```
FUNCTION FWritI16
PARAMETERS pn_handle, pn_value
*-----------------------------------
* NAME
* FWritI16
*
* DESCRIPTION
* FWritI16 writes an unsigned INT16 value <pn_value> at the
* current file position specified by the file handle <pn_handle>.
* FWritI16 will return a .T. if it wrote two bytes. Otherwise,
* FWritI16 will return a .F. value.
*
* SYNOPSIS
* FWritI16(<pn_handle>, <pn_value>)
*
* PARAMETERS
* pn_handle = file handle to write to
* pn_value = the unsigned INT16 value to write
*
* LIMITATIONS
* FWritI16 assumes a valid file handle
*
* DEPENDENCIES
*-----------------------------------
PRIVATE ALL LIKE l?_*
ll_result = .T.
ln_count = 0
ln_value = pn_value
ln_byte1 = MOD(ln_value, 256)
ln_byte2 = (ln_value - ln_byte1) / 256
ln_count = FWRITE(pn_handle, CHR(ln_byte1), 1)
ln_count = ln_count + FWRITE(pn_handle, CHR(ln_byte2), 1)
IF ln_count < 2
   ll_result = .F.
ENDIF
RETURN(ll_result)
*EOF: FWritI16(pn_handle, pn_value)
```

Function FWritI32 writes an unsigned integer 32 value.

```
FUNCTION FWritI32 PARAMETERS pn_handle, pn_value
*-------------------------------------
* NAME
* FWritI32
*
* DESCRIPTION
* FWritI32 writes an unsigned INT32 value <pn_value> at the
* current file position specified by the file handle <pn_handle>.
* FWritI32 will return a .T. if it wrote four bytes. Otherwise,
* FWritI32 will return a .F. value.
*
* SYNOPSIS
* FWritI32(<pn_handle>, <pn_value>)
*
* PARAMETERS
* pn_handle = file handle to write to
* pn_value = the unsigned INT32 value to write
*
* LIMITATIONS
* FWritI32 assumes a valid file handle
*
* DEPENDENCIES
*-------------------------------------
PRIVATE ALL LIKE l?_*
ll_result = .T.
ln_count = 0
ln_value = pn_value
ln_byte1 = MOD(ln_value, 256)
ln_value = (ln_value - ln_byte1) / 256
ln_byte2 = MOD(ln_value , 256)
ln_value = (ln_value - ln_byte2) / 256
ln_byte3 = MOD(ln_value , 256)
ln_value = (ln_value - ln_byte3) / 256
ln_byte4 = MOD(ln_value , 256)
ln_count = FWRITE(pn_handle, CHR(ln_byte1), 1)
ln_count = ln_count + FWRITE(pn_handle, CHR(ln_byte2), 1)
ln_count = ln_count + FWRITE(pn_handle, CHR(ln_byte3), 1)
ln_count = ln_count + FWRITE(pn_handle, CHR(ln_byte4), 1)
IF ln_count < 4
    ll_result = .F.
ENDIF
RETURN( ll_result )
*EOF: FWritI32(pn_handle, pn_value)
```

Determining Record Size from .dbf Header

You can determine the record size in an existing database from the dot prompt:

```
. ? FOPEN('Goods.dbf', 'r'
  4
. ? FSEEK(4,10,0)
  10
. ? FReadI16(4)
  157
. ? FCLOSE(4)
```

To verify that you read the correct record size, enter the following at the dot prompt:

```
. USE GOODS
. ? RECSIZE()
  157
```

Design and Data Surface Programs

The open architecture of the dBASE IV Control Center allows advanced users and developers to write programs that customize the design surfaces of the Control Center. These programs are referred to as *surface programs* in this manual.

A developer can utilize a custom program instead of the design tool built into dBASE IV directly from the Control Center. For example, an entry program can launch a third-party report writing program from the **Reports** panel of the Control Center.

Two menu options from the Control Center data and design surfaces connect to the user-defined programs. The options are **Invoke layout program** and **Load field program**. They support the open architecture of the Control Center by starting up the custom programs you specify in your dBASE IV configuration file.

The dBASE IV configuration file is no longer limited to the name Config.db; it can have any filename permitted by the operating system. The /c switch (explained in Chapter 16 of this manual) lets you pass the name of a configuration file to dBASE at start up from the operating system prompt. This permits different customized versions of dBASE IV to be loaded with ease.

In the absence of the /c switch and configuration filename, dBASE IV will look for and load a file named Config.db, if it exists.

Writing surface programs requires experience in dBASE IV application programming. Anyone writing programs to customize the Control Center should debug them from the dot prompt before launching them from the Control Center surfaces. Without the proper exception handling and exit behavior in your programs, you may lose the ability to return to the Control Center. If this happens, you will need to reboot your computer, then start dBASE IV with a Config.db file that does not reference the surface programs that cause problems.

What This Chapter Covers

- Surface Program Definitions
- Surface Program Support
  - Specifying Design Surface Programs
- Control Center Interactions with Surface Programs
  - Control Center Actions
  - Control Center Exclusions
  - Control Center Passed Parameters
- Entry Programs
- Exit Programs
  - Exit Conditions
  - Design (**Shift-F2**) and Data (**F2**) Key Behavior
- Layout Programs
- Field Programs
- Execution Programs

Surface Program Definitions

The open architecture Control Center supports entry, exit, layout, field, and execution programs.

Entry program — This is a program or procedure that runs when you enter a surface. It can set up a special program environment, run a series of procedures to accomplish a task, display a special menu or popup, or supersede the design surface with another application.

Exit program — This is a program or procedure that runs when you quit a design surface. It can restore the environment to its state prior to the entry program's execution, and generate any code which needs to be generated as a result of user activity on the design surface.

Layout program — This is a program or procedure that gets its name from the surface it starts from; that is, the **Layout** menu of the queries, forms, reports, and labels design surfaces. Layout programs can also change the environment, display a menu, let the user execute the options from its menu, and restore the environment when the user exits.

Field program This is a program or procedure that gets its name from the surface it starts from; that is, the **Fields** menu of the forms, queries, reports, or labels design surfaces. Field programs can also change the environment, display a menu, let the user execute the options from its menu, and restore the environment when the user exits.

Execution program This is a program or procedure that runs an application from the forms, queries, reports, or labels design surfaces before executing the query report, form, or label. An execution program may replace the design surface, and may be a dBASE IV application or an external application.

Surface Program Support

Control Center data surfaces are the **Catalog** menu and **Data** panel. The data surfaces provide internal program hooks for entry and exit programs, but Browse and Edit support only entry programs.

Control Center design surfaces are queries, forms, reports, labels, and applications. In addition to entry and exit programs, queries, forms, reports, and labels support layout, field, and execution programs.

The Applications Generator supports only an entry program. There is no way to launch programs from the Applications Generator surfaces. However, if you create a new report, label, or form in the Applications Generator, then transfer to one of the design surfaces, the programs you have set up in the Applications Generator remain active.

Figure 17-1 shows an overview of these programs.

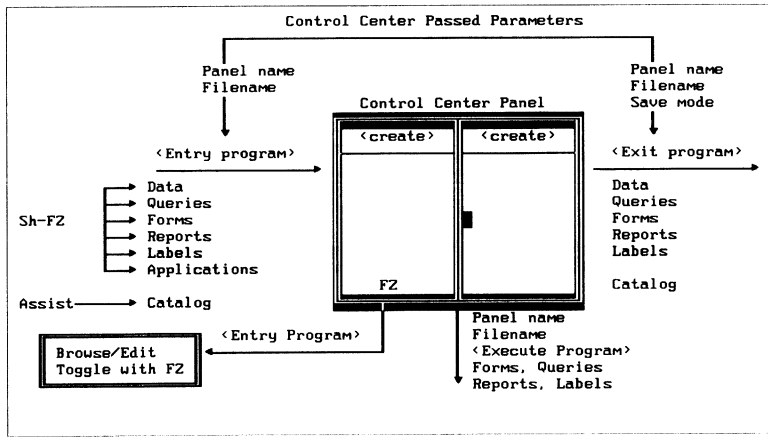


Figure 17-1 Surface program launching overview

As shown in Figure 17-1, the Control Center passes parameters to the surface programs it launches. The complete list of returned parameters is in Table 17-1.

Specifying Design Surface Programs

The user-defined programs for data and design surfaces are specified from the configuration file using special keywords. The program names should be entered into the configuration file as shown below:

Keyword = <entry>,<exit>,<layout>,<field>,<execute>

To skip over a program type, enter a comma as a placeholder for that parameter. Program names support full file specification, can be upper– or lowercase, and must be entered without delimiters.

See Table 17-1 for Config.db keywords and the types of programs supported from each surface. Some surfaces do not support all possible types of surface programs.

Table 17-1 Keywords and supported program types

Control Center Surface	Config.db Keyword	Entry	Exit	Layout	Field	Execute
Catalog menu	PRGCC	yes	yes	yes		
Database design	PRGDATA	yes	yes			
Queries design	PRGQUERY	yes	yes	yes	yes	yes
Forms design	PRGFORM	yes	yes	yes	yes	yes
Reports design	PRGREPORT	yes	yes	yes	yes	yes
Labels design	PRGLABEL	yes	yes	yes	yes	yes
Applications	PRGAPPLIC	yes				
Browse screen	PRGBROWSE	yes				
Edit screen	PRGEDIT	yes				

Control Center Interactions with Surface Programs

The Control Center performs some tasks to support surface programs.

Control Center Actions

- The Control Center passes the name of the file that was highlighted in the data or design panel to the execution program.
- A RETURN TO MASTER or CANCEL statement in the execution program will return the user to the Control Center which will repaint the screen to display its panels.

- A RETURN statement in the execution program will return control to the QUERY, REPORT FORM, LABEL FORM, or the FORMS designer routines.
- If a program file named in the configuration file is not found, the Control Center returns the **File not found** error message.
- The navigation line does not always apply to the external program, and ends up as screen clutter; therefore, the design surfaces turn it off during the execution of any type of external program.

When the external program finishes, the design surfaces restore the navigation line.

Control Center Exclusions

The Control Center does not perform the usual tasks for surface programs as it does for dBASE IV. The Control Center:

- Does not check to see if a database or view is in use.

  The developer must put the necessary files in use.

- Does not open the database or view associated with the highlighted file.

  The developer must save the filename parameter passed by the Control Center and put that database file or view in use.

- Does not search for a format file.

  The developer must SET FORMAT TO the appropriate file inside the execution program.

- Does not display the dialog box to indicate that the file currently in use is not the same as the associated database or view.

  The developer must code such a display box and write the routines to check that the proper files are being used.

- Does not display the **Print** submenu from the **Reports** and **Labels** panels if the execution program ends with a RETURN TO MASTER.

  The developer must code a menu dialog box to handle user input into the printer set-up if this is required.

Restoring Control Center Settings

In order to maintain the integrity of the Control Center and the user's design surface session, your surface program should keep track of the following settings and restore them upon completion:

- Catalogs

The foundation of the Control Center is the catalog file. When returning from a surface program, ensure that you return control to the original catalog file. Commands like CLEAR ALL, CLOSE ALL, and SET CATALOG TO close the active catalog file. If

you use these commands, you must reset the catalog file before ending the surface program. Use the CATALOG() function to save the current catalog, then restore it when ending the surface program.

■ Database files

Commands like USE, CLOSE DATABASES, CLOSE ALL, and CLEAR ALL close an open database file. If you use these commands in your surface program, you must reopen any database files that were in use before the surface program was launched. Use the DBF() function to save the name of any database files in use, then restore them when ending the surface program.

Note
If DBTRAP is ON, and a database file was at the beginning of the surface program, CLOSE ALL, CLOSE DATABASES, and CLEAR ALL leaves that one database file open.

■ Indexes

Certain design surfaces, such as the queries design screen, rely on indexes associated with the user's database files. If your surface program deletes an index that was associated with a database file in use when your surface program was launched, then you must recreate the index before returning to the design surface. Use the MDX(), NDX(), and KEY() functions to save the name and key expressions of any indexes, and then recreate them when ending the surface program.

■ Work area

In your surface program, you can move to other work areas and open and close files in these other work areas. Unless you are using a custom Form, Report, or Label.gen, you must be sure to return to the original work area before your field or layout program returns to the design surface. Use the SELECT() function to save the current work area, then reselect it when ending your surface program.

■ The screen

Through the use of surface programs, you are given complete control of the design and data screens. Your program can clear the screen and add any output you want into it. When returning to the surface, dBASE does not automatically repaint the screen for you. If you want to maintain the user's screen, use the DEFINE WINDOW and ACTIVATE WINDOW commands in the beginning of your surface program, performany screen output, then issue a DEACTIVATE WINDOW command before returning to the surface. Another way to save the surface screen is to issue a SAVE SCREEN command at the beginning of your surface program, then a RESTORE SCREEN command at the end.

Control Center Passed Parameters

The Control Center passes certain parameters to the surface programs it launches. The parameter names have been abbreviated as follows:

P — Control Center panel name

F — Filename under highlight

O — Object name under highlight

T — Object type under highlight

E — Expression entered on surface

S — Save mode

These parameters are summarized in Table 17-2.

Table 17-2 Control Center passed parameters

Control Center Surface	Entry	Exit	Layout	Field	Execute
Catalog	P,F	P,F	P,F		
Data	P,F	P,F,S			
Queries	P,F	P,F,S	P,F	O,T,E	P,F
Forms	P,F	P,F,S	P,F	O,T,E	P,F
Reports	P,F	P,F,S	P,F	O,T,E	P,F
Labels	P,F	P,F,S	P,F	O,T,E	P,F
Applications	P,F				
Edit	P,F				
Browse	P,F				

S = save mode. This is one of three parameters: "SAVE" for save and exit, "RESUME" for save and continue, and "ABANDON" for do not save.

Control Center passed panel parameter names are "CC", "DATA", "QUERY", "FORM", "REPORT", "LABEL", "APPS", "BROWSE", and "EDIT".

Filenames use full file specification including drive and path. The SET FULLPATH setting is ignored.

Entry Programs

Entry programs are run from the Control Center before entering any one of its data or design surfaces.

If the entry program ends with RETURN, dBASE IV returns the user to the design surface that launched the entry program.

If the entry program ends with RETURN TO MASTER or CANCEL, dBASE IV returns the user to the Control Center.

It is possible for a user to use the **F2** and **Shift-F2** keys to enter data and design surfaces repeatedly. The developer must set flags in the entry programs so that potential recursive entries into the same design surface can be handled appropriately. For example, check the TYPE() of a variable before declaring it public.

Common entry program features include the following:

- Checks the status of DBTRAP, sets DBTRAP OFF at the beginning of the program, and restores DBTRAP to its original status when it terminates.
- Saves the user's environment, checks and puts the correct database file in use, and saves the user's file to a file with a .scb extension for use by the exit program.
- Declares parameters for the panel name, filename, and the exit mode the user chooses. These are returned by the Control Center.

Exit Programs

Exit programs are run when leaving the data or design surface. Both RETURN and RETURN TO MASTER or CANCEL end the execution of the exit program and return control to the Control Center.

The behavior of the exit program must reflect the save mode the user chooses. This is explained in the next section.

Exit Conditions

Exit programs have to behave differently depending on which option the user chooses to exit a design surface.

1. **Save and Continue**

   The design file is not closed.

   The appropriate Binary Named List file (.snl, .fnl, .lnl) is created. This captures the user's changes up to the last save, but no code file is generated to save generation time. The developer should set a RESUME flag in the exit program to note that the design has changed and a new BNL file exists.

   If there are additional changes which the user decides to abandon, then the developer can use the DGEN() function to generate code files from the last saved BNL file. If you are unfamiliar with template language code generation, see the DGEN() function in the Using the Template Language section of this manual.

2. **Save and Exit**

   The design file is closed.

   The appropriate Binary Named List file (.snl, .fnl, .lnl) is created but no code file is generated.

   If you want to generate code, use the DGEN() function with the BNL file either in your exit program or from the dot prompt.

3. **Abandon**

   The design file is closed.

   No BNL file is produced.

   The RESUME flag should be checked by the developer to see if a new BNL file was created with an earlier **Save and Continue**. If so, a new design file must be created with DGEN() from the last BNL file.

4. **Abandon to Transfer to Another Design Surface**

   Any time a user leaves one design surface to transfer to another surface, dBASE IV runs:

   - The exit program for the surface being left
   - The entry program for the surface being entered

The user can transfer out of a design surface in one of three ways:

- Return to the calling surface with **F2** or **Shift-F2**, or with the **Exit/Transfer** option from the menu. **Shift-F2** will not run the exit program from the data, forms, labels, and reports surfaces. **F2** will not run the exit program from the labels and reports design surfaces.
- **Save and Exit** from the surface being left. The design file is closed and the appropriate BNL file is created.
- Abandon from the surface being left. The design file is closed and no BNL file is generated.

The behavior of the Control Center when transferring between design surfaces is summarized in Figure 17-2.

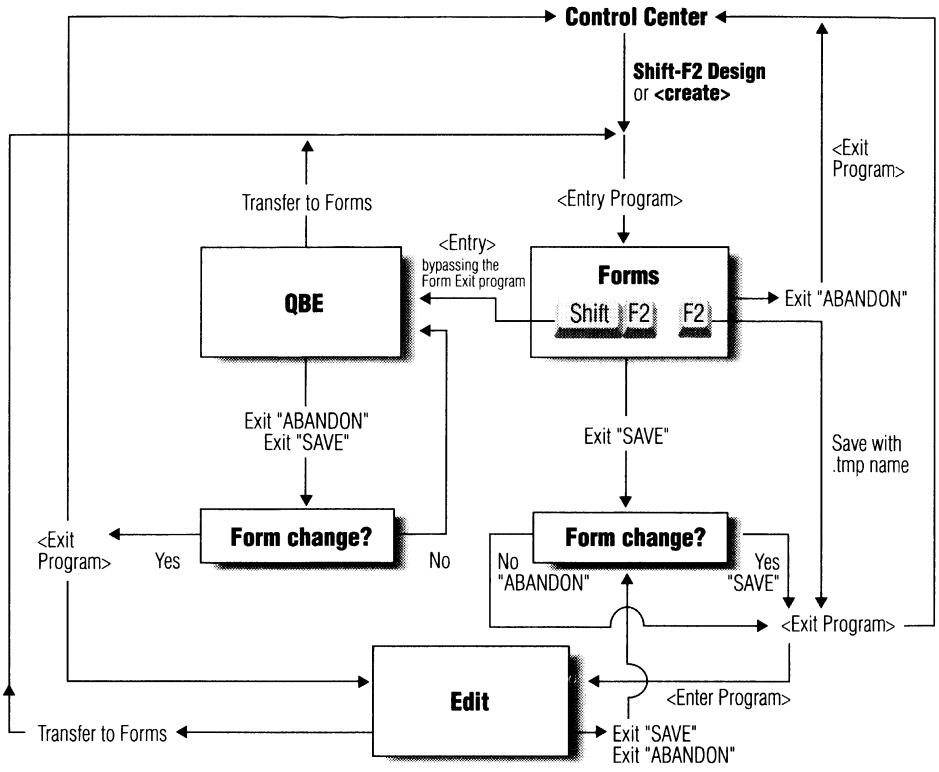


Figure 17-2 Transferring between design surfaces

Design and Data Key Behavior

The behavior of the function keys **Shift-F2** and **F2** are different when design surface programs are active. These actions are summarized below:

Function Key	Activated surface program type			
	Data	Forms	Queries	Reports and Labels
F2	Exit	Exit	Execute program	Entry program to Edit
Shift-F2	Exit	QBE entry	QBE entry	QBE entry

Note that the Control Center calls exit programs only from the data and forms design surfaces. Exit programs are not called from the other surfaces.

The **F2** and **Shift-F2** keys can move the user from surfaces into surface programs. The behavior of these keys with surface programs is summarized in Figure 17-3.

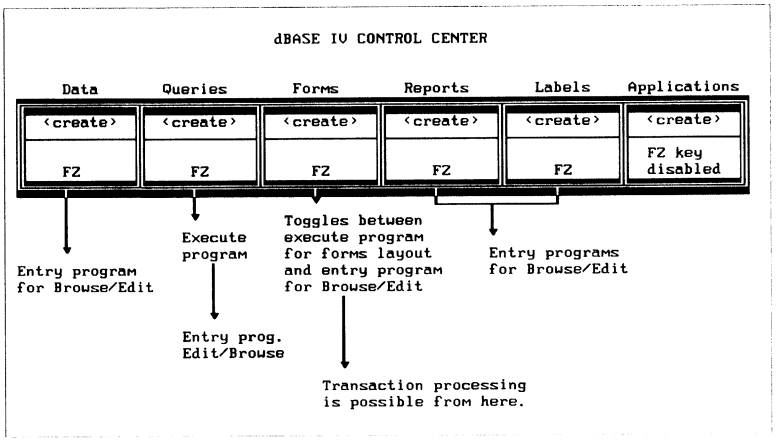


Figure 17-3 **F2** key behavior with entry and exit programs

If you want to generate code from the BNL files, then use the DGEN() function in the exit programs. You can specify a custom .gen file or use the template language Form.gen template file and the BNL filename. For example, to create a form from a BNL file called Profile.snl, add the following line of code to the exit program:

```
dgen("form","profile.snl")
```

Layout Programs

The **Layout** menu's **Invoke layout program** option launches surface programs. Figure 17-4 summarizes the surfaces from which you can launch layout programs and the parameters returned to the layout program by the Control Center.

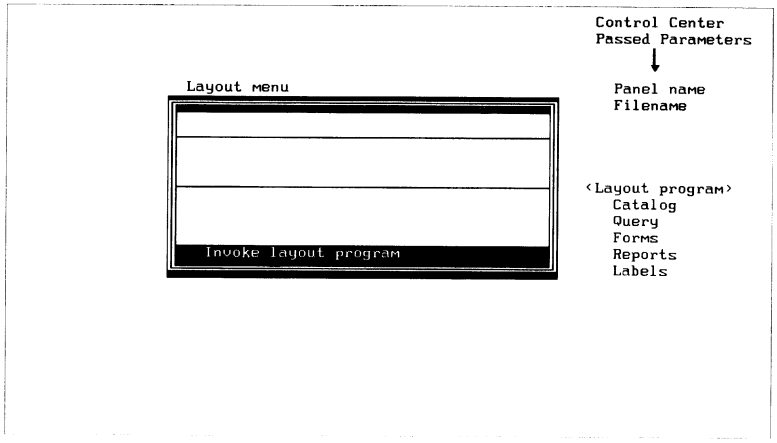


Figure 17-4 **Layout** menu program launching

Field Programs

The **Field** menu's **Load field program** option launches surface programs from the **Forms**, **Queries**, **Reports**, and **Labels** panels. Figure 17-5 shows the field program being launched from the **Add field** option. Parameters returned by the Control Center to the field program are also shown.

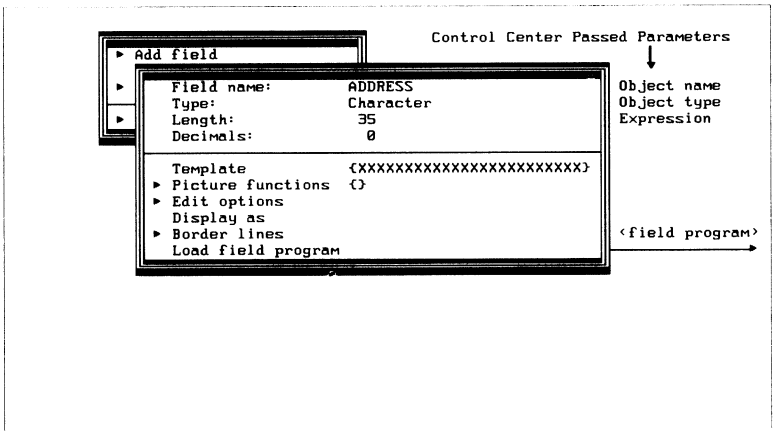


Figure 17-5 **Field** menu program launching

The object name parameters returned by the Control Center for field programs launched from the queries surface are dependent upon the position of the cursor on the surface. Table 17-3 lists the parameters. Actual parameter strings are shown in quotation marks.

Table 17-3 Queries surface returned object name parameters

Cursor Location	**Name of Object**
Pothandle	filename
File skeleton cell	field
Complex index	tag
Calculated field pothandle	"CALC'D FLDS"
Calculated field expression	calculated field name
Calculated field cell	calculated field expression
Condition box	"COBOX"
Field in view skeleton	.dbf field name

The object type parameters returned by the Control Center for field programs launched from the queries surface are listed in Table 17-4.

Table 17-4 Queries surface returned object type parameters

Cursor Location	Type of Object
Pothandle	"HNDL"
File skeleton cell	"FSKEL"
Complex index	"TAG"
Calculated field pothandle	"CHNDL"
Calculated field expression	"CEXP"
Calculated field cell	"CCELL"
Condition box	"COBOX"
Field in view skeleton	"VSKEL"

The expression parameters returned by the Control Center for field programs launched from the queries surface are listed in Table 17-5.

Table 17-5 Queries surface returned expression parameters

Cursor Location	Type of Expression
Pothandle	query operators
File skeleton cell	cell expression
Complex index	tag name
Calculated field pothandle	null
Calculated field expression	calculated field expression
Calculated field cell	cell expression
Condition box	condition box expression
Field in view skeleton	assigned field name

The returned object type parameters from the forms, labels, and reports screens are listed in Table 17-6.

Table 17-6 Forms, labels, and reports object type parameters

Cursor Location	Type of Object
Regular .dbf field	"FLD"
Calculated field	"CALC"
Pre-defined field	"PRE"
Summary field	"SUM"
Memory variable	"MEM"

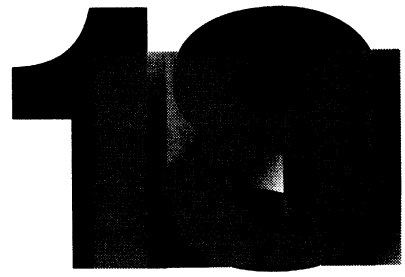
Execution Programs

Execution programs replace the programs that the dBASE IV Control Center launches from its data and design panels.

These programs can also be external applications (.exe files). dBASE IV must be rolled out of RAM with the RUN() function to allow the external application to run.

dBASE IV® Template Language

Learning Template Language



This section explains the dBASE IV template language and how to write template language programs, compile them into templates, and then use them with dBASE IV or with the stand-alone template language interpreter, Dgen.exe. This explanation is intended for programmers, developers, and users familiar with programming basics. You should be familiar with using the command line interface from DOS, and compiling and interpreting a program. A text editor that produces ASCII is essential; all of the templates described in this manual are written and compiled outside of dBASE IV.

This chapter introduces you to the concept of the template language. It describes the basics of writing and compiling simple template language programs.

Chapter 19 describes template language expressions.

Chapter 20 describes a template program called Code_doc.cod. You can run this template to produce formatted text files with a table of contents that references by line number all of the procedures, programs, and user-defined functions in any template file you want to document.

Chapter 21 is a reference to template language commands.

Chapter 22 is a reference to the template language built-in library of functions.

Chapter 23 describes user-defined functions contained in the file Builtin.def.

Chapter 24 describes the template language compiler, Dtc.exe, and the stand-alone interpreter, Dgen.exe.

Chapter 25 describes the error messages produced by Dtc.exe and Dgen.exe.

What is Template Language?

Template language is a programming language that creates custom form, label, report, and application programs for your database files. It generates program files from screen layout objects such as reports, forms, and labels, or from applications designed with the Applications Generator of dBASE IV.

In addition to creating dBASE program code from screen objects, template language is a text formatting language independent of dBASE IV. You can use the template language to create program code in other programming languages such as C or BASIC.

Template language was designed to translate the design objects created at the Control Center into dBASE program text files. dBASE IV has five design tools that use templates: the Applications Generator, the forms design screen, the reports design screen, the labels design screen, and QBE. Each of these design tools creates design object files. Templates created with the template language are used to translate the information saved in these design object files into working dBASE programs or documentation text files. QBE objects require special handling with the IMPORT( ) function to translate the object file to the proper form.

What is a template for a dBASE program? Basically, it is an abstraction of a working dBASE program into a generalized template for that program. For example, if you take a very simple dBASE program:

```
@ 5,10 SAY Vendname
@ 6,10 SAY Vend_Id
```

then use braces to delimit the variables in the program:

```
@ {5},{10} SAY {Vendname}
@ {6},{10} SAY {Vend_Id}
```

and finally replace the delimited variables with template language code:

```
{foreach FIELD_ELEMENT}
@ {ROW_POSITN}, {COL_POSITN} SAY {FLD_FIELDNAME}
{next}
```

you have converted a dBASE program into a template source code file.

Where you had specific screen coordinates, you now have special *selectors* capable of representing all possible screen coordinates. Also, there is nothing specific to dBASE IV in a template file. Everything inside the braces gets compiled by the template compiler, Dtc.exe. Everything outside the braces is passed on as text to the output file.

This ability to format and pass text means the template language can work with other programming languages, too. If you pass through constructs, statements, and variables of another programming language such as C as text, then the template language can output a source code text file that can be compiled in C.

The main function of the template language is to format and output text; its default data format is text. This is the opposite of dBASE programs where the majority of the statements are dBASE language statements and few areas of static text elements must be delimited to be printed. Thus, the template language provides text handling features that are not found in dBASE IV.

Why is it called a template language? The source code template resembles the generated program as much as possible. A few minor changes to a template can create extensive changes to the generated code. Thus, template language programs provide a template or a preset pattern for generated programs. Once you create a template, it will work with many dBASE files saving you hours of coding time.

Many template files with the extension .cod are included with the Developer's Edition of dBASE IV. You can modify these template files with a text editor, then use the template language compiler and interpreter to create your own versions of them. Or, you can write and compile your own templates.

Creating Templates

You create templates by writing ASCII text template program files with any editor. Then you compile these programs with the template compiler, Dtc.exe. The source files use the .cod extension by default. The compiler creates template files that have the default file extension .gen.

Six standard .gen files are included with dBASE IV for use with the four design tools. The source programs for these templates are included with the Template Language Toolkit. You can modify and recompile them to create customized versions for your own use. You can also study these programs to learn more about the template language. As_menu.cod and Ds_doc.cod, listed in Table 18-1, include many other template programs.

In Chapter 20, a special documentation template program called Code_doc.cod is described. You can run Code_doc.cod on any template program source file (.cod) to produce and print out a formatted program listing, including a table of contents of procedures and user-defined functions found in that source code. The Code_doc.cod program produces easy-to-read text for source code files.

Table 18-1 Standard templates

Design Tool	Template	Source Program
Forms	Form.gen	Form.cod
Labels	Label.gen	Label.cod
Reports	Report.gen	Report.cod
Applications Generator:		
Quick Application	Quickapp.gen	Quickapp.cod
Standard Application	Menu.gen	As_menu.cod
Documentation	Document.gen	Ds_doc.cod

NOTE

Dtc.exe and the stand-alone interpreter, Dgen.exe, are not designed to share files on a network. However, the templates you create with them can be used in a multi-user environment. You should develop and test templates in a single-user environment, and then copy them to the dBASE directory on the network.

Format of a Template Language Program

A template language program consists of an introduction, followed by template language commands inside of braces embedded into fixed text. You can place comments in the introduction and in the body of the program.

Introduction

The introduction to a template program includes all text that appears before the first template language command in the program, except for comments. It can be up to 4K, excluding comments. The compiler copies the introduction to the template file exactly as you enter it into the program, including indents, tabs, and line endings. When you use the DOS or dBASE IV TYPE command with a compiled .gen template file, only the introduction is displayed on the screen.

The introduction is a useful place to record the description and modification history of a template. You can TYPE the compiled program file Form.gen at the DOS prompt for an example.

Body

The body of a template program begins with the first template language command. Template language commands are enclosed in braces ({}) to distinguish them from the fixed text, so the beginning of the body is actually marked by the first left brace ({). Anything outside of braces (except for comments) is text that is copied unaltered to the output file.

Comments

Comments can be entered in the introduction and in the body. Comments begin with two slashes (//). Outside of braces, each comment line begins with a set of slashes in the first column and continues to the end of the line. Within braces, a comment can begin anywhere on the line, and continues to the end of the line or the closing brace, whichever comes first.

Embedding Commands in Text

Text outside of braces in a template language program is copied to the output file exactly as you enter it. Template language commands inside of braces are evaluated when the template is run, and may result in text that is also sent to the output file.

Indentation and line endings are important in the text of a template, especially if the output file is a dBASE program. Within braces, however, indentation and line endings are meaningless except inside of quotation marks.

A backslash character at the end of a line of text, outside of braces, suppresses the carriage return, thus joining the next line to the end of the current line. At any place but the end of the line, the backslash overrides any special meaning of the next character. For example, two backslashes (\\) are necessary in directory paths in template programs; the first backslash overrides the special meaning of the second backslash. The second backslash is output to a file as a text character.

Pending Values

The result of the last template language command in a set of braces is called the pending value. By default, the pending value is sent to the output file. The following example adds *menuct* and *fieldct,* and assigns the result to *itemct.* The value of *itemct* is the pending value that is sent to the output file between the words *are* and *items.*

```
There are {itemct = menuct + fieldct} items to process
```

Sometimes you do not want to send a pending value to the output file. To suppress the result, place a semicolon (;) before the closing brace:

```
Processing items...please wait.{itemct = menuct + fieldct;}
```

Compiling a Template

Dtc.exe compiles the readable form of a template file (a .cod file) into a form that the template interpreter can run (a .gen file). The compiler supports path names for the source file and for any files contained in the source program. To compile, either the template language source program and any files it includes must be in the current directory, or the filenames must include complete directory paths.

The syntax for the Dtc.exe command line is:

DTC -i <inputfile> [-o <outputfile>] [-l <listfile>] [-a] [-z]

The -i, -o, -l, and -a switches must be in lowercase. The space between the switch and the filename specification is optional.

The -i argument is required. This is the name of the input template program that you want to compile. You must enter the full filename, including the extension (usually .cod). Drive and path specifications are supported.

The -o argument is optional. It specifies a name for the compiled output (.gen) file. Drive and path specifications are supported. If you use this option, include the .gen extension with the filename. If you do not use the -o argument, DTC names the output template file with the same name as the input file and gives it a .gen extension.

The optional -l argument names a list file. Dtc.exe lists the source lines with line numbers to this file.

The -a argument, also optional, causes the compiler to list template language assembly instructions. When you combine the -a and -l options, the assembly instructions are written to the list file. If you use the -a option without the -l option, the assembly instructions are output to the screen.

The -z option reduces the template .gen file by 30 percent and increases the execution speed of the template. If the -z option is used, you cannot see your template source code in the debugger, because the line number information used to find the code is not output.

Use the -z option after a template program is completely debugged and is correct.

Running a Template

The template compiler creates intermediate code that must be interpreted by dBASE IV or the stand-alone template interpreter, Dgen.exe, included with the Template Language Toolkit. Use Dgen.exe to test templates before you integrate them into the dBASE IV environment, or to create stand-alone templates that do not use dBASE design objects.

Using Dgen.exe

The syntax for the DGEN command is:

DGEN -t <template.gen> [-i <obfile>] [-l <listfile>] [-n] [-p<parameters>]

The -t, -i, -l, -n, and -p arguments must be in lowercase. The space between the switch and the filename is optional.

The -t argument specifies the compiled template (.gen) that is to be interpreted. Include the extension with the filename.

Use the -i argument to list design object files that the template uses. If the template uses more than one object file, list them separated by commas. You can use the standard DOS wildcards * and ? to specify groups of files that match a pattern, for example *.app.

The -1 option creates a list file to receive a diagnostic listing.

If you include the -n argument, DGEN displays the options and filenames you enter on the screen.

The -p option followed by a delimited text string passes the parameters in the text string to the template program you are interpreting.

Design object files created with the Applications Generator (.app, .pop, .bar, and so on) are compatible with Dgen.exe. However, the design object files created from the forms, reports, and labels design screens (.scr, .frm, and .lbl files) must be translated to a format that is compatible with Dgen.exe using the DEXPORT command.

The DEXPORT command is explained later in this chapter. See *Language Reference* for the syntax of DEXPORT.

Using Templates with Environment Variables

The template interpreter in dBASE IV is activated when you save designs on the forms, reports, and labels design screens, and when you choose **Begin generating** from the Applications Generator **Generate** menu.

With the Applications Generator, you can enter the name of your own template by choosing **Select template** from the **Generate** menu. However, the forms, reports, and labels design screens have no place to enter a template name. There are two ways that you can execute your own templates from these screens:

- Replace the Form, Report, or Label .gen file provided by dBASE IV with your own .gen file
- Use the DOS SET command to create an environment variable that names your template

Table 18-2 lists the dBASE IV template (.gen) filenames and the DOS SET command you can use to substitute your own templates.

Table 18-2 Templates and DOS environment variables

Design Screen	Standard Template	DOS Environment Variable
Forms	Form.gen	SET DTL_FORM=<filename>
Reports	Report.gen	SET DTL_REPORT=<filename>
Labels	Label.gen	SET DTL_LABEL=<filename>

You may use drive and path specifications with the SET environment variable when you specify template filenames. For example, if you created a template to print envelopes named *Envelope.gen* on the C drive under the dBASE directory, you could run this template from the labels design screen by entering the following command from the DOS prompt before you start dBASE IV:

```
SET DTL_LABEL=C:\DBASE\ENVELOPE.GEN
```

If you always want to use your customized templates with the dBASE IV design tools, copy the original Form.gen, Label.gen, and Report.gen files to a diskette, or to backup names. Name your custom template files with these specific template filenames. dBASE IV will always use your templates, and you don't need to set DOS environment variables.

Using the DEXPORT command

It is possible to save design files to Binary Named List (BNL) files and to generate the BNL files from the dot prompt or in user programs. This functionality is provided by the DEXPORT command and the DGEN( ) function.

The DEXPORT command generates Binary Named List files. DEXPORT also allows flexibility in file extensions.

Design objects have distinct filenames with DEXPORT. The filename defaults to the same name as the design object and DEXPORT adds the following default extensions:

- SCREEN uses .snl
- REPORT uses .fnl
- LABEL uses .lbl

Any filename permitted by the operating system can be used instead.

Simple Template Programs

The first example, Simple.cod, shows the parts of a template language program and gives instructions for compiling and running the program. Simple.cod is a template that reads the current DOS directory and writes each filename to an output file with the default name of Text.out assigned to it by the interpreter.

The second example, Textpul1.cod, reads a simple screen form and extracts all text elements out of it.

The third example, Textpul2.cod, reads the same simple screen form and extracts both text and field information from it.

Template for Reading a Directory

Study the example to identify the introduction, body, and comments. Notice how braces are used to enclose template language commands, how comments are used both inside and outside of braces, and how commands are mixed with text.

```
// The introduction begins on the next line.
   Simple.cod -- first template
  A simple template to show the structure of a
   template language program.

  // The next line begins the body of the program.
   {
     INCLUDE "Builtin.def"     // include built-in functions
      var fcount;               // variable to hold count of
      fcount = 0;               //    files in directory
   }
   Files in this directory
   -----------------------
   {foreach DOSFILE}
     {++fcount, DOSFILE}
   {next;}
   -----------------------
   Listed {fcount} files
```

Compiling and Running Simple.cod

Follow these steps to compile and execute Simple.cod:

1. Using a text editor, create a file named Simple.cod in the template language directory. Enter the sample text exactly as shown into this file.

2. Compile Simple.cod with Dtc.exe from the DOS prompt at the template language directory:

   ```
   DTC -i Simple.cod
   ```

   If you see any error messages, make sure your copy of Simple.cod matches the example, then try the DTC command again. When the message **Compile complete (no errors)**. appears, DTC has created the Simple.gen template.

3. Use Dgen.exe to execute Simple.gen by entering the following:

   ```
   DGEN -t Simple.gen
   ```

   The result of interpreting Simple.gen with DGEN is a default text file named Text.out. When you display the contents of Text.out with the DOS TYPE command, you'll see a list of the filenames in the template language directory.

   The count this template returns is different from the count DOS returns, because DOS counts directories as files.

Extracting Elements From a Screen Object

In this example, the Textpul1.cod template reads a simple screen form and extracts all text elements out of it. It uses the same **foreach** construct that you saw in Simple.cod to traverse the object. Note that in this example, an output file for the text is created and uses the same name as the screen object with a .fmt extension.

Null Value Handling

In storing the physical representation of a screen object such as a form, the template language selectors do not store numbers with the value zero, or null strings. This means that screen objects that are placed at 0, 0 screen coordinates do not get a value of zero for beginning location. When no value is assigned by the selector, the template will not compile correctly because a null string is returned by the selector.

To have objects that are located at 0, 0 on the screen compile correctly, a UDF named nul2zero( ) is defined in the Builtin.def file. This UDF converts all null strings to zero, as shown in the Textpul1.cod file in the next section.

Creating the Template

Using a text editor, create a text file named Textpul1.cod. Enter the following text exactly:

```
This is a simple template that extracts all text elements from a form.
{
include "form.def"  //form selectors
include "builtin.def" // builtin function library
//Create the output file with an .fmt extension
CREATE(NAME +".FMT")
//Start a loop through only text (text_element) on the form surface
FOREACH TEXT_ELEMENT
}
@ {nul2zero(ROW_POSITN)}, {nul2zero(COL_POSITN)} SAY "{TEXT_ITEM}"
{
NEXT  // Get next chunk of text until no more
RETURN 0;
}
```

Now compile this file from the DOS prompt at the template language directory:

```
DTC -i Textpul1.cod
```

This program compiles to give you a template program named Textpul1.gen.

Creating a Screen Object

Now you need to create a design object to use with Textpul1.cod. Start dBASE IV. Put any database you like in USE, so that you can put some fields on the design object surface. The following example uses Orders.dbf from the sample files directory.

From the Control Center, go to the Forms generator and create a new form called Text. Or, if you prefer, from the dot prompt type:

```
CREATE SCREEN TEXT
```

In either case, a dBASE IV form design surface opens up. On the first line type This is text area 1. Press ↵.

Press **F5** and add the field Cust_id (or any field you like). Press ↵ twice. Type This is text area 2. and press ↵.

Press **F5** and add the field PO_NUMBER. Press ↵.

Press **F5** and add the field INVOICED. Press ↵.

Press **F5** and add the field DATE_TRANS. Press ↵ twice. Type This is text area 3. Press **Ctrl-End**.

Now you have a screen composed of text elements and field elements. See Figure 18-1.

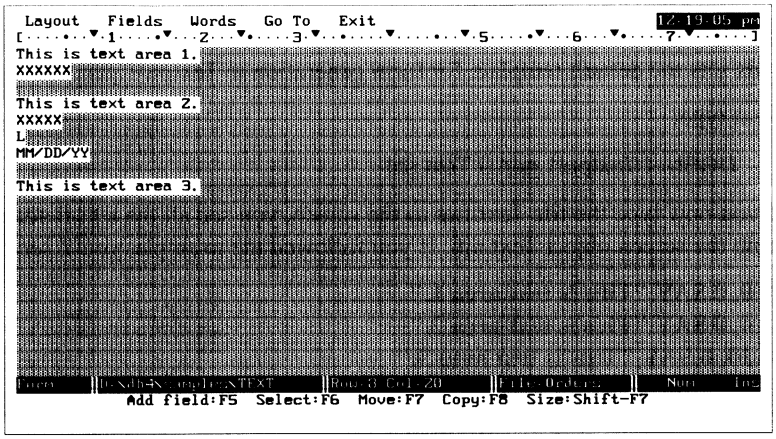


Figure 18-1 dBASE screen elements

When you press **Ctrl-End**, dBASE IV generates this screen.

Translating a Screen Object

The screen file created by dBASE IV must be translated into BNL object file format with the DEXPORT command before you can extract the text from it.

At the dot prompt type:

```
. DEXPORT SCREEN Text.scr
```

This creates a new file called Text.snl which can be used with both the DGEN( ) function from the dot prompt, or Dgen.exe from the DOS prompt.

To use the DGEN( ) function, at the dot prompt type:

```
. DGEN("TEXTPUL1","Text.snl")
```

This specifies the template you created at the beginning of this section, Textpul1, as the template file to use with the translated file Text.snl to generate the Text.fmt file. The contents of the Text.fmt file are shown in the next section.

Extracting Text from a Screen Object

If you want to use Dgen.exe from the operating system, rather than using the DGEN( ) function, then you need to set a DOS environment variable to specify your custom template Textpull.

Quit dBASE IV and return to the DOS prompt. Now type:

```
SET DTL_FORM=TEXTPUL1.GEN
```

This sets the default template for form generation to Textpul1.gen.

Next type:

```
DGEN -t Textpul1.gen -i Text.snl
```

If you now type the file Text.fmt to the screen, you see all the text fields extracted from the screen object Text.npi. If you followed the example, you'll see the following:

@ 0, 0 SAY "This is text area 1."
@ 4, 0 SAY "This is text area 2."
@ 10, 0 SAY "This is text area 3."

This template has ignored the fields you put on the screen and extracted only the text areas into the file that uses the name of the screen object, in this case Text, and adds the .fmt extension. Note that the nul2zero( ) UDF has returned zero values for the screen coordinates.

NOTE
If you had not changed the DTL_FORM default setting to Textpul1.gen, the standard template Form_gen would create an .fmt file that would contain all the elements from the design surface.

Now if you expand Textpul1.cod so that the FOREACH command uses CASE statements, you can handle both text and field elements.

Extracting Text and Field Elements from a Screen

Create a text file called Textpul2.cod. Enter the following text exactly:

```
A template for pulling out text elements and field elements.
{
include "form.def" //form selectors
include "builtin.def" // builtin function library
//Create the output file with a .fmt extension
CREATE(NAME +".FMT")
//Start a loop through for elements on the form surface
FOREACH ELEMENT
  case ELEMENT_TYPE of
  @TEXT_ELEMENT:
}
@ {nul2zero(ROW_POSITN)}, {nul2zero(COL_POSITN)} SAY "{TEXT_ITEM}"
{
  @FLD_ELEMENT:
}
@ {nul2zero(ROW_POSITN)}, {nul2zero(COL_POSITN)} GET {FLD_FIELDNAME}
{
  ENDCASE
NEXT //Get next chunk of text or next field until no more
RETURN 0
}
```

Now compile this file from the DOS prompt at the template language directory by typing:

```
DTC -i Textpul2.cod
```

This program compiles into the template program named Textpul2.gen.

You already have a translated screen form that has both fields and text elements from the Textpul1 example. You need to change the DOS variable for the form designer to the Textpul2.gen template. From the DOS prompt type:

```
SET DTL_FORM=TEXTPUL2.GEN
```

Now run this template to extract both text and field elements. The output .fmt file from this template will also be called Text.fmt, and it will write over the first Text.fmt you created with Textpul1.gen. If you don't want to overwrite Text.fmt, change this template file so that it names the output file something else. For example, the following change would name the output file Text1.fmt. Remember that you must recompile the .cod file after you make changes to it.

```
create(NAME +"1"+".FMT")
```

```
DGEN -t Textpul2.gen -i Text.snl
```

And if you type the file Text.fmt to the screen, you will see both text and field elements extracted and dBASE code written for the screen layout.

@ 0, 0 SAY "This is text area 1."
@ 1, 0 GET CUST_ID
@ 3, 0 SAY "This is text area 2."
@ 4, 0 GET PO_NUMBER
@ 5, 0 GET INVOICED
@ 7, 0 GET DATE_TRANS
@ 9, 0 SAY "This is text area 3."

This example showed how to write and then expand a simple template program.

If you run the dBASE IV standard template Form.gen with this screen form, you'll get the PICTURE elements extracted also. At the DOS prompt type:

```
DGEN -t Form.gen -i Text.snl
```

An abbreviated version of the Text.fmt file produced with the Form.gen template follows. The Form.gen template produces code that includes the PICTURE elements that were missing from the Textpul2.gen template.

```
*************************************************************************
*-- Name.......: TEXT.FMT *-- Date.......: 2-15-90
*-- Version....: dBASE IV, Format 1.5
*-- Notes......: Format files use "" as delimiters!
*************************************************************************
*-- Format file initialization code
*-----------------------------------------------
.
.
.
*-- @ SAY GETS Processing.
*-----------------------------------------------

*--  Format Page: 1
@ 0,0 SAY "This is text area 1."
@ 1,0 GET Cust_id PICTURE "XXXXXX"
@ 3,0 SAY "This is text area 2."
@ 4,0 GET Po_number PICTURE "XXXXX"
@ 5,0 GET Invoiced PICTURE "L"
@ 7,0 SAY Date_trans
@ 9,0 SAY "This is text area 3."

*-- Format file exit code
*-----------------------------------------------
```

Form.cod is a complete form template that can process all possible types of screen elements. Chapter 20 describes a special template Code_doc.cod, for printing out formatted code files with line numbers, table of contents, and page numbered headers. You might use that template to print out the Form.cod file, or other source code examples you want to look at in detail.

Running Templates from dBASE IV

You can generate program code from design objects and run Dgen without leaving the dBASE IV dot prompt.

Use the screen file Text.scr you created to use with Textpul1.cod, or create a new screen at the forms design screen. Or, use CREATE/MODIFY SCREEN command at the dot prompt. Save the design with **Ctrl-End**.

From the dot prompt type:

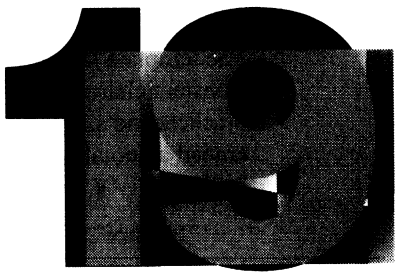
```
. DEXPORT SCREEN Text.scr
```

This creates the BNL file with the default name of Text.snl to use with DGEN( ).

```
.? DGEN("form","text")
  0
```

The zero returned indicates successful creation of the Text.fmt program file. If DGEN( ) returns a negative number, this number corresponds to the positive error code listed in Chapter 25 of this manual.

Template Language Expressions



The template language has data types, variables, operators, commands, and functions that you combine into expressions. This chapter describes these elements of the language and how to use them in programs.

Data Types

The template language supports three types of data: numeric, character, and logical.

Numeric Data Type

Numbers are 32-bit signed integers that range between – 2,147,483,650 and + 2,147,438,649. Floating point numbers are not supported.

Constant numbers can be entered directly into template language expressions. Start negative numbers with a minus sign (–). A plus sign (+) can be entered in front of a positive number, but is not required. You can enter hexadecimal (base 16) numbers by prefixing the hexadecimal digits with 0x or 0X. The hexadecimal digits A through F can be entered in uppercase or lowercase.

Following are some examples of valid numeric constants:

1022

+ 47

– 27335

0x1b

0x0F3B

Character Data Type

The template language character data type stores any series of up to 237 characters. Constant character strings are delimited with double quotation marks (""). Square brackets and single quotation marks are not supported in template expressions. To embed a quotation mark within a character string, precede the quotation mark with a backslash (\"). Here are some examples of character string constants:

```
"January"
"_pdriver = \"Hplas2i.pr2\""
```

Logical Data Type

The template language represents logical (true or false) data using the numeric or character data types. Numbers with a value of zero and empty character strings ("") are false when used in conditional expressions. Any non-zero number or non-empty string is true.

Names

Template language programs use symbolic names for data stored in memory and in design object files, the built-in functions, user-defined functions, and special variables called *cursors* that navigate lists of data. Some names are predefined in the language; others you declare in your programs.

A name in the template language is a sequence of 1 to 255 letters, digits, and underscores (_). The first character must be a letter or an underscore. Names are not case-sensitive, so *FILE_PTR* is equivalent to *File_Ptr*.

Memory Variables

Template language programs use memory variables to store working data. You must declare memory variables with the VAR command before you can use them. VAR defines a name for a variable, and assigns it a null value. The data type and value are determined when you store data to a variable, and they can change throughout the execution of a template.

A program can have any number of VAR commands, and each can list one or more variable names, separated with commas. VAR can appear anywhere in a program, but must occur before any of the variables it declares are used. As with all template language commands and expressions, VAR can be spread over two or more lines. Following are some examples of VAR commands.

```
{VAR counter;}              // declare a single variable
   {VAR item1, item2, item3;} // declare three variables
   {VAR list_name,
      list_type,            // declare three more variables
      list_items;
}
```

Enumerated Variables

Enumerated variables have a constant value throughout the execution of a template. They are declared with the ENUM command, which lists the name and value for each variable.

The following ENUM command creates twelve variables for the months in the year:

```
{ENUM January = 1,   February = 2,   March     = 3,
      April   = 4,   May      = 5,   June      = 6,
      July    = 7,   August   = 8,   September = 9,
      October = 10,  November = 11,  December  = 12;}
```

If you omit the assignment part of a declaration, ENUM assigns the next sequential value. Because the month numbers are sequential, the assignments can be omitted from the previous example. However, the assignment for January is necessary in order to start numbering at 1 instead of 0:

```
{ENUM January = 1, February, March, April, May, June,
      July, August, September, October, November, December;}
```

You can reset the counter in an ENUM command, as demonstrated by the following example. The comments show the values that are assigned.

```
{ENUM Payroll_dept = 1001,  // Department 1001
      AcctsRec_dept,        // Department 1002
      AcctsPay_dept,        // Department 1003
      Exec_dept = 2001,     // Department 2001
      HR_dept,              // Department 2002
      MIS_dept;             // Department 2003
}
```

You can also declare character strings in an ENUM command, but you must reset the counter when changing from character variables to numeric variables:

```
{ENUM c_standard = "W+/B",
      c_enhanced = "N/BG",
      status_ln  = 22,
      msg1_ln,              // line 23
      msg2_ln               // line 24
}
```

Selectors

Selectors are names in the template language that return values from the built-in functions, data stored in design object files, and filenames in a disk directory.

The SELECTORS command declares selectors by assigning names to numbers called item identification (IID) numbers. IID numbers are instructions that the template interpreter acts upon. They are fixed in the implementation of the template language and cannot be changed.

The .def files in the Template Language Toolkit define all the available selectors, so you do not need to declare them. You will need to study these files to learn the names of the selectors. Extensive comments explain which data is returned by the selectors.

Use the INCLUDE command to include .def files in your programs. Builtin.def should be included in all template language programs. It declares selectors for the built-in functions and special data items. Templates that access design object files must include an appropriate .def file for the design type. Table 19-1 lists the .def files that declare selectors for each type of design object file.

Table 19-1 Definition files for design objects

Object	.def file
Form (.scr)	Form.def
Report (.frm)	Report.def
Label (.lbl)	Label.def
Applications Generator (all objects)	Applctn.def
Import objects (.dbf,.mdx.qbe,.upd)	Schema.def

Templates generally include two .def files: Builtin.def, and the .def file from Table 19-1 that is required for the design type the template processes. You cannot combine Form.def, Report.def, Label.def, Applctn.def, and Schema.def in a single template program because they share common IID numbers. Schema.def contains the selectors for objects translated with the IMPORT( ) function. QBE objects require processing with the IMPORT( ) function to convert their format to a template language layout object.

You can add new selector names to existing IID numbers by changing the definitions in the .def files. For example, in the Builtin.def library file, RTRIM( ) is assigned to IID 312. This IID instructs the template interpreter to trim trailing blanks from a character string. If you change RTRIM( ) to TRIM( ) by editing the Builtin.def file, you will get compilation errors in existing template language programs. Instead, add a new selector declaration and assign TRIM( ) to IID 312. This creates a synonym for RTRIM( ) instead of renaming it.

The loop selectors for each of the .def files are listed in Table 19-2.

Table 19-2 Definition of loop selectors for .def files

def file	Loop Selector	IID #	Description
Form.def	ELEMENT	1000	All element types by row & column
	BOX_ELEMENT	1086	Box element
	FLD_ELEMENT	1130	Field element
	TEXT_ELEMENT	1085	Text element
Report.def	ELEMENT	1000	All element types by row & column
	BAND_ELEMENT	1102	Report band element
	BOX_ELEMENT	1086	Box element
	FLD_ELEMENT	1130	Field element
	TEXT_ELEMENT	1085	Text element
	RULER_ELEMENT	1119	Ruler element
	PARA_ELEMENT	1118	Paragraph end marker element
	PAGE_ELEMENT	1117	Page break marker element
Label.def	ELEMENT	1000	All element types by row & column
	BAND_ELEMENT	1102	Band element (always first element)
	FLD_ELEMENT	1130	Field element
	TEXT_ELEMENT	1085	Text element
Schema.def	ELEMENT	1000	All element types by row & column
	BAND_ELEMENT	1102	QBE Skeleton element
	FIELD_ELEMENT	1130	Field element
	TAG_ELEMENT	1119	Index Tag element
	FRAME_TEXT_ELEMENT	1175	Element FRAME_TEXT_ELEMENT

(continued)

Table 19-2 Definition of loop selectors for .def files *(continued)*

def file	Loop Selector	IID #	Description
Applctn.def	ELEMENT	1000	All element types by row & column
	BOX_ELEMENT	1086	Box element
	FLD_ELEMENT	1130	Field element
	TEXT_ELEMENT	1085	Text element
	TREE	0330	Tree Element

Accessing the Selector IID in a Program

Sometimes it is necessary to find a selector's IID number instead of the data it normally returns. For example, using an IID number as an argument in a user-defined function (UDF) makes it possible to write more generalized and useful UDFs.

Prefixing a selector name with an at sign (@) returns the IID number of the selector. The following example displays the data returned by a selector and the selector's IID:

```
The selector name is: Menu_Name
The value returned is: {MENU_NAME}
The identification number is: {@MENU_NAME}
```

The IIDC( ) function returns the identification number of an item pointed to by a cursor.

Cursors

Cursor variables are used in template language programs to navigate lists of data. Here are some examples of data lists that cursors can navigate:

- Fields on a format screen
- Lines of help text for a pop-up menu selection
- Filenames in a directory

Lists in Design Object Files

Design object files store specifications for screen forms, reports, labels, and the various objects you can create with the Applications Generator. The information is stored as lists of *elements* and *attributes.*

An element is the specification for a single item in a design, for example, a field on a format screen. Attributes describe an element in detail. A field element has attributes that tell whether the field is a database field or a calculation, where it is placed, how it is formatted, and so on.

Design object files store a list of elements. The first element in the list is the *frame element.* The attributes of this element provide information about the design as a whole, such as the design type and the number of elements it contains. Selectors that return data from the frame element are available at all times in a template language program.

After the frame element, the list can contain any number of elements that specify various items in the design, such as fields, boxes, and text items. Each type of element has a different set of attributes. Selectors are defined in the .def files for all possible element types and attributes.

Accessing Elements with Cursors

Cursors keep track of the active element in a list. Selectors that return attributes get their data from the element addressed by a cursor. Cursors are usually declared as a part of a FOREACH command, but you can create them explicitly with the MAKEC( ) function.

FOREACH...NEXT is a looping construct that executes a set of template language commands for each member of a list. The FOREACH command selects the members of the list and, optionally, names a cursor variable.

The syntax for the FOREACH command is:

FOREACH <loop selector> [<cursor var>] [IN <cursor var>]
<commands>
NEXT [<cursor var>]

The <loop selector> specifies the elements or attributes to process. The .def files define loop selectors for each type of element that appears in a design. For example, a FOREACH loop that processes boxes on a format screen would use the BOX_ELEMENT selector defined in Form.def. The following example shows how to establish this cursor:

```
{INCLUDE "Form.def";
INCLUDE "Builtin.def";
FOREACH BOX_ELEMENT}
   <commands to process box attributes>
{NEXT}
```

The <cursor var> is an optional name for the cursor. A cursor name is used to pass the cursor as an argument to a user-defined function or to qualify attributes that belong to a cursor. Use the dot operator (.) to separate a cursor name from a selector. The dot operator treats the expression on the left as a cursor and the expression on the right as a selector or integer. For example, to reference the row position of an element addressed by a cursor named Field_ptr:

```
Field_ptr.ROW_POSITN
```

The IN <cursor name> clause of the FOREACH command nests a cursor within an existing cursor. The inner cursor processes a list with members that are attributes of the element addressed by the outer cursor. The following example uses nested cursors to print descriptions of the items on each menu in an application.

```
{INCLUDE "Applctn.def";
 INCLUDE "Builtin.def";
 FOREACH TREE menu}
Menu Name: {MENU_NAME}
{   FOREACH FLD_ELEMENT flds IN menu}
   Bar {flds} of {menu.MENU_NAME}: {ACT_SUMRY}
{   NEXT flds
 NEXT menu;}
```

In the example above, the second reference to the MENU_NAME attribute is qualified with the menu cursor name. When there is more than one active cursor, the template searches from the innermost cursor to the outermost cursor until it finds an element with the attribute. Qualifying MENU_NAME in this example overrides the default attribute search. The interpreter retrieves the menu name from the outer cursor.

Operators

Operators work on data values to produce modified values. The template language has many operators. Several of the operators are not found in the dBASE language, but will be familiar to C programmers.

Operators are applied to operands. Operands can be constant values, memory variables, values returned by selectors or functions, or expressions composed of any of these elements.

Operands can be numeric, character, or logical expressions. The operator produces a new value with a data type defined by the operator and, in some cases, by the data types of the operands.

Unlike dBASE IV, the template language does not produce a **Data type mismatch** error when operands are the wrong data type for an operator. Instead, the template language converts operands to the data types required by the operator. You can use the VAL( ) and STR( ) functions to explicitly convert data types.

Assignment Operators

Assignment operators store values to variables. The dBASE language has just one assignment operator: the equal sign (=). The template language supports the equal sign and two additional assignment statements that include the + and – operators.

The assignment operators are listed in Table 19-3. The table includes an example of each operator and the equivalent form using the equal sign operator.

Table 19-3 Assignment operators

Operator	Operation	Example	Equivalent
=	assign	a = b	a = b
+ =	add and assign	a + = b	a = a + b
– =	subtract and assign	a – = b	a = a – b

Assignment operators must have a variable name on the left. A feature of the template language is that you can chain assignments together, as in the following example:

```
{var a, b, c; a = b = c = 1;}
```

This expression assigns 1 to c, b, and a.

Numeric Addition and String Concatenation Operator

The plus sign (+) performs numeric addition or string concatenation. It takes two numeric or character operands. If both operands are numeric, the result is the sum of the operands. If both operands are character, the result is the second operand concatenated (joined) to the first operand. If one operand is numeric and the other character, the template language concatenates the number to the character string. The following template example illustrates the outputs of the + operator:

```
{}
Number and number: {6 + 4}
String and number: {"6" + 4}
Number and string: {6 + "4"}
String and string: {"6" + "4"}
```

This template produces the following text output file:

```
Number and number: 10
String and number: 64
Number and string: 64
String and string: 64
```

Numeric Subtraction and Substring Delete Operator

The minus sign (–) calculates the difference between the first and second of two numeric operands. If both operands are character, the minus sign removes the first instance of the second character string from the first. If the second character string is not a substring of the first, the first operand is returned unaltered. If the operands are mixed numeric and character data types, the template language converts the character expression to a number and calculates the difference.

The following template example illustrates the outputs of the – operator:

```
{}
Number and number: {4 – 6}
String and number: {"4" – 6}
Number and string: {4 – "6"}
String and string: {"64" – "4"}
```

This template produces the following text output file:

```
Number and number: –2
String and number: –2
Number and string: –2
String and string: 6
```

Numeric Multiplication and String Concatenation Operator

The asterisk (*) operator computes the product of two numeric operands. If both operands are character strings, the result is the second string concatenated to the first, just like the plus sign operator. When the operands are mixed data types, the result is determined by the data type of the first operand. If the first operand is character, the second is converted to a character string and concatenated to the first. If the first operand is numeric, the second is converted to a number, and the product is returned.

The following template example illustrates the outputs of the asterisk operator:

```
{}
Number and number: {3 * 5}
String and number: {"3" * 5}
Number and string: {3 * "5"}
String and string: {"3" * "5"}
```

This template produces the following text output file:

```
Number and number: 15
String and number: 35
Number and string: 15
String and string: 35
```

Numeric Division Operator

The slash (/) calculates the difference between two numeric operands. Character operands are converted to numbers before the calculation is performed. Because the template language does not support floating point operations, the result of the division is the integer part of the quotient. The remainder is lost, but can be calculated by applying the modulus operator (described next) to the same operands.

The following template example illustrates the outputs of the division operator:

```
{}
Number and number: {10 / 3}
String and number: {"10" / 3}
Number and string: {10 / "3"}
String and string: {"10" / "3"}
```

This template produces the following text output file:

```
Number and number: 3
String and number: 3
Number and string: 3
String and string: 3
```

Modulus Operator

The percent sign (%) calculates the remainder after dividing the second operand into the first. Character operands are converted to numbers before the calculation is performed. The result is always a number.

The following template example illustrates the outputs of the modulus operator:

```
{}
Number and number: {10 % 3}
String and number: {"10" % 3}
Number and string: {10 % "3"}
String and string: {"10" % "3"}
```

This template produces the following text output file:

```
Number and number: 1
String and number: 1
Number and string: 1
String and string: 1
```

Comparison Operators

The comparison operators, listed in Table 19-4, compare two numeric or string operands. The result is 0 if the comparison is false and 1 if the comparison is true.

Table 19-4 Comparison operators

Operator	Operation
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
>	greater than
>=	greater than or equal to

NOTE
Remember to use the double equal signs for equality. The single equal sign signifies assignment in template language. Using the single equal sign will result in errors in your template programs.

The type of comparison performed is determined by the data type of the first operand. If the first operand is numeric, the second operand is converted to a number, if necessary, and the values of the two are compared. If the first operand is character, the second is converted to character, if needed, and the strings are compared character-by-character.

The template language does not have a counterpart to the dBASE SET EXACT command to alter the way that character strings are compared. Strings are compared as if SET EXACT is ON, except that trailing spaces are also significant. The template language considers two strings equal if they are exactly the same length and are composed of exactly the same characters, including trailing spaces. See the AT( ) function for doing substring comparisons.

Logical Operators

Logical operators take one or two logical operands and produce a logical result. There are three logical operations: AND, OR, and NOT. These operators are not case sensitive. There are two synonymous operators for each of the operations, as listed in Table 19-5.

Table 19-5 Logical operators

Operator	Operation
!, NOT	Logical NOT
&&, AND	Logical AND
||, OR	Logical OR

In the template language, a logical true value is a character string with one or more characters, or a non-zero number. A logical false value is an empty character string ("") or a zero. The logical operators return 1 for true and 0 for false.

The logical AND operators (&& and AND) produce a true value if both operands are true. The logical OR operators (|| and OR) produce a true value if one or both of the operands is true.

The logical NOT operators (! and NOT) produce the opposite of a single operand. If the operand is true, ! and NOT return false. If the operand is false, ! and NOT return true.

Increment and Decrement Operators

The increment and decrement operators of the template language are allowed only as prefix operators.

The + + increment operator adds 1 to the variable following it. The – – operator subtracts 1 from the variable following it. The output is a numeric value. If you use these operators as a prefix for a text string, the text is treated as a number, and the output is 1 or – 1.

Increment and decrement operators are best used for moving the cursor variable to the next item or the preceding item in the cursor's set.

Table 19-6 shows the increment and decrement operators and the equivalent assigment using the equal sign alone.

Table 19-6 Increment and decrement operators

Operator	Operation	Example	Equivalent
+ +	increment,	+ + a	a = a + 1
– –	decrement	– – a	a = a – 1

Ternary Operator

The template language ternary operator (?:) is similar to the dBASE IIF( ) function. It takes three operands. The first operand is a logical expression, and the second and third operands can be character, numeric, or logical. The form of the ternary operator is:

<exp1> ? <exp2> : <exp3>;

If <exp1> is true, the result is <exp2>. If <exp1> is false, the result is <exp3>. The following example uses the ternary operator to output a string or the word *null* if the string is empty:

```
{m_item ? m_item : "null"}
```

Bit Manipulation Operators

The template language provides operators that manipulate bits within numbers. Template language numbers are stored in 4 bytes and have 32 bits. Each bit can have a value of 1 or 0. When the value is 1, the bit is said to be *on* or *set.* When the value is 0, the bit is *off* or *cleared.* All of the bit operators take two numeric operands, except for the one's complement operator, which inverts the bits in a single operand. The bit manipulation operators are listed in Table 19-7.

Table 19-7 Bit manipulation operators

Operator	Operation
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
<<	Shift bits left
>>	Shift bits right
~	One's complement

Bitwise AND Operator

The bitwise AND operator (&) performs a logical AND operation on two numbers. Each bit in the first operand is compared to the corresponding bit in the second operand. If both bits are 1, the corresponding bit is set in the result. If either of the bits is 0, the corresponding bit is reset in the result. The result is a number with bits set for each bit that is set in both of the operands.

Bitwise OR Operator

The bitwise OR operator (|) performs a logical OR operation on two numbers. Each bit in the first operand is compared to the corresponding bit in the second operand. If either bit is 1, the corresponding bit is set in the result. If both bits are 0, the corresponding bit is reset in the result. The result is a number with bits set for each bit that is set in either of the two operands.

Bitwise XOR Operator

The bitwise XOR operator (^) is a combination between the bitwise AND and OR operators. It performs an exclusive OR operation on two numeric operands. A bit in the result is set when one, but not both, of the bits in the operands is 1. If the bits in the operands are the same (both 1 or both 0) the bit in the result is reset.

Bitwise Shift Operators

The bitwise shift operators (<< and >>) shift the bits in the first operand left or right the number of positions specified by the second operand. The vacated bits are reset to zero.

Shifting bits left is the same as multiplying the number by successive powers of two. For example, *5<<1* is the same as *5*2* (10), and *5<<4* is the same as *5*2*2*2*2* (80). Similarly, shifting bits right is the same as dividing by powers of 2.

NOTE
The bitwise shift left operator (<<) preserves the sign bit of the shifted value. Shifting too far left (numeric overflow) results in a 0 value. Shifting too far right (numeric underflow) results in a –1 for negative numbers, and a 0 for all other values.

The One's Complement Operator

The one's complement operator (~) is a unary operator that complements (reverses) each bit in its operand. The operator is placed in front of its operand. For example, *~0* yields *– 1*, which has all 32 bits set to 1.

Precedence of Operators

Within an expression unary operators are evaluated first. If more than one unary operator appears in sequence (such as ! –a), evaluation of the sequence is from right to left.

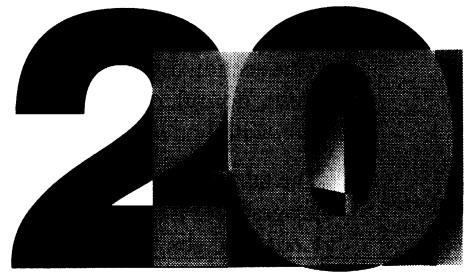
After unary operators are evaluated, binary operators are evaluated in order of precedence, highest precedence first. Among operators of equal precedence, evaluation is from left to right.

The precedence of operators is given in the following table. Operators on the same line have the same precedence. See Table 19-8.

Table 19-8 Precedence of Operators

Operator							
!	NOT	~	+	–	++	––	(all unary)
*	/	%					
=	–	(binary)					
<<	>>						
<	<=	>	>=				
==	!=						
&							
|	^						
&&	AND						
||	OR						
?:							
=	+=	–+					

A Source Printing Template



Introduction

This chapter describes a special template program, Code_doc.cod. This template can be used with any template source code file (.cod or .prg) including itself, and produces a formatted code listing. It provides line numbering, bolding, required page breaks, and long line wrap to current indentation level. It uses the line numbers to create a table of contents for all procedures and user-defined functions (UDFs) contained in the source file. This is a tool for documenting all template program source files.

An electronic copy of the source template for the Code_doc.cod file is located in the Samples subdirectory. Copy this to the template language directory or use the complete file path to run it. This template can be used only with the stand-alone interpreter Dgen.exe.

First select a template code file which you want to print as formatted text. This can be any source code file from the template language directory such as Form.cod or Ds_doc.cod.

Then make sure that the printer you are connected to is on line. This template supports HP LaserJet and Epson. If your printer is neither of these and does not support Epson emulation, you'll need to edit the printer control codes in Code_doc.cod before you can produce a satisfactory printout. Changing printer control codes is explained before line 211 of the template in this chapter.

Change directories to the template language directory. From the DOS prompt, type:

```
DGEN -t Code_doc.gen
```

You will be prompted for the template code file you want to document. Enter the name of a .cod file. Depending on the size of the template you are documenting, the program will run for a few seconds to a few minutes. It will create a file with .otl as the extension. This .otl file is a printer formatted file; it contains embedded printer control sequences, hard page break codes, and headers with page numbers at the top of each page. Next, the template prints a formatted document using the .otl file.

Code_doc.cod File

This section discusses the modules of the Code_doc.cod file. The formatted file Code_doc.otl, obtained by running Code_doc.cod on itself, is used as the printed version with line numbers. The source files do not have line numbers.

The table of contents was generated by using the Code_doc.cod template as the input program for itself. The header and page numbering is also done with the template Code_doc.gen.

```
2-07-90  1:04p                    CODE_DOC.COD            Page: 1

Table of Contents for Procedures of code_doc.cod
------------------------------------------------------------------------------
cod_proc()                                                          186
db_proc()                                                           203
page_Header()                                                       124
printer_setup()                                                     211
print_cod_file()                                                    158
print_table_of_contents()                                           135
process_table_of_contents()                                          82
sort_table_of_contents()                                            112

Source Follows:
```

Next, the program code begins. Three comment lines (two of them blank) are followed by the program header which is passed through unchanged. The program header section can be up to 4K, and is a good place to put the program name, purpose, and revision information. When you execute this template with DGEN, the header and notes are displayed on the first screen that comes up. Notes explain the usage of this template.

```
1    //
2    // Module Name: CODE_DOC.COD
3    //
4    Program Formatter and Procedure Table of Contents Builder
5    ---------------------------------------------------------
6
7
8
9    Notes:
10
11
12
13   - This template MUST only be used with the standalone DGEN.EXE (template
14     interpreter). Syntax  C>dgen -t code_doc.gen
15
16   - This program makes a call to the DOS SORT.EXE program. You should make
17     sure that your DOS PATH statement contains the directory where
18     SORT.EXE resides.
19
20   - For printer formatting codes, only Epson & HP LaserJet II are supported!
21     You could add your own printer codes to the "printer_setup( )" udf at the
22     bottom of this template.
23
24   - The output of this template is a ".otl" file (for outline) with the same
25     root file name. For example, Form.cod outputs Form.otl after it is run
26     through this template
```

The first curly brace starts the template program. The Builtin.def file contains the template language function library. It must be included in all template language programs. Next, variables separated by commas are declared as a group.

```
{
 include "builtin.def";  // Builtin Functions

 var cod_filename, temp, line_counter, crlf, laser, page, page_length,
 sort_column, sort_string, start_print, end_print, set_printer, reset,
 first time, total lines;

```

Then values are assigned to some of the defined variables. These can be numeric values or values returned by template language functions, as is the case with the cod_filename variable which is assigned to the filename returned by the function ASK_USER( ).

```
 crlf = chr(10)
 line_counter = 0
 page = 1
 first_time = 1
 cod_filename = askuser(" Enter program name to make Outline for
 (required) ",cod_filename,12)
 laser = " "
 do while at( upper(laser), "YN") == 0
    laser = upper(askuser(" Is report to be sent to a HP Laser? (Y/N)","Y",1))
 enddo

```

Depending on the value returned by the ASK_USER( ) function, the user-defined function Printer_setup( ) inserts the correct printer control codes into parts of the formatted output file. Printer_setup( ) starts on line 211 of this printed version. You can edit this UDF in the Code_doc.cod file to create a custom version of Code_doc.cod with control sequences for any printer.

The template program checks to make sure the .cod file you named exists, then opens a temporary file to write the values returned by the user-defined function Process_table_of_contents( ). Next it runs a DOS sort on the table of contents list, and prints the sort output to a temporary file.

```
 Printer_setup()

 // Determine column number for the DOS SORT program
 sort_column = fileroot(upper(cod_filename)) == "PRG" ? "10" : "1";
 page_length = 78

 if (!cod_filename or !Textopen(cod_filename)) then goto nogen;

 if !create(fileroot(cod_filename)+".tml") then goto nogen; endif

```

The UDF on line 58 sorts the table of contents produced in line 52 above and provides a special page header for it.

```
54    Process_table_of_contents()
55
56    cls()
57
58    Sort_table_of_contents()
59    Page_Header();
60   }
61
62
63   Table of Contents for Procedures of {cod_filename}
64
-------------------------------------------------------------------------------
65
```

The UDF beginning on line 67 prints the sorted table of contents.

```
66   {
67    Print_table_of_contents();
68   }
69
```

The table of contents is printed out as page 1. Next, the template prints out the entire .cod file.

```
70   Source Follows:
71   ---------------
72
73
74   {
75    Print_cod_file()
76    nogen:
77    return 0;
78
```

The next sections of the code contain the user-defined functions in this template. These can be copied and used in other template programs, provided you make sure that any variables that are called in the UDF are also declared in the new template.

```
//-------------------------------------------------------
79   // UDF's follow
80
//-------------------------------------------------------
81
```

The UDF beginning on line 82 builds the table of contents.

```
82  define process_table_of_contents()
83   cls()
84   say_center(13, "Procedure found:")
85   do
86      temp = textgetl();
87      while temp != eof
88          line_counter = line_counter + 1
89
90          nmsg("Processing line: " + alltrim(str(line_counter)))
91
92          if db_proc() or cod_proc() then
93
94             say(15, 0, space(80))
95             say(16, 0, space(80))
96             say_center(15, temp)
97
98             pmsg("Found procedure at line: " + alltrim(str(line_counter)))
99             if cod_proc() then
100                    temp = substr(temp,at("DEFINE ",temp)+8)
101                    print(alltrim(substr(temp,1,73)))
102            else 103             print(alltrim(substr(temp,1,73)))
104            endif
105            tabto(75)
106            print(line_counter + crlf)
107         endif
108  enddo
109  total_lines = line_counter
110 enddef
111
```

Next is the *sort_table_of_contents( )* UDF that sorts the temporary file created with *process_table_of_contents( ),* and pipes the sorted output to a second temporary file. This UDF also creates the .otl file for the final output from Code_doc.cod.

```
112 define sort_table_of_contents()
113  // Prepare to sort the Table of contents
114  create(fileroot(cod_filename)+".tm3")
115  sort_string = "SORT /+" + sort_column + " <" +
116                fileroot(cod_filename) + ".tm1 >" +
117                fileroot(cod_filename)+".tm2"
118  // Execute the DOS SORT program
119  exec(sort_string)
120  create(fileroot(cod_filename)+".otl")
121  print(reset + set_printer + crlf)
122 enddef
123
```

The next UDF is *page_Header( )* which prints the date, the .cod filename being documented, and the page number. You could easily edit one line in the .cod file and personalize this header. The existing line 131 is:

```
131     print("Page: " + alltrim(str(page)) + crlf + crlf)
```

Change it to something like:

```
131     print("Page: " + alltrim(str(page)) + crlf +"Written by Your Name for
Your Company")
```

This text string replaced a carriage return, line feed (crlf) character in the original template. The added text will print on the next line after the page number. You can leave the CRLF in. The added text can be anything you like, such as your name or an expanded title for the .cod file.

The page_Header( ) UDF begins on line 124. Notice that the header UDF ejects a blank page before starting the printing and the page numbering.

```
124  define page_Header()
125     pageject()
126     first_time = 0
127     print(alltrim(date()))
128     tabto(35)
129     print(upper(cod_filename))
130     tabto(70)
131     print("Page: " + alltrim(str(page)) + crlf + crlf)
132     return
133  enddef
134
```

The UDF beginning on line 135 prints the sorted table of contents.

```
135  define print_table_of_contents()
136    // Process sorted file so that it has page header information
137    textopen(fileroot(cod_filename)+".tm2")
138    line_counter = 0
139    pmsg("Processing Sorted Table of contents")
140    do
141        temp = textgetl();
142        while temp != eof
143            if curline() > page_length then
144               page += 1
145               Page_Header()
146            endif
147            line_counter = line_counter + 1
148            nmsg("Processing line: "+alltrim(str(line_counter)))
149            print(alltrim(temp) + crlf)
150    enddo
151    textclose()
152    exec("DEL " + fileroot(cod_filename) + ".tm? > nul")
153    // Start source listing on next page
154    page += 1
155    Page_Header()
156  enddef
157
```

The UDF beginning on line 158 counts the lines and assigns line numbers to the template file and page numbers on the page headers.

```
158 define print_cod_file()
159   Textopen(cod_filename)
160   nmsg(" ")
161   pmsg("Adding "+ cod_filename + " w/line numbers to the bottom of the outline")
162   line_counter = 0
163   do
164      temp = textgetl();
165      while temp != eof
166          if curline() > page_length then
167             page += 1
168             Page_Header()
169          endif
170          line_counter = line_counter +1
171          print(line_counter)
172          tabto(5)
173
174          if db_proc() or cod_proc() then
175              print(" "+substr(temp,1,at(" ",temp)))
176              print(start_print +
177                    rtrim(substr(temp,at(" ",temp)+1)) + end_print + crlf)
178          else
179              print(" "+rtrim(temp)+crlf)
180          endif
181   enddo
182   textclose()
183   print(reset + pageject())
184 enddef
185
```

The *cod_proc( )* procedure begins on line 186. This procedure checks to see if you are processing a .cod file. If it finds a define command, the template checks to make sure it is a template language define and not a dBASE DEFINE command. If present, then the names of these procedures would be printed in the table of contents by the process_table_of_contents( ) UDF.

```
186 define cod_proc()
187    if ( ( at("DEFINE ",upper(temp)) >= 1 and at("DEFINE",upper(temp)) <= 6)
188                and (at("*", upper(temp)) != 1 or at("NOTE ",upper(temp)) != 1)
189                and at("BOX ", upper(temp)) == 0
190                and at("WINDOW ", upper(temp)) == 0
191                and at("POPUP ", upper(temp)) == 0
192                and at("BAR ", upper(temp)) == 0
193                and at("PAD ", upper(temp)) == 0
194                and at("MENU ", upper(temp)) == 0
195                and fileroot(upper(cod_filename)) != "PRG"
196       ) then
197              return 1
198      else
199              return 0
200      endif
201 enddef
202
```

The *db_proc( )* procedure begins on line 203. This procedure is similar to cod_proc( ) above on line 186. This one looks for the presence of dBASE procedures, or user-defined functions in the code template. If present, then the names of these routines, prefixed by the uppercase word FUNCTION or PROCEDURE, are printed in the table of contents by the process_table_of_contents( ) UDF.

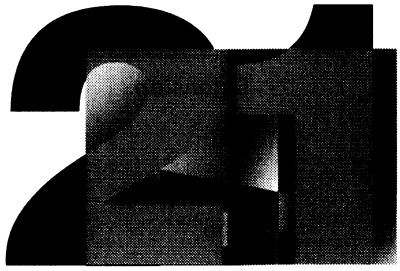
```
203  define db_proc()
204     if at("PROCEDURE ", upper(temp)) == 1 or at("FUNCTION ", upper(temp)) == 1 then
205           return 1
206     else
207           return 0
208     endif
209  enddef
210
```

The *printer_setup* function begins on line 211. The two options are LaserJet II and Epson. If you need to use a different printer, edit the Code_doc.cod file and type in the correct codes for your printer. If you have neither the LaserJet, nor the Epson, the easiest change would be to change the codes for the Epson, and always respond N to the screen prompt **Is report to be sent to a HP Laser**? Then you would get the printer you substituted for the Epson. The printer control codes are found in the manuals shipped with that printer. Make sure that the font sizes you choose are equal to the ones in this template. You may have to test the output with a few different fonts until you get it right. The page breaks may walk with larger fonts.

In the example below, the letters Esc represent the Escape character (ASCII 27). The display of the Escape character depends on the text editor you use.

```
211  define printer_setup()
212    if laser == "Y" then
213       // HP LaserJet II
214       reset == "Esc E" // Reset
215       set_printer = "Esc &k2G Esc &a5L Esc &k4S Esc &l8D"
//        Line Term, 5 char left margin, 8 lpi, and 12 CPI
216       start_print = " Esc &k0S"+" Esc (s3B" // Bold 10cpi
217       end_print   = " Esc &k4S"+" Esc (s0B" // Line print  12cpi
218    else
219       // Epson
220       reset = " Esc @" // Reset
221       set_printer = " Esc l Esc 0 Esc // 5 char left margin, 8lpi, Elite
222       start_print = "Esc W1 Esc E"    // Expanded, Bold
223       end_print   = " Esc W0 Esc F  Esc M" // Off Expanded, Off Bold, 12cpi
224    endif
225  enddef
226  }
```

Commands



This chapter describes the commands used in the template language.

Commands are explained in functional groups, such as compiler commands, definition commands, program flow commands, and loop commands. Under each functional group, the most important command is explained first.

Command Syntax

The symbols and conventions used for writing the command syntax of the template language are listed in Table 21-1.

Table 21-1 Symbols and conventions

Symbol	Use
< >	Angle brackets indicate items for which you must type in a value. Do not type the angle brackets.
[ ]	Square brackets indicate an optional item. Do not type the square brackets.

File Naming Conventions

Filename conventions are the same as in DOS. Filenames can be up to eight characters long. They must begin with a letter and cannot contain embedded blank spaces. Letters, numbers, and underscore characters are permitted.

File extensions can be up to three characters and can include letters and numbers. The extensions listed in Table 21-2 are created by the Control Center design tools.

Table 21-2 Reserved file extensions

Extension	File type
.app	Application object
.bar	Bar menu
.pop	Pop-up menu
.str	Structure pick list
.fil	Files pick list
.val	Values pick list
.bch	Batch pick list
.scr	Screen format
.lbl	Label format
.frm	Report format
.npi	Screen, label, and report formats translated for Dgen.exe

NOTE
See Chapter 24, "Creating Templates," for more information on .npi files.

Compiler Commands

The commands listed in Table 21-3 activate certain options during compilation. To use these commands, you must embed them inside the template program code file. They are intended to turn on and turn off selectively the assembler and compiler listings to the output file. This is useful when you want a selective listing of only portions of the code, or during debugging to list problem areas of the source file.

Table 21-3 Compiler commands

Command	Description
INCLUDE <filename>	Includes information from named file.
#CODON	Include assembler statements (pseudo code) in compiler listing.
#CODOFF	Omit assembler statements from compiler listing. This is the default.
#LSTON	Turns list output on if –l was used in the command line when invoking the compiler. This is the default.
#LSTOFF	Turns off list output.

NOTE
See Chapter 24, "Creating Templates," for more information on compiler commands.

Definition Commands

The commands described below define variables and user-defined functions.

DEFINE

DEFINE declares a user-defined function for use within the template language.

Command Syntax

DEFINE <funcname> ([<varname>] [,<varname>...])[RETURN] ENDDEF

Rules

User-defined functions must begin with DEFINE and end with ENDDEF.

DEFINE statements may not be nested within other DEFINE statements.

Anything declared within a DEFINE statement is local to that statement.

The pending value when RETURN or ENDDEF is reached is the return value of the function.

Example

```
{DEFINE endofpage();
 VAR length;
 length = CURLINE();
 IF length > 60 THEN
    pagecnt+=1;
    PAGEBREAK();
    PRINT(CRLF+" Page "+STR(pagecnt));
 ENDIF
 RETURN;
 ENDDEF;}
```

ENUM

ENUM substitutes a number or string constant value for a symbolic name at compile time. This helps make a program more readable by assigning values to all the symbol names listed. If integers are specified, the default values are the integer values in order following the first or subsequent assignments.

Command Syntax

ENUM <symbol name>=<exp> [,<symbol name> [= <exp>]...]

Rules

After the first numeric assignment to a symbolic name, the subsequent symbols are assigned sequential values. A new assignment restarts the sequence.

Spaces may be used instead of commas.

Sequential values are 0-n, A-Z.

The entire ASCII standard set is available. Values higher than 255 will not be output.

NOTE
ENUM creates a list of constants assigned by the compiler. You cannot change the value assigned to the constant when running the compiled template.

Example

```
{ENUM menu = 1,
    dialog,
    keyword,
    new_start = 6,
    more;}
```

This assigns 1 to menu, 2 to dialog, 3 to keyword, 6 to new_start, and 7 to more.

SELECTORS

SELECTORS defines selectors and assigns numeric selector IDs.

Command Syntax

SELECTORS <selector declaration list>

The specified identifiers are defined as selectors and the specified integers become the ID numbers. Selector declarations are global in scope even if defined within a function. Identifier integers cannot be used anywhere in the template except as selectors.

NOTE
Use selectors with caution, if at all.

VAR

VAR declares variable names for later use within a template.

Command Syntax

VAR <varname> [,<varname>...];

Rules

<varname> must begin with a letter or underscore. Successive characters may be letters, underscores, or digits.

A variable name may be up to 255 characters long.

Variables declared within a DEFINE statement are local to that function. Variables declared outside a DEFINE statement are global to the template. Local variables take precedence over global ones.

Example

```
{VAR x,y,menu_one,temporary_count;}
```

Program Flow Commands

CASE...ENDCASE

CASE...ENDCASE selects one course of action from a set of alternatives.

Command Syntax

CASE <exp> OF
    CASE <n>:<commands>
    CASE <n>:<commands>
    .
    .
    .
    [OTHERWISE:<commands>]
ENDCASE

Rules

CASE selectors (n:) must be numeric.

Example

```
{CASE MENUACT OF
   1: GOTO nogen
   2:}Today is {DATE()}
  {3: INCLUDE "Test.doc"
 ENDCASE}
```

GOTO

GOTO directs the program to jump to the specified label.

Command Syntax

GOTO <label>

Rules

The specified label name follows the same rules as a VAR name; that is, it can be up to 255 characters long and is not case sensitive.

The label name should appear on a command line by itself and must have a colon (:) as its last character.

Example

```
{scrntop:}
 {IF count > 20 THEN GOTO scrnbot ENDIF
    line = SCREEN(count);
    IF count <10 THEN
       linecnt = "0"+STR(count);
    ELSE
       linecnt = STR(count);
    ENDIF
    PRINT(linecnt+":"+line+CRLF);
    count +1 =
    GOTO scrntop;}
{scrnbot:}
{RETURN;}
```

IF...THEN...ENDIF

IF...THEN...ENDIF tests for a true or false condition and branches to separate commands depending on the result.

Command Syntax

IF <condition> [THEN] <commands> [ELSE <commands>] ENDIF

Rules

Nesting rules are the same as for the dBASE IF...ENDIF command.

In cases where the meaning is unambiguous, the IF structure may be abbreviated, as in the second part of the example.

Example

```
{IF a < b THEN x = 25;}
    text for a = {a} and x = {x}
 {ELSE x = 15;}
    text for b = {b} and x = {x}
 {ENDIF}

 {IF "pigs" != "aviators" GOTO obviously}
 Learn something new every day.
 {obviously:}Pigs can't fly
```

RETURN

Inside user-defined functions, RETURN exits to the caller and provides the pending value. Outside a function, it terminates a template run.

Command Syntax

RETURN [<exp>]

Rules

This command is optional inside functions. If it is not provided, the RETURN value will be the pending value, if any, provided by the function.

When used outside functions, RETURN terminates the template run. If the run terminates with a RETURN value other than 0, it indicates an error condition. This means you must specify RETURN 0 at the end of your template.

NOTE
Non-zero RETURN error messages will be displayed if you are generating files from within dBASE IV. If you are using the stand-alone interpreter, run DGEN within a batch file and check the DOS ERROR LEVEL number.

Example

```
{DEFINE sqr(x)
 RETURN x*x;
 ENDDEF}
```

Loop Commands

FOREACH...NEXT

FOREACH...NEXT moves sequentially through repeated occurrences of DOS files, design object files, elements, and attributes.

Command Syntax

FOREACH <loop selector> [<cursor var>] [IN <cursor var>] <commands>
[<cursor var>] NEXT [<cursor var>]

Rules

Each construct declares a cursor variable that moves through some repeated data until it reaches the end of the data.

The FOREACH loop syntax always ends with the NEXT [<cursor var>] command.

The loop selectors can be found in the following .def files:

- **Applctn.def**
  - Loop selectors for all object files

    ELEMENT
    BOX_ELEMENT
    FLD_ELEMENT
    TEXT_ELEMENT
    TREE

  - Specific loop selectors for .app files

    Frame Selector:

    TEXT_ITEM

  - Specific loop selectors for .pop, .fil, .str, .val, .bar, .bch files

    Frame Selectors:

    MENU_HELP
    MENU_BEFORE
    MENU_AFTER

    Field Selectors:

    INLINE_DO
    ITEM_HELP
    ITEM_BEFORE
    ITEM_AFTER

- **Form.def**
  - Loop selectors

    ELEMENT
    BOX_ELEMENT
    FLD_ELEMENT
    TEXT_ELEMENT

- **Label.def**
  - Loop selectors

    ELEMENT
    FLD_ELEMENT
    TEXT_ELEMENT

- **Report.def**
  - Loop selectors

    ELEMENT
    BAND_ELEMENT
    BOX_ELEMENT
    FLD_ELEMENT
    TEXT_ELEMENT
    RULER_ELEMENT
    PARA_ELEMENT
    PAGE_ELEMENT

Example

```
{
 DEFINE menu_hlp(mhead,mline);
   //
   // THIS UDF IS USED FOR PROCESSING MENU LEVEL HELP TEXT
   //
   Endofpage();
   Print_Head=1;
   FOREACH MENU_HELP
     IF MENU_HELP THEN
       IF Print_Head THEN
         PRINT(crlf+mhead+crlf+mline+crlf);
         LMARG(2);
         Print_Head=0;
       ENDIF
       PRINT(RTRIM(MENU_HELP)+crlf);
       endofpage();
     ENDIF
   NEXT;
   endofpage();
   LMARG(0);
  RETURN;
 ENDDEF; }
```

In this example the loop selector MENU_HELP is used. The details of this selector are in Applctn.def. It is a frame-level selector that can occur in all design object files except an application object file. You must therefore check for it at the frame level of the correct design object file. The Menu_hlp( ) UDF is called from within a template which is already referencing a specific design object file.

It is not necessary to explicitly declare a cursor variable if it is not going to be referenced by another expression or loop.

DO...WHILE/UNTIL...ENDDO

This construction is similar to the dBASE command DO WHILE...ENDDO. It repeats the enclosed commands while the specifed condition is true.

Command Syntax

DO [<commands>] WHILE/UNTIL [<condition>] [<commands>] ENDDO

Rules

Commands which appear between the DO and the WHILE will be executed at least once.

Commands between the WHILE and the ENDDO are executed after the WHILE test. If the test returns false, the statements will not be executed.

If UNTIL is used in place of WHILE, the test is reversed. Statements between the UNTIL and the ENDDO will be executed before the test. Even if the test returns false, the statements will have been executed once.

Both WHILE and UNTIL may be used to exit conditionally from the FOR and FOREACH loops.

FOR...NEXT

This construction is often used for incremental looping.

Command Syntax

FOR <varname> = <expN1> TO <expN2> [STEP <constant>] <commands> NEXT [<varname>]

Rules

The counter <varname> is declared by the FOR statement. It is initialized to <expN1>.

The test FOR <varname> <= <expN2> is performed at the start of the loop. If <varname> is less than or equal to <expN2>, the loop is performed. NEXT increments <varname> by <constant>. The loop may be executed from none to many times depending on the results of the test.

If STEP is not specified, NEXT increments by one.

You cannot use a variable name for the STEP constant, but you may use the ENUM command to assign a readable symbol to <constant>.

EXIT

EXIT leaves the current loop, and control is passed to the first statement following the loop construct.

Command Syntax

Beginning of loop [EXIT] End of loop

Rules

This command may be used in any of the loop constructs: DO WHILE, DO UNTIL, FOREACH, and FOR.

Control is passed to the command following the NEXT or ENDDO statement.

LOOP

LOOP jumps back to the beginning of the current loop.

Command Syntax

Beginning of loop [LOOP] End of loop

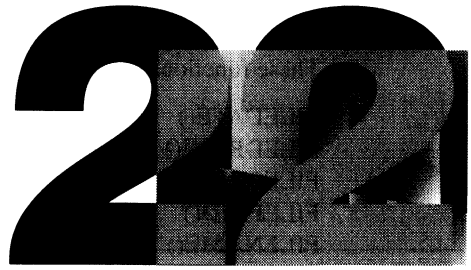
Rules

Inside a FOREACH or FOR loop, control advances to the next WHILE or UNTIL test.

If there is no WHILE or UNTIL test, the current cursor or counter is incremented and control is passed to the starting FOREACH or FOR.

Inside a DO WHILE, control advances to the next WHILE or UNTIL test. If there is no test, control passes to the top of the DO loop.

Library Functions



This chapter describes the function selectors provided in the Builtin.def library of functions. Each function is also known as a selector and has a specific ID number associated with it in this library. You may not change any of these ID numbers in the Builtin.def file. To do so will cause compilation errors. If you need a synonym for a function already assigned a selector ID number in Builtin.def, then add the new synonym and assign the same selector ID number to the synonym.

Functional Groupings

The function selectors provided by the Builtin.def library can be organized into the following application groups:

Input Text File Functions

These functions access DOS text files as input:

COPY()
TEXTCLOSE()
TEXTGETC()
TEXTGETL()
TEXTGPOS()
TEXTOPEN()
TEXTSPOS()

DOS Filename Parsing Functions

These functions support the use of DOS filenames:

FILEDATE()
FILEDRIVE()
FILEEXIST()
FILEFIND()
FILENAME()
FILEOK()
FILEPATH()
FILEROOT()
FILETYPE()
FILESIZE()
PATHEXIST()

Cursor Primitives

These functions create and manipulate cursor variables which access object layout structures:

ATOMC()
COUNTC()
EOC()
IIDC()
MAKEC()
NEXTC()
SETC()
TYPEC()
VALC()

String Manipulation Functions

These functions provide string manipulations similar to their counterparts in dBASE IV:

ALLTRIM()
AT()
ATALPHA()
BACKSLASH()
LEN()
LOWER()
LTRIM()
REPLICATE()
RTRIM()
SPACE()
SUBSTR()
TOKEN()
UPPER()

Number Handling Functions

These functions provide basic numeric comparisons:

MAX()
MIN()

Data Conversion Functions

These functions convert data type between string and numeric values:

ASC()
CHR()
COL1()
COL2()
ROW1()
ROW2()
SCREEN()
STR()
VAL()

Output File Formatting Functions

These functions direct the output stream of generated text to files and provide basic page formatting:

APPEND()
CREATE()
CURLINE()
PAGEJECT()
PAGEL()
POKE()
PRINT()
TABTO()

System Functions

These functions provide access to the built-in trace debug facilities of Dgen.exe, dBASE SET values, and DOS environment variables:

ARGUMENT()
BREAKPOINT()
DATE()
DEBUG()
EXEC()
GETENV()
IMPORT()
NAMESET()
NAMETOKEN()
NEWFRAME()
NUMSET()
STRSET()
VERSION()

User Interaction Functions

These functions allow a template program to interact with the user to get input during program generation, or control the screen display:

ASKUSER()
CGET()
CLS()
CPUT()
CURSOR_POS()
NMSG()
PAUSE()
PMSG()

ALLTRIM(<expC>)

ALLTRIM() trims all leading and trailing spaces from a string.

Usage

This function allows you to handle a character string without its leading and trailing blanks.

Example

```
{VAR a, b;
 a="    Your Company "
 b=ALLTRIM(a)
 PRINT("First example is "+a+".")
 PRINT ("Second example is "+b+".")}
```

This will print the following:

```
First example is    Your Company  .
 Second example is Your Company.
```

See Also

LEN(), LTRIM(), RTRIM()

APPEND(<filename>)

APPEND() concatenates text to the end of an existing text file. If the named file does not exist, it will be created.

Usage

This function allows you to add text to a file.

Example

```
{IF FILEEXIST("Some.app") THEN
   APPEND("Mainmenu.prg")}
   DO MAIN
 {ELSE
   APPEND("Mainmenu.prg")}
   DO ERRORS
 {ENDIF}
```

This example tests for the existence of the Some.app file. If the file is in the default directory, the text *DO MAIN* is placed in the Mainmenu.prg file. Otherwise, the text *DO ERRORS* is added to the Mainmenu.prg file.

NOTE
Each main template should have either a CREATE() or APPEND() function near the start, otherwise output will go to a file called Text.out.

See Also

CREATE()

ARGUMENT()

ARGUMENT() returns either the literal text string which is passed as the second argument to the dBASE DGEN() function, or the literal text string specified with the -p argument to stand-alone Dgen.exe.

Usage

This function returns the second portion of the argument that was passed to DGEN(<expC1> [,<expC2>]).

expC1 is the name of the gen file.

expC2 is the parameters to pass to the gen file.

The template programmer is responsible for parsing the parameter string with the template function TOKEN() and converting the case of the string.

If no parameters or string functions were passed to DGEN(), ARGUMENT() returns a null.

Examples

To generate a program from Myform.scr in a dBASE program:

```
DEXPORT SCREEN Myform   && Creates Myform.snl
ln_result= DGEN("FORM.GEN","MYFORM.SNL")
```

To generate a program from Myform.scr in a Template Language program:

```
{
lc_form = ARGUMENT()
NEWFRAME( lc_form )
}
```

See Also

DGEN() and DEXPORT in *Language Reference*, TOKEN() in this chapter.

ASC(<expC>)

ASC() returns the ASCII value of the first character of a string.

Usage

This function enables you to determine the ASCII value of the character at the far left of the string. You may use the result as a flag or as an indication of uppercase or lowercase.

Example

```
{PRINT(ASC("alphabet"))}
```

This will print the ASCII value of *a,* which is 97.

See Also

CHR(), STR(), VAL()

ASKUSER(<expC1>,<expC2>,<expN>)

ASKUSER() opens a box in which it shows the prompt string (<expC1>), displays a default answer string (<expC2>), and assigns a maximum length for input (<expN>). 78 is the maximum length that can be specified for <expC1>. The response is captured to a previously named variable.

Usage

This function allows you to prompt the user during the template generation process for specific information, such as a filename.

Example

```
{VAR Filename;
 IF NOT MENU_VIEW THEN
     Filename=ASKUSER("Please enter a database name ",
   "Client.dbf",12)
 ENDIF}
```

The selector MENU_VIEW is checked. If it is null, the template asks the user to enter a filename. The value entered is assigned to the Filename variable. The default filename is Client.dbf.

See Also

CGET(), NMSG(), PAUSE(), PMSG()

AT(<expC1>, <expC2>)

AT() performs a substring search. It returns the location of <expC1> in <expC2>. Zero (0) is returned if <expC1> is not found in <expC2>.

Usage

This function allows you to determine the existence of embedded information, or to locate the starting position of information you wish to use.

Example

```
{VAR Location;
 Location=AT("Prog","Advanced Programmer's Guide");
 PRINT(location)}
```

The Location variable will hold the value 10.

ATALPHA(<expC>)

ATALPHA() finds the first non-blank character in an expression.

Usage

This function skips over blanks to the beginning of essential text.

Example

```
{VAR Location;
 Location=ATALPHA(" Edit Client Info");
 PRINT(Location)}
```

The Location variable will hold the value 2, since the first non-blank character is in position 2 of the string.

See Also

AT()

ATOMC(<cursor>, <expN>)

ATOMC() returns the attribute value of the selector referenced by the cursor variable and the IID given by <expN>.

Usage

This function allows you to reference selectors by number rather than name.

Example

```
{FOREACH FLD_ELEMENT k }
 Field {k}:the value of selector 123 is {ATOMC(k,123)}
 {NEXT k;}
```

NOTE
The ATOMC() function is part of the implementation of the FOREACH loop, but it may be called directly.

See Also

COUNTC(), MAKEC(), NEXTC()

BACKSLASH()

BACKSLASH() inserts a backslash (\) into the output text file. A backslash within command braces is the instruction to ignore the special meaning of the next character. It is equal to the quoted string "\\". Outside of curly braces, it works as a continuation character and discards the carriage return linefeed character at the end of a line.

Usage

Use BACKSLASH() as an *escape* character to suppress the special meaning of characters in template language. For example, if you need to use curly braces as text characters in a special template, you would use BACKSLASH(){ and BACKSLASH()}.

Example

The following three lines all do the same thing:

```
d:{BACKSLASH()}Test.prg
 {PRINT("d:\\test.prg")}
 d:\test.prg
```

BREAKPOINT(<expC>)

BREAKPOINT() displays the value of the specified expression and requests the level of debugging action to implement. The breakpoint expression will be written to the list file specified by using the -l option of the DGEN stand-alone interpreter.

Usage

This function displays the message specified with the <expC> argument, on the prompt line, and also displays a navigation message that explains how to change the debug trace level. If you press a key from 0 to 6, this number is taken as the new debug trace level. If you press any other key, processing continues with the statement after BREAKPOINT(). See DEBUG() for an explanation of debug levels.

Variation in dBASE IV

From the Control Center, DEBUG() trace function is limited to level 1, the talk level. Therefore, BREAKPOINT() is limited to trace levels 0 and 1 in dBASE IV.

See Also

DEBUG()

Example

```
{VAR temp; temp = 1;
 FOR i = 1 TO 10
 BREAKPOINT("The value of temp is " + STR(temp));
 gettemp(); // user-defined function
 NEXT
 }
```

CGET()

CGET() accepts any keystroke and returns the ASCII decimal value.

Usage

This function waits for keyboard input from the user. When a key is pressed, it returns the ASCII value for that key and program execution continues with the next statement after CGET().

Example

```
{VAR key; key=""; PMSG("Continue Y/N?");
 DO WHILE NOT AT(UPPER(CHR(key)),"YN")
    key=CGET();
 ENDDO}
```

CHR(<expN>)

CHR() returns the ASCII character for the number specified in <expN>. The range for <expN> is 0–255.

Usage

This function allows you to display characters for which there may be no keyboard equivalent. By concatenating them together, you can create your own boxes or other graphic shapes. If <expN> is invalid, it returns a null string.

Example

```
{PRINT(CHR(24))}
```

This example will output an up arrow.

See Also

ASC()

CLS()

CLS() clears the screen and positions the cursor to the home location 0,0.

Usage

This enables you to clear the entire screen in readiness to write more information to it. There are no arguments allowed for this function.

Example

```
{CLS()}
```

See Also

ASKUSER(), CGET()

COL1()

COL1() returns the number of the first column of an object frame. If there is no frame, it returns a 0.

Usage

This function determines the column number of the left edge of an object.

Example

```
@ {ROW1()},{COL1()} TO {ROW2()},{COL2()}
```

This example writes dBASE code for a box.

COL2()

COL2() returns the number of the last column of an object frame. If there is no frame, it returns 79.

Usage

This function allows you to determine the location of an object.

Example

```
@ {ROW1()},{COL1()} TO {ROW2()},{COL2()}
```

This example writes dBASE code for a box.

COPY(<filename>)

COPY() copies the contents of the named text file into the current output text file.

Usage

This function allows you to insert information from auxiliary files into the output file.

Example

The following example copies Proc_lib.prg into the newly created file Myproc.prg.

```
{CREATE("Myproc.prg");
 COPY("Proc_lib.prg")}
```

See Also

APPEND(), CREATE(), TEXTCLOSE(), TEXTOPEN()

COUNTC(<cursor>)

COUNTC() returns the current index of a cursor within a collection.

Usage

This function returns a number representing the current relative position of a cursor. Cursor is the pointer variable to examine. Zero indicates the cursor was invalid or uninitialized, or referenced an empty collection.

This function is equivalent to referencing the cursor in most cases.

Example

The following template results in the cursor position being printed.

```
{
include "builtin.def"
include "form.def"
foreach FLD_ELEMENT F}
Field {COUNTC(F)} has picture template \
{F.FLD_TEMPLATE}

{next F
return 0;
}
```

See Also

CURSOR _POS()

CPUT(<expC>)

CPUT() writes the specified string at the current screen cursor location.

Usage

This function allows you to place messages anywhere on the screen or to selectively clear parts of the screen. The cursor position is usually set by CURSOR_POS(). See the two user-defined functions SAY_CENTER() and SAY() in the Builtin.def file, located in the template language directory of your hard disk.

Example

```
{CURSOR_POS(23,0); CPUT(SPACE(80));}
```

See Also

CLS(), CURSOR_POS()

CREATE(<expC>)

CREATE() creates and opens a text file. If you do not CREATE() and name an output file, or APPEND() to an existing text file, text output from the template program goes to a default file named Text.out.

Usage

This function allows you to specify the text file for the output of the template. This function returns 1 for TRUE, when the file is successfully created and opened, or 0 for FALSE, when the file cannot be opened for any reason. <expC> is the name of the new file to be created and opened, and it can include full drive and path prefix. If the named file exists, it will be overwritten.

Example

```
{INCLUDE "Builtin.def"
ENUM OUTFILE = "Typing.tst"
CREATE(OUTFILE);
}
The quick brown fox jumps over the lazy dog.
```

This template creates the output file Typing.txt and writes the text following the closing brace to it. Note the use of an enumerated symbolic constant as the argument to CREATE(). Also note the use of the semicolon after the CREATE() function. This function returns a value which would have been printed before the text string. The semicolon discards the numeric value returned by the function.

See Also

APPEND()

CURLINE()

CURLINE() returns the number of line feeds output since the last form feed. It does not include line feeds output with the POKE() function.

Usage

This function allows you to do your own page formatting for the text being placed in the output file. Note that CURLINE() starts counting from zero; therefore, the first line count is zero.

Example

```
{Linecnt=CURLINE();
 IF Linecnt > 60 THEN
 PAGEJECT()
 ENDIF}
```

See Also

PAGEBREAK(), PAGEL()

CURSOR_POS(<expN, expN>)

CURSOR_POS() sets the current cursor position on the screen. Row and column are zero-based, with 0,0 at the top left corner.

Usage

This allows you to position the cursor anywhere on the screen before writing to it.

Example

```
{CURSOR_POS(23,0); CPUT(SPACE(80));}
```

DATE()

DATE() returns the system date and time in the format: 5-31-87 2:30p.

Usage

This function allows you to include the system date and time in files you generate.

Example

```
Application Date: {DATE()}
```

Assuming you start executing the template program on January 31, 1990 at 2:30 p.m., this program would print *Application Date: 1-31-90 2:30p.*

See Also

FILEDATE()

DEBUG(<expN>)

DEBUG() sets or resets the program debug trace level. The trace level is set by using a value from 0 to 6 for <expN>.

0 = Quiet	Odometer only
1 = Talk	Odometer + display generated text, interrupt off
2 = Examine	Odometer + display generated text + interrupt on
3 = Trace	Examine with display of source code
4 = Walk	Trace with keystroke wait on each source line
5 = Animate	Trace + Stack trace + Opcode trace
6 = Step	Animate with keystroke wait on each opcode

When DEBUG() is waiting for a keystroke, the input key causes execution to advance. If the key is a number from 0 to 6, the debug level is reset to the indicated level. Otherwise, the keystroke is discarded.

This function always returns a null string.

Usage

Levels 2 and 3 interrupts allow the input of new level numbers, 0 through 6.

Levels 5 and 6 code streams are sent to the list file specified in the – l option of the DGEN command line. Please see Chapter 6 for more information on DGEN.

NOTE
This function is primarily for use with the DGEN template interpreter. In dBASE IV, only levels 0 and 1 are available. The **Display While Generating** *option in the Apgen* **Generate** *menu has the same effect as issuing a DEBUG(1) from inside a template program.*

Example

```
{IF myvar == 0 THEN DEBUG(3); ENDIF}
```

See Also

BREAKPOINT()

EOC(<cursor>)

EOC() tests whether the cursor variable has reached the end of the set. If there are more elements of the set, 0 is returned. If the end has been reached, 1 is returned.

Usage

This function allows you to test if the entire set has been scanned.

Example

```
{FOREACH FLD_ELEMENT k ) field (k)
 {NEXTC(k);
 IF EOC(k) THEN GOTO done;}
 {NEXT k;
 done:}
```

This example loops through the fields referenced by the cursor variable *k*. The EOC() function tests to see if the cursor has advanced beyond the end of the set of fields.

NOTE
This function is part of the implementation of the FOREACH loop. However, it is available to be used directly.

EXEC(<filename>)

EXEC() calls DOS and executes a DOS command when a template is operating under the DGEN stand-alone interpreter.

Usage

This function executes the DOS command line specified by <filename>. Template processing continues upon completion of the DOS action.

Example

```
{EXEC("dir  *.prg");}
```

NOTE
Do not use this function in dBASE IV. This is for the stand-alone interpreter Dgen.exe only.

FILEDATE(<filename>)

FILEDATE() returns the date and time the specified file was created or last saved. The date is returned in the form YYYY-MM-DD. The time is returned as HH:MM:SS (24-hour clock). For example: 1957-11-14 14:00:00.

Usage

This is useful for keeping track of template versions during development.

Example

```
{VAR fdate; fdate = FILEDATE ("Menu.gen")}
```

See Also

DATE()

FILEDRIVE(<filename>)

FILEDRIVE() returns the drive letter from a DOS filename.

Usage

This function determines the drive letter, and returns the letter of the drive in the same case as you entered it in the filename argument.

Example

```
{VAR drive;
 drive=FILEDRIVE("D:\test\test.dbf");
 PRINT(drive)}
```

This will print the drive letter *D*.

FILEERASE(<filename>)

FILEERASE() erases the specified filename. The current directory is assumed unless a drive or path is specified. If a drive or path is given, the filename is treated as unambiguous.

Usage

This function returns the value 1 (true) if the file was deleted. If the file could not be deleted, 0 (false) is returned.

Example

```
{VAR Myfile; Myfile = "Mytest.doc";
 IF FILEEXIST(Myfile) THEN FILEERASE(Myfile) ENDIF}
```

FILEEXIST(<filename>)

FILEEXIST() tests for the existence of a file on a disk.

Usage

This function allows you to determine the existence of a named file on disk. If it does not exist, for example, you may have to prompt the user for a correct entry.

Example

```
{VAR filename;
 filename="Custmast.dbf";
 IF NOT FILEEXIST(filename) THEN
    PAUSE("File "+filename+" does not exist")
 ENDIF}
```

See Also

ASKUSER()

FILEFIND(<filename>,<search flag list>)

FILEFIND() returns the file specifications requested by the search flags you set after the filename.

Usage

List the multiple search flags in the order in which you want the specifications returned. Use no spaces or delimiters between search flags. DOS wildcard characters are not supported.

Search Flag	Returns
A	unambiguous SYSTEM file name
1	unambiguous NORMAL file name
2	current DOS drive\directory
3	DOS path
4	dBASE SET PATH
5	dBASE home drive\directory
6	default drive
7	SQL Home

Parameters A, 1, 2, 3 are used with stand-alone Dgen.exe.

Parameters 4, 5, 6, 7 are used only with the DGEN() function from within dBASE IV.

Examples

If you installed dBASE IV in C:\DBASE:

```
{
FILEFIND("DBASE.EXE","5")
}
```

returns:

C:\DBASE;\DBASE.EXE

If you want to look for the location of CATALOG.CAT the same way dBASE IV looks for it:

```
{
FILEFIND("CATALOG.CAT","245")
}
```

would return the full path for the first occurrence of CATALOG.CAT by searching the current directory, next the dBASE SET PATH, and finally the dBASE home directory.

FILENAME(<filename>)

FILENAME() extracts the filename and extension from a fully specified path.

Usage

This function returns just the filename and dot extension, if any, as a file may be kept in different subdirectories on different systems.

Example

```
{ x = FILENAME("c:\\db4\\test\\apps\myprog.app") }

The filename is {x}
```

This would print *The filename is myprog.app.*

FILEOK(<filename>)

FILEOK() checks for a valid DOS filename.

Usage

This function is useful in trapping user data entry errors at the time of template generation.

Example

```
{VAR filename;
 filename="test\\doc.prg";
 IF FILEOK(filename) THEN
    CREATE(filename)
 ELSE
    filename=ASKUSER("Please enter a valid filename","",12)
 ENDIF}
```

This checks whether the Filename variable holds a valid DOS filename. If so, the file is created. If not, the program asks for a valid filename.

See Also

ASKUSER(), NMSG(), PAUSE()

FILEPATH(<filename>)

FILEPATH() returns the path portion of a complete filename.

Usage

This function isolates the path from the full description of the named file. It is useful when creating additional files.

Example

```
{VAR path, filename;
 filename="c:\\apgen\\test.prg";
 path=FILEPATH(filename)}
```

This results in the variable path holding *\apgen*.

FILEROOT(<filename>)

FILEROOT() returns the root of a filename (up to eight characters).

Usage

This function isolates the filename, up to the first eight characters, from the full description. This allows you to name subsidiary files with the same root but different extensions.

Example

```
{VAR filename, root;
 filename="c:\\apgen\myap\\test.prg";
 root=FILEROOT(filename)}
```

This results in the variable root holding the root of the Test filename, that is, Test.

FILESIZE(<filename>)

FILESIZE() returns the size of the specified file.

Usage

This function is useful in file comparison.

Example

```
{VAR fsize;
 fsize = FILESIZE("menu.gen")}
```

FILETYPE(<filename>)

FILETYPE() returns the three-character extension of a filename.

Usage

This allows you to check for the existence of a particular kind of file and act upon the resulting type.

Example

```
{VAR filename, ext;
 filename="c:\\test\\test.prg";
 ext=FILETYPE(filename)}
```

This results in the variable *ext* holding the file extension *prg*.

GETENV(<expC>)

GETENV() returns the value of a DOS environment variable. The result is a string or null if not found.

Usage

This allows you to check specific DOS environmental variables and act accordingly. This is available only with the DGEN stand-alone interpreter.

Example

```
Path returning template.
{ include "Builtin.def" PRINT(GETENV("PATH"))}
{return 0;}
```

This example returns the current DOS PATH command setting. Other DOS settings such as COMSPEC and PROMPT can also be checked with this function.

IIDC(<cursor>)

IIDIC() returns the integer selector value of the attribute pointed to by the cursor. The result is a number.

Usage

This allows you to determine the existence of a specific attribute by its internal identification number. The names of selectors in an INCLUDE file may be changed, but the numbers must remain the same.

Example

```
{FOREACH ATTRIBUTE k }
 attribute {k} is iid {IIDC(k)} with a value of {VALC(k)}
 of type {TYPEC(k)}
 {NEXT k;}
```

IMPORT(<expC1>,<expC2>,<expN>)

IMPORT() changes a dBASE schema object file into a template language layout object.

Usage

This function converts four types of dBASE data objects into template layout objects. <expC1> is the name of the dBASE input file you want to import. <expC2> is the name of the template output file. <expN> is a number representing the type of dBASE file you are importing. The supported file types are:

<expN>	Object file type
1	.qbe file, a dBASE read-only query object
2	.upd file, a dBASE update query object
3	.dbf file, a dBASE database file
4	.cat file, a dBASE catalog file

Example

This example uses the IMPORT() function to print out information about a specific database file in the current directory. You can use this program on other files by changing the filenames of the input and output files.

```
// IMPORT() Example: Print the header of a DBF file
{
     include "builtin.def"
     include "schema.def"

     //- Import the DBF file to make the DTO BNL object
     import( "GOODS.DBF", "GOODS.DTO", 3 )

     //- Load the DTO file into memory for processing
     newframe( "GOODS.DTO" );

     //- Create the file GOODS.PRT with header information
     create( NAME + ".PRT" );
}
Structure for database : {FRAME_PATH + NAME}.DBF
Number of records      : {TABLE_RECS}
Date of last update    : {TABLE_MOD_DATE}
{

return 0;
}
// END: IMPORT.COD
```

The output file Goods.prt will contain information similar to the following:

```
Structure for database : GOODS.DBF
Number of records      : 33
Date of last update    : 3-13-91
```

See Also

Schema.def library file

LEN(<expC>)

LEN() returns the length of the specified string.

Usage

You will often want to know the actual length of a string that may have been trimmed or extracted. You may, for example, have only a certain number of positions in which to print some information.

Example

```
{VAR menuname;
 menuname="Budgets";
 PRINT(LEN(menuname))}
```

This prints a 7, the length of the menu name.

LMARG(<expN>)

LMARG() sets the left margin for subsequent text output. A negative argument, 0, or 1 will reset the margin to 1.

Usage

This function helps format your text output. You could use it to provide more readable code for documentation purposes.

Example

```
action=0
 escape=27
 ret2caller=23
 DO WHILE .NOT. (action = ret2caller .OR. LASTKEY() = escape)
  {LMARG(4)}
 ACTIVATE POPUP {MENU_NAME}
 DO CASE
 CASE BAR() = {ROW_POSITN}
 {LMARG(7)}
 {//
 // command related to this action
 //}
 action={MENU_ACT}
 {LMARG(4)}
 ENDCASE
 {LMARG(1)}
 ENDDO
```

In this example the generated dBASE code is for a pop-up menu to be reactivated unless the action is *return to caller* or the last key pressed was **Esc**. Although the code in the example is flush to the left, LMARG() will indent the code first by three spaces and then by six spaces. LMARG(1) resets the output text stream to the beginning of the line.

LOWER(<expC>)

LOWER() converts a string to lowercase.

Usage

This function helps style your output. You can convert an all uppercase response into an initially capitalized word.

Example

```
{VAR test,test2;
 test="APPLICATION";
 test2=SUBSTR(test,1,1)
 +LOWER(SUBSTR(test,2,LEN(test)-1))}
```

The Test2 variable will hold the word *Application.* Everything to the right of the first character has been converted to lowercase.

See Also

UPPER()

LTRIM(<expC>)

LTRIM() trims the leading blanks of a string.

Usage

This function enables you to format a line of text by eliminating the blanks in front of a word.

Example

```
{VAR mstring,mstring2;
 mstring=" Application";
 mstring2=LTRIM(mstring);
 PRINT(mstring2)}
```

This results in the printing of just the word *Application* without the blank in front of it.

See Also

ALLTRIM(), RTRIM()

MAKEC(<expN>[,<cursor>])

MAKEC() creates a cursor using the internal selector number. Use the construction @<selector> for <expN>. Use <cursor> if creating a DOS file cursor.

NOTE
*All DOS file specifications are valid for <cursor> (for example, *.pop).*

Usage

This function enables you to declare a specific cursor variable independently of the FOREACH construct.

Example

```
j = MAKEC(@FLD_ELEMENT)
```

MAX(<expN>,<expN>)

MAX() returns the maximum value of two numeric expressions.

Usage

By comparing assorted pairs of numbers with this function, you can isolate the largest number.

Example

```
{VAR result,num1,num2;
 num1=10;
 num2=20;
 result=MAX(num1,num2);
 PRINT(result)}
```

The number printed is 20, the larger of the two variables.

See Also

MIN()

MIN(<expN>,<expN>)

MIN() returns the minimum value of two numeric expressions.

Usage

By comparing assorted pairs of numbers with this function, you can isolate the smallest number.

Example

```
{VAR result,num1,num2;
 num1=10;
 num2=20;
 result=MIN(num1,num2);
 PRINT(result)}
```

The number printed is 10, the smaller of the two variables.

See Also

MAX()

NAMETOKEN(<expC>,<expN>)

NAMETOKEN() parses a string for valid dBASE variable names.

Usage

This function provides parsing support for strings that contain dBASE variables. <expN> allows you to pass one of three option codes to this function.

<expN>	Option
–1	Validate <expC> as a variable name
0	Count the name and non-name tokens in <expC>
1...n	Return the nth token in <expC>

Example

This example shows use of the NAMETOKEN() function in a template program, and the contents of the Nametok.out file.

```
{ include "builtin.def"  // Builtin functions
var  arg_list,
     ln_token_cnt,
     lc_current_token,
     ln_counter      ;

  arg_list = argument()
  if arg_list == "" then
    return 1;
  endif

  Create( "NameTok.out" );
  ln_token_cnt = NameToken( arg_list, 0 );
}
  Token string = [{ arg_list }]
  Number of tokens = { ln_token_cnt }
{
  FOR ln_counter = 1 TO ln_token_cnt
    lc_current_token = NameToken( arg_list, ln_counter );
}
      Token ({ ln_counter }) = [{ lc_current_token }]
{
  NEXT ln_counter;

  return 0;
}
```

The output file Nametok.out contains the following tokens.

```
Token string = [a=b+c+4]
Number of tokens = 6
 Token (1) = [a]
 Token (2) = [=]
 Token (3) = [b]
 Token (4) = [+]
 Token (5) = [c]
 Token (6) = [+4]
```

NEWFRAME(<expC>)

NEWFRAME() allows you to change the current object being worked on by the template language. A 0 is returned if the change is successful, a 1 if not.

Usage

This function is used during generation so that multiple related objects can be processed in one run. Its primary use in the Applications Generator is to load the main menu specified in the application object. The command FOREACH TREE...NEXT then automatically loads objects further down the tree.

Examples

```
NEWFRAME("myprog.pop")

NEWFRAME(MENU_MAIN)
```

In the second example, MENU_MAIN is a selector defined in Applctn.def.

NEXTC(<cursor>)

NEXTC() allows you to advance the specified cursor by one position. There is no return value.

Usage

This function directly adjusts the cursor position beyond the normal single step of the FOREACH...NEXT construct.

Example

```
{FOREACH FLD_ELEMENT k }
 field {k}
 { NEXTC(k);}
 {NEXT k;}
```

This example moves the cursor to every other field.

NMSG(<expC>)

NMSG() displays a message on the navigation line. This is line 23 in normal mode, and line 41 in 43-line EGA mode.

Usage

The message is centered on the navigation line, and characters beyond 80 are truncated.

Example

```
{NMSG("Any key to continue...");}
```

See Also

PMSG()

NUMSET(<expN>)

NUMSET() returns the internal numeric setting of dBASE indicators. They are defined and documented in Builtin.def. NUMSET() is available only in dBASE IV. If the template is run in DGEN, NUMSET() will return 0 for all dBASE settings.

Usage

NUMSET() returns the state of a dBASE number-valued SET command.

Example

```
{NUMSET(_FLGCLOCK)}
```

This example returns a 1 if the clock is set on, a 0 if the clock is set off.

NOTE
_FLGCLOCK is one of the setting arguments specified by the ENUM command in Builtin.def.

PAGEJECT()

PAGEJECT() outputs a CHR(12), the form feed character. It also resets the CURLINE() function line-count to 0.

Usage

You can format your output text into pages with this function. With additional coding, you can provide headings and footers for your documentation.

Example

```
{IF CURLINE() > 60 THEN
     page=+1;
     PAGEJECT()
 ELSE
     GOTO begin
 ENDIF}
```

PAGEL(<expN>)

PAGEL() sets a constant page length for the output. A negative argument is the same as PAGEL(0) and means page breaks.

Usage

If you do not need conditional page ejects, this is an easy way to format the text output for documentation templates.

Example

```
{VAR file, page, counter;
 PAGEL(62);}
```

This sets the default page length for the output to 62 lines per page. After every 62 lines, a form feed character is inserted and CURLINE() is reset to 0.

PATHEXIST(<expC>)

PATHEXIST() returns a 1 if the path exists, a 0 if it does not. However, if <expC> is a null string, this means the current directory; therefore, the null string always returns a 1. If an invalid directory name is specified, that name is treated as a null string, and it too returns a 1.

Usage

This is useful for validating the path.

Example

```
{PATHEXIST("c:\\DOS")}
```

PAUSE([<expC>])

PAUSE() halts template interpretation and displays a message to the user on line 24 or 42, depending on the display size. The system waits for any keystroke before continuing. If the optional expression is omitted, no message is displayed.

Usage

This function is helpful for debugging templates.

Example

```
{VAR filename;
 filename="test.prg";
 IF NOT FILEEXIST(filename) THEN
    PAUSE("File "+filename+" is not in current
    directory")
 ENDIF}
```

PMSG(<expC>)

PMSG() displays the specified message on the prompt line. This is line 24 in normal mode, and line 42 in the 43-line EGA mode.

Usage

The message is centered on the prompt line, and characters beyond 80 are truncated.

Example

```
{PMSG("Any key to continue...");}
```

POKE(<expN>|<expC>|<cursor>|[,<expN>|<expC>|<cursor>...])

POKE() inserts the specified character expression into the text without any formatting.

<expN> is a number argument which is converted to a string before being output. <expC> is a character string to be output without modification. <cursor> is a cursor argument which is converted to the cursor position before being output.

Usage

This function is useful for sending escape sequences to the printer, or for creating binary files. Any CHR(10) linefeeds sent with POKE are not counted by the CURLINE() function.

Example

```
{POKE("This will be output", SPACE(10), "so will this")}
```

PRINT(<expN>|<expC>|<cursor>|[,<expN>|<expC>|<cursor>...])

PRINT() outputs the character expression into the text file.

NOTE
A carriage return (CHR(13)) is automatically inserted after a linefeed character, CHR(10). CURLINE() and LMARG() settings are still maintained.

Usage

You may use this function to output text within the command structure.

<expN> is a number argument which is converted to a string before being output. <expC> is a character string to be output without modification. <cursor> is a cursor argument which is converted to the cursor position before being output.

Example

```
{VAR mstring, crlf;
 mstring = "Application ";
 crlf = CHR(10);
 PRINT(mstring+crlf)}
```

REPLICATE(<expC>,<expN>)

REPLICATE() repeats the specified character expression <expN> times.

Usage

This function is useful for creating lines and other pictorial elements.

Example

```
{VAR a,crlf;
 crlf=CHR(10);
 a="Application: "+name;
 PRINT (a+crlf);
 PRINT(REPLICATE("-",LEN(a)))}
```

The word *Application:* plus the entered name will be underlined.

ROW1()

ROW1() returns the number of the first row of an object frame. If there is no frame, 0 is returned.

Usage

This function allows you to determine the specific size of an object.

Example

```
@ {ROW1()},{COL1()} TO {ROW2()},{COL2()}
```

This example outputs dBASE code for a specific size box.

ROW2()

ROW2() returns the number of the last row of an object frame. If there is no frame, 24 is returned.

Usage

This function allows you to determine the specific size of an object.

Example

```
@ {ROW1()},{COL1()} TO {ROW2()},{COL2()}
```

This example outputs dBASE code for a specific size box.

RTRIM(<expC>)

RTRIM() removes the trailing blanks from a string.

Usage

This is useful for eliminating trailing spaces in evaluating a string.

Example

```
{VAR test;
 test="Application ";
 PRINT(LEN(RTRIM(test)))}
```

This results in 11, the length of the string without the trailing blanks.

SCREEN(<expN>)

SCREEN() extracts a line from a screen object created by the Applications Generator only.

Usage

SCREEN() is helpful when documenting menus.

Example

```
{VAR scrn_line;
 scrn_line=1;
 PRINT(SCREEN(scrn_line))}
```

This will print line 1 of the work surface to the text file.

SETC(<cursor>,<expN>)

SETC() moves the specified cursor to a new relative position.

Usage

Although it is part of the internal implementation of the FOREACH loop, this function may be used directly to adjust the cursor position.

Example

```
{FOREACH FLD_ELEMENT k} Field {k} { SETC(k,2);} {NEXT k;}
```

This example advances the cursor *k* by two positions. The loop has the effect of visiting every third field.

SPACE(<expN>)

SPACE() inserts the specified number of blanks into the text file.

Usage

This function is helpful for formatting documentation templates, or for padding character variables.

Example

```
{VAR page;
 page=1;
 PRINT("Page "+page+SPACE(55)+DATE())}
```

This example will print the page number and the date 55 spaces apart.

See Also

ALLTRIM(), LTRIM(), REPLICATE(), RTRIM()

STR(<expN>)

STR() converts a number to a string.

Usage

This function allows you to convert a numeric type to a character type in order to print the number as part of a character string.

Example

```
{VAR num1,num2;
 num1=34;
 num2=num1*12;
 PRINT("The number is "+ALLTRIM(STR(num2)))}
```

This will print *The number is 408.*

NOTE
If the value passed to STR() is already a character type, it will be returned unchanged.

See Also

VAL()

STRSET(<expN>)

STRSET() returns the internal character setting of dBASE indicators. These declarations are found under ENUM for STRSET() in the Builtin.def file. This function is for dBASE IV only; it returns a null string from the DGEN stand-alone interpreter.

Usage

STRSET() returns the state of a specific dBASE setting.

Example

```
{STRSET(_DEFDRIVE)}
```

This example would return a D if the dBASE default drive had been set to D.

NOTE
_DEFDRIVE is one of the setting arguments declared by the ENUM command in Builtin.def.

SUBSTR(<expC>,<expN1>[,<expN2>])

SUBSTR() extracts a substring from <expC>, starting at location <expN1> for <expN2> characters. If <expN2> is not specified, the substring to the end of <expC> is returned.

Usage

This function is useful for isolating a portion of a character string.

Example

```
{VAR test,test2;
 test = "Empire State building";
 test2 = SUBSTR(test,AT("S",test),5)}
```

This will pick up the word *State* from the string *Empire State building.*

TABTO(<expN>)

TABTO() moves over to the specified column <expN> for just the current line before continuing output. A negative argument sets TABTO to 0. As with LMARG(), 0 and 1 are equivalent.

NOTE
Space characters (ASCII 32) are inserted. The actual tab character is not used.

Usage

You can use this function for formatting the text of the output file. For example, you may want all code comments to begin in the same place.

Example

```
Load menu.bin{TABTO(40)} && A binary program for menu support.
```

This will print the comment at column 40.

TEXTCLOSE()

TEXTCLOSE() closes a file opened by TEXTOPEN(). Files are automatically closed by subsequent TEXTOPEN() requests or upon the end of a template.

Usage

You should try to close text files when not in use, to reduce some processing overhead.

Example

```
{INCLUDE "Builtin.def"; ENUM eof = -1; VAR open, temp, crlf, line; line = 0;
  crlf = CHR(13)+(CHR(10)); open = TEXTOPEN("myfile.doc");
 IF open THEN temp = "a";
    DO temp = TEXTGETL(); WHILE temp != eof
       CPUT(temp); CPUT(crlf); line = line+(LEN(temp)/79+1)
       IF line > 21 THEN PAUSE("More..."); line = 0; CLS(); ENDIF
    ENDDO
 ENDIF
 TEXTCLOSE(); PAUSE("Press a key...")
 RETURN 0;}
```

This program opens a file called Myfile.doc and displays the first 21 lines of text on the screen. Lines longer than 80 characters should be word wrapped by the monitor hardware. The division by 79 is to keep track of the number of lines printed on the monitor. Once 21 lines are written, the template pauses and displays **More**.... Any key pressed clears the screen and displays the next 21 lines of text. This is repeated until the EOF character is read.

TEXTGETC()

TEXTGETC() returns a character from the current position of the open text file.

Usage

The file position is advanced by one character. When the end-of-file is encountered, – 1 is returned by the function. The end-of-file may be a **Ctrl-Z** (ASCII 26) or the physical end of the file.

Example

```
{ENUM eof = -1; VAR open, temp;
 open = TEXTOPEN("myfile.doc");
 IF open THEN temp = "a";
    DO temp = TEXTGETC(); WHILE temp != eof
       temp = TEXTGETC();
    ENDDO
 ENDIF
 TEXTCLOSE();}
```

This reads one character at a time out of the file Myfile.doc.

TEXTGETL()

TEXTGETL() returns a line from the current position of the open text file. The line may be up to 254 characters long. Line-end characters and CRLFs are removed. In a line longer than 254 characters, the first 254 are returned and the position is set to the first character of the remainder of the line.

Usage

The file position is advanced to the first character past the end of line. When the end-of-file is encountered, – 1 is returned by the function. The end-of-file may be a **Ctrl-Z** (ASCII 26) or the physical end of the file.

Example

```
{INCLUDE "Builtin.def"; ENUM eof = -1; VAR open, temp, crlf, line; line = 0;
 crlf = (CHR(10); open = TEXTOPEN("myfile.doc");
 IF open THEN temp = "a";
    DO temp = TEXTGETL(); WHILE temp != eof
       CPUT(temp); CPUT(crlf); line = line+(LEN(temp)/79+1)
       IF line > 21 THEN PAUSE("More..."); line = 0; CLS(); ENDIF
    ENDDO
 ENDIF
 TEXTCLOSE(); PAUSE("Press a key...")
 RETURN 0;}
```

This template gets one line at a time out of the Myfile.doc file. See TEXTCLOSE() for further discussion.

TEXTGPOS()

TEXTGPOS() returns the current file position of the open text file.

Usage

The file position is a number where 0 is the first character in the file. If no file is open, 0 is also returned.

Example

```
{ENUM eof = -1; VAR open, temp, linepos;
 open = TEXTOPEN("myfile.doc");
 IF  open THEN temp = "a";
    DO temp = TEXTGETL();WHILE temp != eof
      linepos = textgpos();
      CPUT(temp); CPUT("Position =" + STR(linepos);)
    ENDDO
 ENDIF
 TEXTCLOSE();}
```

The function TEXTGPOS() displays the current position in the Myfile.doc file.

TEXTOPEN()

TEXTOPEN() opens and initializes a text file for input. The file position is set to the start of the file. If the file is successfully opened, a 1 (true) is returned. If the file could not be opened, a 0 (false) is returned.

Usage

If a file is open and a subsequent TEXTOPEN() is issued, the previously opened file is closed.

Example

```
{ENUM eof = -1; VAR open, temp,;
 open = TEXTOPEN("myfile.doc");
 IF open THEN temp = "a";
    DO temp = TEXTGETL(); WHILE temp != eof
       CPUT(temp);
    ENDDO
 ENDIF
 TEXTCLOSE();}
```

Here TEXTOPEN() opens a file called Myfile.doc. If the file is successfully opened, the DO WHILE loop is executed.

See Also

APPEND(), CREATE(), TEXTCLOSE(), TEXTGETC(), TEXTGETL(), TEXTGPOS(), TEXTSPOS()

TEXTSPOS(<expN>)

TEXTSPOS() sets the file position in the text file to the number specified. The position specified should be a non-negative number.

Usage

Use 0 to reset the file position to the start of the file. A position past the end of the file sets the file position to the end of the file.

Example

```
{ENUM eof = -1; VAR open, temp, linepos;
 open = TEXTOPEN("myfile.doc"); TEXTSPOS(80*3);
 IF open THEN temp = "a";
    DO temp = TEXTGETL(); WHILE temp != eof
       CPUT(temp); CPUT("Position = "+ STR(linepos);)
    ENDDO
 ENDIF
 TEXTCLOSE();}
```

This example uses TEXTSPOS() to position on line 4 of a text file in order to skip a text file header and begin processing the relevant data.

TOKEN(<expC1>,<expC2>,<expN>)

TOKEN() parses a delimited string and returns the argument specified by the number in expN.

<expC1> is the delimiter character. If more than one character is specified, only the first one is used.

<expC2> is the delimited string containing one or more arguments. If this is a null string, TOKEN() returns the original string value and the number of arguments is 1.

<expN> is the number of the argument to return. If equal to zero, TOKEN() returns the total number of delimited strings as a quoted string.

Example

```
string = "A,B,CDE"
TOKEN( ",", string, 3 )
returns a "CDE" TOKEN( ",", string, 0 )
returns a "3" TOKEN( "B", string, 2 )
returns a ",CDE" TOKEN( ",", "A,B,,D", 3 ) returns a null
```

See Also

ARGUMENT() in this chapter, and DGEN() in *Language Reference*.

TYPEC(<cursor>)

TYPEC() returns a number indicating the type of the attribute currently pointed to by the cursor. The returned values are:

0 = integer
1 = string
2 = layout element
3 = control center object
4 = DOS filename
5 = not a valid cursor

Usage

This allows you to determine the type of a specific attribute.

Example

```
{FOREACH ATTRIBUTE k }
 attribute {k} is iid {IIDC(k)} with a value of {VALC(k)}
 of type {TYPEC(k)}
{NEXT k;}
```

The type number of the attribute value will be output.

UPPER(<expC>)

UPPER() converts a string to uppercase.

Usage

This function creates a copy of a string and converts the copy to uppercase. It does not change <expC>. It returns the converted copy of <expC>.

Example

```
{VAR app,name;
 name="menu.prg"
 app=UPPER(name)}
```

The App variable will hold MENU.PRG, even though the original entry was in lowercase.

VAL(<expC>)

VAL() converts a string to a number.

Usage

This is useful if you need to do arithmetic operations. You may, for example, want to keep an internal documentation number.

Example

```
{
   include "builtin.def"
}
String 1. {val("123")}
String 2. {val("")}
String 3. {val("2" + "2")}
String 4. {val("10 fish")}
String 5. {val(13)}
```

This template will return the following output file:

String 1. 123
String 2. 0
String 3. 22
String 4. 10
String 5. 0

NOTE
VAL() starts at the left end of the string and strips leading blanks until it encounters the first non-digit character. This is similar to the operation of the dBASE IV VAL() function.

VALC(<cursor>)

VALC() returns the actual value of the attribute pointed to by the cursor. The returned value may be a number or a string.

NOTE
The value must be printable to be accessible. It must be an attribute, not an element or object.

Usage

This allows you to determine the value of a specific attribute.

Example

```
{FOREACH ATTRIBUTE k }
 attribute {k} is iid {IIDC(k)} with a value of {VALC(k)}
 of type {TYPEC(k)}
{NEXT k;}
```

VERSION()

VERSION() returns the generator version number.

Usage

This is helpful in documenting the version of the Applications Generator used.

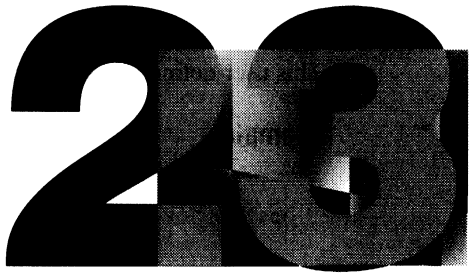
Example

```
Application Documentation Date: {DATE()}
Created using Generator Version: {VERSION()}
```

See Also

DATE(), FILEDATE()

User-Defined Functions of Builtin.def



The Builtin.def file is a function library for the template language. In addition to the built-in functions described in Chapter 22, it contains some useful user-defined functions (UDFs). These user-defined functions are described in this chapter.

The naming convention for template UDFs uses all lowercase characters to distinguish them from the library functions in Chapter 22.

You must compile the examples with Dtc.exe and interpret the .gen files with Dgen.exe to an output file, or use the DGEN() function to see the returned values.

say_center(mrow, mstring)

This user-defined function allows you to center justify and display a text string at a given row on the screen. It assumes a screen width of 80 for centering.

Example

```
{
mrow = 5
mstring = "Hi there"
say_center(mrow, mstring)
}
```

say(mrow, mcol, mstring)

This user-defined function allows you to display a text string at a specific row and column position on the screen.

Example

```
{
mrow = 5
mcol = 20
mstring = "At row 5 column 20"
say(mrow, mcol, mstring)
}
```

abs(value)

This user-defined function converts any negative number to its absolute value.

Example

```
{
abs(-1)
}
```

This will return a 1.

beep(value)

This user-defined function sounds the PC speaker for the specified number of times.

Example

```
{
beep(5)
}
```

This would sound the PC speaker five times.

cap_first(string)

This user-defined function returns the text string inside the parentheses with its first letter capitalized.

Example

```
{
cap_first(builtin.def)
}
```

This returns:

```
Builtin.def
```

nul2zero(numbr)

Template language returns a null string for screen attributes that evaluate to zero. This UDF was developed to convert null strings to zero where a zero value must be printed in a template generated program file.

Example

```
{
nul2zero(ROW_POSITN), nul2zero(COL_POSITN)
}
```

This program construct ensures that whenever a row or column position is at 0, the generated program contains zeros instead of null strings.

Creating Templates

The process of creating a working template in the template language involves the use of a text editor suitable for programming, the template language compiler (Dtc.exe), and the interpreter (Dgen.exe). This chapter provides the details of using the template language compiler and interpreter.

A standard development sequence would be to write the template, compile it using Dtc.exe, and then test it using the stand-alone interpreter, Dgen.exe. The interpreter provides a debugger that can be invoked by functions within your template or by command line switches.

Template Language Compiler

The compiler is a stand-alone program named Dtc.exe. Template source files have the .cod extension. These are the input files for the compiler. The compiled Dtc.exe output file has the .gen extension.

The Dtc.exe compiler processes the input source code file and any files included in it.

Command Syntax

The Dtc.exe compiler is called from the DOS prompt and has required and optional arguments. These arguments must begin with a hyphen and be in lowercase. The space following the argument letter is optional. Paths are supported.

The command line syntax for the compiler with arguments is as follows:

DTC -i <inputfile> [-o <outputfile>] [-l <listfile>] [-a] [-z]

The arguments are:

-i <inputfile> (required)	name of (.cod) template file to compile
-o <outputfile> (optional)	name for the compiled output file
-l <listfile> (optional)	file to receive the diagnostic listing
-a (optional)	echo template language assembly code instructions to the screen
-z (optional)	suppresses the source position numbers in the compiled output file.

If you do not specify an input file, a Help screen showing correct usage will appear.

Although many source files may be incorporated into a single compilation through the INCLUDE command within the template, only one output file is created.

If you do not specify an output filename, the compiler uses the input filename and adds the .gen extension to create the output file.

The -l option writes a list of the source lines with line numbers to the specified file. If you do not specify -l, the output goes to the screen.

The -a option includes assembly listings with pseudocode source lines in the listing output file. If you do not specify a listing file, the output of this argument goes to the screen.

The command line arguments may be omitted and the source code and assembly code listings can be turned on or off from within the template source code file. The compiler directives to control the output to listings are as follows:

- #LSTON and #LSTOFF turn on and off the source listing with line numbers to the output file.
- #CODON and #CODOFF turn on and off the assembly code listings with the source lines in the output list file.

These commands may be used to list an entire file, or just selected parts of a file to the source listing.

The -z option reduces the template .gen file by 30 percent and increases the execution speed of the template. If the -z option is used, you cannot see your template source code on the debugger, because the line number information used to find the code is not output.

Use the -z option after a template program is completely debugged and is correct.

Template Language Interpreter

The stand-alone interpreter, Dgen.exe, is similar in all respects to the version used by dBASE IV with the addition of some debugging features. Template interpreter input files are the .gen output of Dtc.exe. The debug features are invoked using functions within the template and by command line arguments.

Command Syntax

The Dgen.exe interpreter is called from the DOS prompt and has required and optional arguments. These arguments must begin with a hyphen and be in lowercase. The space following the argument letter is optional. Paths are supported.

The command line syntax for the interpreter with arguments is as follows:

DGEN -t <template.gen> [-i <objfile>] [-l <listfile>] [-n] [-p<parameters>]

The arguments are:

-t <template.gen> (required)	name of (.gen) compiled template file to process
-i <objfile> (optional)	filespec of object(s) to merge
-l <listfile> (optional)	file to receive the diagnostic listing
-n (optional)	echo command line arguments to the screen
-p <parameters>	passes a parameter string to the ARGUMENT( ) function

The design object files are optional. You may specify none, a single file, or a group of files. To specify a group of files, use the form *.app or use a comma-separated list, such as *.*app*,*.*pop*, or *.*bar*.

NOTE
Applications Generator design object files may be used directly. Forms, reports, and labels design object files that can be used by DGEN must be created using the DOS environment variable DTL_TRANSLATE, or by using the DEXPORT command. See Table 18-2 in Chapter 18 and the next section in this chapter for more information on DOS environment variable use. Use the translated file with the .npi extension for -i <objfile>.

The list file is an optional file to receive diagnostic listing information from the debugger.

The -n option echoes the command line arguments to the screen.

The -p option is a method of passing text string parameters to template programs. For example, you can pass a delimited string to DGEN as follows:

```
DGEN -t Form.gen -i Myform.npi -p parm1,parm2,parm3
```

Then use the ARGUMENT( ) function to get back the string "parm1,parm2,parm3".

The TOKEN( ) function can parse the returned text string and extract a specific parameter.

DOS Environment Variables

You can use the DOS environment command SET to tell DGEN which template (.gen) file to use. This is useful when you change default template files to customized files, or if you choose to keep your .cod files in a separate directory from the stand-alone template interpreter.

You may use many SET variables; the limit is determined by your DOS environment space. Include the SET variables that you use regularly in your Autoexec.bat file so that they are always initialized. If you get a DOS error message informing you that you are out of environment space, refer to the SHELL command in your DOS Reference manual.

Table 24-1 DOS environment variables

Variable	Description
SET DTL_TRANSLATE = on	Tells dBASE IV to create an additional source file with the extension .npi when reports, forms, or labels generators are used. The .npi files can then be used with the -i option of DGEN.
SET DTL_NOGEN = on	Suppresses dBASE code generation. This is mainly used with DTL_TRANSLATE in order to create new design object files without generating dBASE code.
SET DTL_TRACE = on	This will display all the elements and attributes of a design object except for frame level attributes. Look at a .def file under *Frame level selectors* for the list of frame level attributes.
SET DTL_CODFILE = <directory:\path\>	This tells Dgen.exe where to find the .cod files and lets DEBUG display the source during execution.

Debugger

The debugger is called from within a template through the functions DEBUG( ) or BREAKPOINT( ).

DEBUG(<expN>)

This function turns on the debugger within Dgen.exe at a specified level of diagnostics. The level is a number from 0 to 6. Levels above 1 are ignored by the dBASE IV internal code generator.

Debugging template programs occurs during several different stages in program development. First is during compilation. Dtc.exe will give error messages for most basic syntax errors.

Next, you add the debugging commands BREAKPOINT( ) and DEBUG( ) to the template source code file, and recompile the source code. Dgen.exe uses the debugging commands to display information on the DEBUG screen. If the template you are debugging is a text handling template and produces no dBASE code, then the debugging can occur entirely out of dBASE IV. If the template produces dBASE code, then compile this template-generated dBASE code in dBASE IV. The dBASE IV internal compiler will catch the syntax errors in the generated code.

The DEBUG( ) function must be present in any file before BREAKPOINT( ) can be activated. When DGEN encounters the DEBUG( ) function, depending on the level set with the numeric argument of DEBUG( ), it may pause and let you step through the code or animate. BREAKPOINT( ) displays the value of <expC> at the message line of the screen. Breakpoint messages are written to the log file if the -l option is used with DGEN.

Figure 24-1 shows the DEBUG screen with the simple template Textpul2.cod from Chapter 18. To get this screen to display, edit the Textpul2.cod file. Enter the DEBUG(4) function after the first left brace. Also, enter BREAKPOINT ("Processing Field Elements") before the template starts processing field elements.

Compile the changed Textpul2.cod. From the DOS prompt at the template language directory, type:

```
DTC -i Textpul2.cod
```

Use the .gen file with DGEN to start debugging:

```
DGEN -t Textpul2.gen -i Text.npi
```

The screen shown in Figure 24-1 appears. This is a split screen. The top half displays the source code line by line, as you press a key. The bottom half displays the code lines as they get generated.

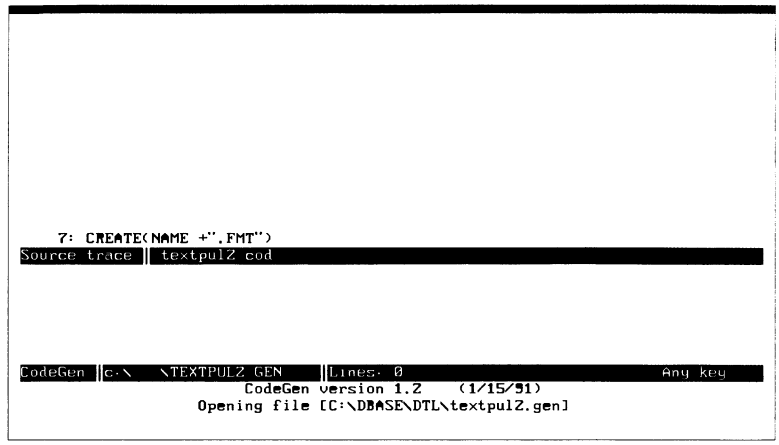


Figure 24-1 Debug screen

Because the DEBUG( ) talk level is set to 4, you can step through the program line by line. The BREAKPOINT( ) function on line 16 displays the message entered as its argument. It also reminds you that if you press any number between 0 and 6, you can change the debug level at this time. See Figure 24-2. If you change to 0 or 1, you can no longer get into DEBUG.

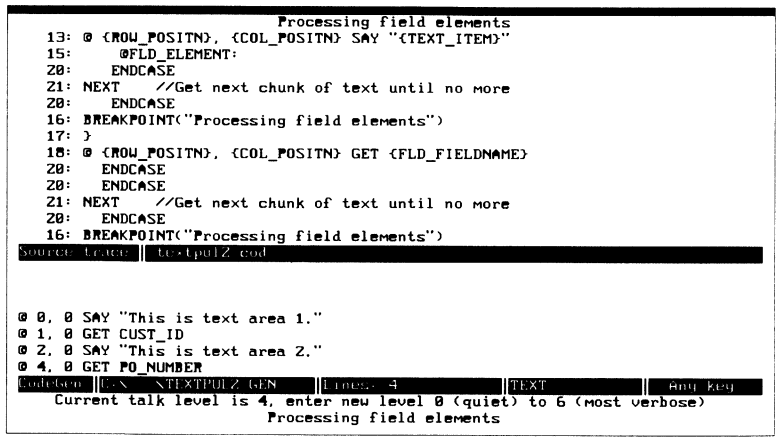


Figure 24-2 BREAKPOINT( ) message to change the DEBUG( ) level

To terminate looping between lines 16 and 20 of the code, enter 1. This sets DEBUG( ) to 1 and terminates stepwise command execution of the program.

BREAKPOINT( ) and DEBUG( ) are also described in Chapter 22.

Dtc and Dgen Error Messages

The template compiler, Dtc.exe, and the stand-alone interpreter, Dgen.exe, send error messages to the screen. This chapter describes the error messages produced by the compiler and the interpreter. Use the DOS redirection command (>) if you wish to send the messages to the printer or a disk file. The error messages take the following form:

<infile>
<line number>: <suspect code line>
--->'word causing error'
Error <#>: Error message

Template Compiler Error Messages

This section lists and explains the template compiler error messages.

101 String constant expected after INCLUDE keyword
Correct form is {include "abc.xyz";}. Filename must be in quotes.

102 Include file '<filename>' not found
No file by the specified name exists. The default path is the one current when DTC was invoked.

103 Access violation error trying to include file '<filename>'
This could occur if the specified file is a directory or on a network with read access denied, an operating system besides DOS, or DOS with file sharing turned on.

104 Error condition trying to include file '<filename>'
The compiler received an unexpected DOS error while trying to do an INCLUDE. Experiment with renaming or combining files, or put them on another drive.

105 Too many INCLUDEs (trying to include file '<filename>')
The compiler ran out of table space. Combine into fewer files.

106 Duplicate variable declaration
A variable was named twice — the same variable was declared twice in the same context, that is, globally or locally. Or, a local variable in a function has the same name as one of the argument variables.

107 Comma expected at this point in a variable declaration
The correct form is {VAR a, b} not {VAR a b}. Compilation continues with the missing comma assumed.

108 Unexpected termination of a variable declaration
A token other than a comma, semicolon, identifier, or right brace was encountered in a variable declaration, for example, {VAR a, b,ENDIF}. The compiler terminates the variable declaration and attempts to use the token as part of a statement.

109 Duplicate selector declared
The same selector was declared more than once. Two different selectors can have the same value, but cannot have the same name. For example, the second occurrence of *field* in the following would cause an error message: {SELECTORS field 1, text 2, field 1;}.

110 Prospective selector name is already in use
An identifier is already in use as something other than a selector, such as a variable name, function name, or label. Example:

```
{VAR moveleft, movecount;
 SELECTORS xyz 2, moveleft 29 //'moveleft' is already in use
                              // as a variable.}
```

111 Selector definition expected
A selector definition consists of an identifier followed by number. A number is expected following the SELECTORS keyword. If a comma follows a definition, another keyword plus definition is expected. Anything else causes this message to appear, and the extraneous tokens are skipped over.

112 Selector number expected here
The number expected as the second half of a selector definition is missing.

113 Comma expected at this point in a selector declaration
The comma used to separate selector definitions is missing, for example, {SELECTORS a 1 b 2} instead of {SELECTORS a 1, b 2}. Compilation continues, with the missing comma assumed.

114 Unexpected termination of a selector declaration
After a selector definition (identifier followed by number), a comma, semicolon, or right brace is expected.

115 Keyword THEN inserted here
The keyword THEN is expected at this point and is inserted just before the current token.

116 Keyword ELSE inserted here
The keyword ELSE is expected at this point and is inserted just before the current token.

117 Target label expected after THEN GOTO or ELSE GOTO
There is no destination specified for the GOTO. The GOTO is ignored, and the compiler attempts to continue parsing with the current token. For example, {IF a < b THEN GOTO -20} is treated as if it were {IF a < b THEN –20}.

118 Expression required following IF
There must be an expression between the IF and THEN keywords.

119 IF statement terminated
Normally an IF statement is terminated by a corresponding ENDIF. However, tokens that normally terminate other constructs (such as NEXT, or ENDCASE, or end of input, if not part of a corresponding FOREACH or CASE...OF nested within the IF statement) cause the open IF statement to be terminated. In the example {CASE x OF...IF a<b THEN c=d ENDCASE}, the compiler terminates the IF statement, on the assumption that the ENDCASE probably matches some preceding CASE...OF. Even if this assumption is not true, the IF statement is terminated.

120 This IF statement needs to return a value
This message appears when an IF statement follows a binary operator but does not yield a value. Since ELSE is missing from the IF statement in the following example, there is no value to take part in the addition: {a = 250 + IF b<c THEN 50 ENDIF}.

121 DO statement terminated
Normally a DO statement is terminated by a corresponding ENDDO. However, tokens that normally terminate other constructs (such as ENDIF, ENDCASE, or end of input, if not part of a corresponding IF or CASE...OF nested within the DO statement) cause the pending DO statement to be terminated. In the example {CASE x OF...DO PRINT(xyz) ENDCASE}, the compiler terminates the DO statement on the assumption that the ENDCASE matches some preceding CASE...OF. Even if this assumption is not true, the DO statement is terminated.

122 Variable name required following FOR
The FOR keyword must be followed by an existing variable name or a name that would be allowed as a variable (that is, one that starts with a letter and is not already in use as a selector).

123 Bad or missing TO expression
The TO keyword, used in the FOR loop header, must be followed by an expression.

124 Bad or missing STEP expression
The STEP keyword, used in the FOR loop header, must be followed by an expression.

125 FOR statement terminated
Normally a FOR statement is terminated by a corresponding NEXT. However, tokens that normally terminate other constructs (such as ENDIF, ENDCASE, or end of input, if not part of a corresponding IF or CASE...OF nested within the FOR statement) cause the open FOR statement to be terminated. In the example {CASE x OF...FOR xyz = 1 TO 20; PRINT(xyz) ENDCASE}, the compiler terminates the FOR statement on the assumption that the ENDCASE probably matches some preceding CASE...OF. Even if this assumption is not true, the FOR statement is terminated.

126 Selectors 'makec' and 'eoc' required before using FOREACH
The FOREACH loop creates a cursor and advances it through a list of items by using function calls to MAKEC and EOC. The standard built-in definitions file, which is usually INCLUDEd at the beginning of a template, defines these selectors for you.

127 Selector or value-yielding expression required following FOREACH
The FOREACH keyword must be followed by a selector or an expression returning a value. For example:

{FOREACH IF a THEN b ENDIF}
The IF statement does not yield a value.

{FOREACH} {FOREACH IN}
No expression supplied at all.

{FOREACH xyz}
{FOREACH xyz p}
{FOREACH xyz p IN q}
These are all correct syntax.

128 Expression required after IN
If the IN keyword is used in a FOREACH statement, an expression, usually a variable name, must follow the IN. For example, {FOREACH xyz IN} does not have an expression following the IN. {FOREACH xyz IN p} is the correct syntax.

129 FOREACH statement terminated
Normally a FOREACH statement is terminated by a corresponding NEXT. However, tokens that normally terminate other constructs (such as ENDIF, ENDCASE, or end of input, if not part of a corresponding IF or CASE...OF nested within the FOREACH statement) cause the open FOREACH statement to be terminated. In the example {CASE x OF...FOREACH xyz; PRINT(xyz) ENDCASE}, the compiler terminates the FOREACH statement on the assumption that the ENDCASE probably matches some preceding CASE...OF. Even if this assumption is not true, the FOREACH statement is terminated.

130 NEXT variable mismatch — loop header is:
An optional variable name may be supplied following the NEXT keyword. If it is supplied, it is checked to see that it matches the variable used in the header of the loop.

131 Not inside any loop currently
WHILE, UNTIL, LOOP, and EXIT all have no meaning outside of a DO, FOR, or FOREACH loop.

132 Expression required following WHILE or UNTIL
A value-yielding expression must follow the WHILE or UNTIL keywords. For example, {WHILE a>bB} is correct; {WHILE GOTO xyz} is incorrect.

133 Overriding previous definition of '<identifier>'
The ENUM statement allows you to define a given identifier as a synonym for a previously defined symbol. The identifier assigned is already being used for a variable name, user-defined function, label, or other attribute. The assignment in the ENUM statement takes precedence over the previous meaning of the identifier. This message is not given if the only previous use of the identifier was in ENUM statements.

134 Erroneous expression in ENUM
When an ENUM statement sets an identifier to be a synonym to a value that is an expression rather than a single token, the expression must contain only numeric constants or symbols ENUMed to constants. For example, in {ENUM xyz=49+'abc';}, 'abc' is not a numeric constant.

135 Value-yielding expression required between CASE and OF
You must specify an expression between CASE and OF that returns a value.

136 This CASE...ENDCASE construct has no case instances in it
There must be at least one CASE instance between OF and ENDCASE.

137 Case instances must be inside a CASE...ENDCASE construct
A construct like CASE 25 that occurs outside a CASE...ENDCASE statement would trigger this message.

138 Duplicate case instance
You used the same CASE constant in two different CASEs within the same construct, for example, {CASE a of 1: 'xyz'; 1: 'abc' ENDCASE}.

139 Variable used in a case instance - constant required
The CASE instance you have named has not been defined as a constant. The CASE...ENDCASE construct provides an efficient way to match the value of the selection expression against a list of constants, and to jump directly to the appropriate code for that case. This is easier to read, and executes faster than a series of IF...ENDIF statements. To compare the selection expression against a list of other expressions with values that will not be known until run time, use a series of IF...ENDIF statements.

140 Syntax error in CASE selection expression
The CASE...OF construct expects a single value-yielding expression between CASE and OF. Any punctuation or keywords that are not part of a valid expression will result in this message. For example, { CASE NEXT } has an inappropriate keyword, and { CASE ; } has an inappropriate semicolon. { CASE 2*myfunc(x,y) OF } is a valid expression.

141 Duplicate user-defined function name
A user-defined function of the given name has already been defined. The new definition will be compiled normally, but any calls will go to the original definition.

142 Nested function definition
Function definitions cannot be nested.

143 Missing name in function definition
The DEFINE keyword was not followed by an identifier. Compilation will continue as if the DEFINE keyword had not been seen; this may result in more error messages.

144 Comma expected in parameter list, inserted
A comma was missing in a parameter list. For example, {DEFINE x(a b)} was used instead of {DEFINE x(a,b)}. The parser assumes the comma and continues parsing.

145 Function definition parameter list terminated by unexpected token
The parameter list should be a list of identifiers separated by commas and terminated by a right parenthesis. Any other token terminates the parameter list.

146 Function call construct terminated
Function call arguments are separated by commas and end with a right parenthesis. A token that does not belong to the current expression in the list, and that is not a comma or a right parenthesis, causes this error. In the example {a = myfunc(2*b THEN);}, the THEN would cause this error.

147 Selector expected following '@'
This results from any token except a selector following the '@' operator. Compilation continues as if the '@' were not present.

148 Dot not allowed following constant, deleted
The dot is only used following a cursor to select an item from a list. Numeric constants must be integers; there is no decimal point.

149 Variable unknown
You used an identifier as if it were a variable name, but there was no declaration of that identifier as a variable. Note that if a variable is declared inside a function definition, it disappears as soon as the ENDDEF for that function is processed.

150 Terminating string delimiter missing
Example: { a ="There's no quote on the end of this string ...}

151 Unexpected token; skipped
The current token does not fit any current or pending constructs, and does not start a new one. Example:

{VAR a,b,c;
a = b + NEXT}
If no FOREACH construct is open, the NEXT is out of place.

{if a == b ; then}
The semicolon is out of place inside an IF condition.

152 Empty parentheses
A pair of parentheses contains no value-yielding expression, for example,
{a = b+()+c}.

153 Terminating open left parenthesis
A corresponding closing right parenthesis was not encountered before one of the following: a right brace, INCLUDE, THEN, ELSE, or ENDIF keyword, end of file, end of input, or semicolon. The opening left parenthesis is considered closed. Examples:

{a = b*(c+d; } Semicolon before closing right parenthesis
{a = b/(c*d} Missing right parenthesis

154 No pending left parenthesis
A right parenthesis was encountered with no corresponding left parenthesis open. Example: { a = b); }.

155 Duplicate label definition
The same name is used as a label in two different places. The first definition has precedence.

156 Bad target of GOTO; skipped
The GOTO keyword should be followed by an identifier which cannot be a selector. The GOTO is ignored.

157 Missing operand
The construct to the left of a binary operator does not yield a value. The compiler substitutes a zero for the missing operand.

158 Operator deleted
This is caused by an operator with no operand or by a binary operator following another binary operator. The star is deleted by the compiler in the following examples:

{ a = b* } No operand following the star
{ a = b + * c } The star follows plus, which is a binary operator here

159 Post-increment/decrement not supported
The '++' and '--' operators can precede a variable (pre-increment), but cannot follow a variable (post-increment). For example, in { VAR a,b,c; a = b++;}, the '++' following the b is illegal.

160 Requires lvalue
You attempted to store a value into something other than a variable, or into an undeclared variable. In the example { VAR number; numbr = 20;}, the misspelled name results in this message. In { 36 = 29; }, the lack of a variable results in the message.

161 Function never defined
A user-defined function call was made, but no definition for a function by that name was encountered.

162 Label never defined
A GOTO to a specified label occurred somewhere in the function just ended, but the label was not defined within the function. Or, a GOTO to a specified label occurred somewhere outside any function definitions but the label was not defined at that level.

163 Terminates open construct(s):
An unmatched terminating keyword, such as ENDIF, was encountered while an opening keyword, such as CASE, was pending. The compiler identifies the terminating keyword, then displays beneath it each opening keyword that is terminated.

164 No IF pending; skipped
ENDIF is seen when no IF is pending.

165 No CASE statement pending; skipped
ENDCASE is seen when no CASE is pending.

166 No DO pending; skipped
ENDDO is seen when no DO is pending.

167 No FOR or FOREACH pending; skipped
NEXT is seen when no FOR or FOREACH is pending.

168 No DEFINE in progress; skipped
ENDDEF is seen when no DEFINE is open.

901–999 (Internal: <internal-message>)
These numbers indicate internal errors within the compiler. Try rearranging the statement or expression that seems to have caused the error, and contact Borland Software Support.

Dgen Interpreter Error Messages

This section lists and describes the stand-alone interpreter error messages.

1 Insufficient Memory
This message indicates that there was not enough memory to open the resource file Dbase3.res when you initialized the interpreter.

2 System Error
This message indicates that a non-recoverable system error has occurred.

3 Initialization Error
This message indicates that the interpreter could not be initialized.

8 Could not open resource file
The required resource file Dbase3.res could not be opened.

34 Bad file name
This indicates that an invalid filename was referenced by an internal call.

36 File was read only
An attempt was made to write to or erase a file which is read-only.

120 Insufficient dynamic memory
The amount of dynamic RAM available to the interpreter is not enough to contain the layout and data variables used in the template program you are running. Increase available RAM by removing any TSR programs. If you're within dBASE IV, close files and clear menus.

121 Faulty read from I/O stream
An error in reading a DOS file has occurred. The problem may be a corrupted layout, text, or template file.

122 Template introduction overflow
Template program introduction (header) is limited to 4K. Edit the template source file and reduce the introduction.

123 Invalid template signature [...]
This indicates that the compiled template is corrupt. The text inside the square brackets is the invalid data found in place of the signature.

125 Template compiler/interpreter [...] mismatch
The version of the compiler used with the template is incompatible with the interpreter. The numbers inside the square brackets are the versions of the compiler and interpreter, respectively. Recompile the template with a compatible version of Dtc.exe.

126 Template not located [...]
The template file whose name is shown in the square brackets could not be found. The interpreter search path begins at the current DOS directory, next looks in the directory from which Dgen or dBASE IV was started, and last checks the directories in the DOS PATH.

127 Insufficient heap memory
This indicates there was not enough memory to initialize the interpreter. Increase available RAM by removing any TSR programs. If you're within dBASE IV, close files and clear menus.

128 Invalid frame on interpreter stack
Indicates an internal logic error in the template code output by Dtc.exe. Use the DEBUG() function to locate the code that caused the error. Then edit the source file to correct the wrong expression and recompile.

129 Nonzero template return value [...]
This occurs only from within dBASE IV. Templates that are run from the Control Center must always return 0. Other returned values result in this message.

130 Jump to undefined label
The compiler failed to resolve a jump instruction. To locate the bad instruction, examine the compiler listing assembly code for jump instructions flagged with an "-F".

131 Interpreter stack underflow
This message indicates an internal logic error in the template pcode output by the compiler. Trace the execution of the pcode using the DEBUG() function to locate the template code which caused the error. Then edit the source code to remove the error by simplifying the expression that caused it.

134 Argument mismatch for binary operator
This message indicates that a binary operator could not be processed because its operands were of the incorrect type.

137 Argument mismatch for built-in function
This message indicates that a function from the Builtin.def library could not be processed because its parameters were invalid or incorrect.

140 Object not located [...]
This message indicates that an attempt to load a layout object from disk failed. The name of the object which was not located is in the square brackets.

141 Unknown built-in function [...]
This message indicates that an unknown library function was referenced in the template. A reference to a layout attribute selector name followed by parentheses will cause this error. For example. ROW_POSITN() appears to the compiler as a library function call, but will generate this error message.

142 System error [...]
This message is a catch-all for internal system failures. The number in the square brackets refers to an internal code and is useful when reporting problems.

143 Too many files open
This message occurs only from the Control Center. The available file handles have been used up. Close some of the work areas in dBASE IV.

149 Command line error
This message occurs only from Dgen.exe and indicates that the command line syntax was incorrect. Check to see that you are using only valid command line switches.

157 Unknown opcode
This occurs when the compiled template is corrupted and contains an unknown opcode.

161 Built-in function [...] failed
This message indicates that a library function failed due to an internal problem. The number inside the square brackets indicates the selector ID of the function that failed.

SQL

SQL Basics
Starting SQL
SQL Queries
Joins and Subqueries
Combining SQL and dBASE IV
Embedding SQL Commands

SQL Basics

SQL (Structured Query Language) is an advanced database language that can be used in dBASE IV programs to define, display, and update data. Since it was developed in the mid-1970's, SQL has been adopted by many companies as a database language standard for both mainframe and minicomputer environments.

SQL consists of a small number of powerful commands. A single SQL command can often replace a number of dBASE IV data definition, data manipulation, and indexing commands.

You can execute SQL commands interactively at the SQL prompt, a command prompt similar to the dot prompt, or combine them with dBASE commands in a database application program. The SQL commands provided by dBASE IV are a subset of those in IBM's DATABASE 2 (DB2) and SQL/DS mainframe computer database products.

This chapter describes SQL database concepts, with an emphasis on using SQL commands in dBASE IV programs.

Relational Database Language

SQL is a relational database language. In a relational database, you see data as a collection of related tables.

SQL Tables

Each table in a relational database is a set of data that is arranged in rows and columns. The intersection of a row and a column contains a data value.

A SQL *table* can be likened to a dBASE database file. Table columns and rows correspond to the fields and records of a database file.

Arranging data in table form makes it easier to visualize. Figure 26-1 is an example of a SQL table.

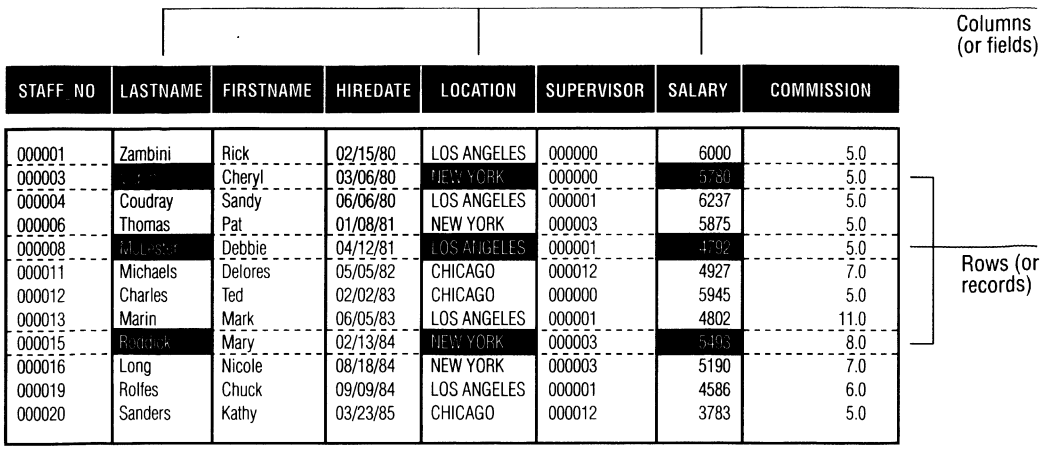


Figure 26-1 SQL table

SQL Views

The SQL language is used to create another type of table called a *view.* A SQL view is a subset of the rows and columns of one or more tables.

A view is often referred to as a *virtual* table because it does not actually contain data. Rather, the view provides a window through which you can look at data in the underlying or *base* tables on which the view is defined. A SQL view through which you look at data can be likened to a dBASE view query.

An *updatable* view is similar to a normal view, except that you can use it to update and insert data in the view's base tables. Such a view can be compared with a dBASE update query.

Data displayed in a view is dynamic. As data in the base tables changes, the data seen through the view also changes. Similarly, when you change the data in an updatable view, data in the base tables is also changed.

Views are discussed in more detail in Chapter 27.

Nonprocedural, Set-Oriented Data Access

You use a dBASE IV command to specify what you want to do with the data in a single database file and how to do it. In SQL, you also use commands to specify operations. However, in SQL:

- Each command can encompass operations on more than one table at a time.
- You specify the operation that you want to perform on table data, not a procedure for accessing it. The SQL system itself determines the best procedure for performing the operation.

- A SQL command typically operates on *sets* of records rather than on individual records.

This *nonprocedural, set-oriented* approach is different from that of other database languages, including dBASE IV. Other languages require you to understand how and where data is stored on your computer before you can use commands to get to it. Then, you can access only one record at a time.

You enter dBASE commands to open database files in work areas, and move record pointers within each open file to access individual records. In SQL, you enter a single command to specify the set of data that you want to retrieve from one or more tables. You use a similar set-oriented approach to inserting, updating, and deleting data.

By reducing the number of commands that you need to enter for these operations, SQL saves you time. Figure 26-2 shows the number of dBASE commands that you'd need to enter to perform a typical SQL query.

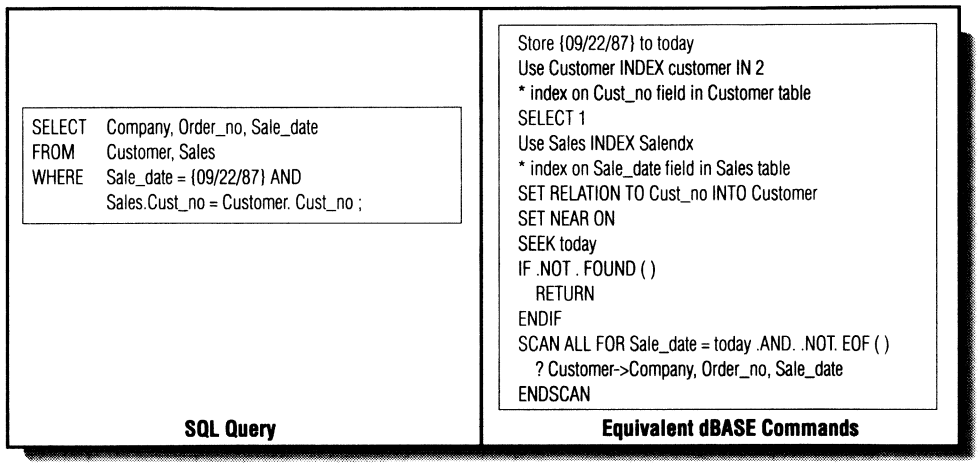


Figure 26-2 dBASE commands needed to perform a SQL query

Interactive and Embedded Use

Using *interactive* SQL, you enter commands one at a time at the SQL prompt and the results are displayed immediately on your screen. This is similar to entering dBASE IV commands at the dot prompt.

Using *embedded* SQL, you create dBASE IV program files that include SQL commands. In mainframe and minicomputer environments, SQL is typically used with programming languages such as COBOL, FORTRAN, or C to develop database applications. In dBASE IV, you use SQL with dBASE IV commands to create database applications that run on microcomputer systems.

dBASE IV provides a rich programming language for using SQL capabilities. You can use dBASE IV commands to construct menus, design forms for report and data entry, and control the operation of a program.

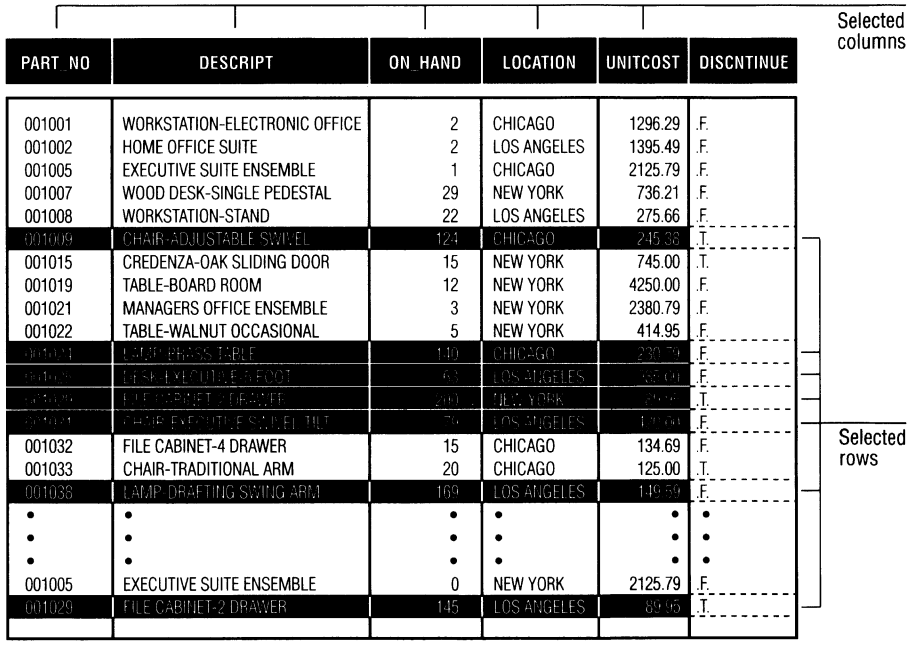
Using SQL

Consider the following SQL query:

```
SELECT Part_no, Descript, On_hand, Location, Unitcost
  FROM Inventry
  WHERE On_hand > 50;
```

Figure 26-3 shows the result of the query, also referred to as a result table. In this example, a SELECT command retrieves information from the Inventry table for each part whose in-stock quantity is greater than 50.

Inventry table



PART_NO	DESCRIPT	ON_HAND	LOCATION	UNITCOST	DISCNTINUE
001001	WORKSTATION-ELECTRONIC OFFICE	2	CHICAGO	1296.29	.F.
001002	HOME OFFICE SUITE	2	LOS ANGELES	1395.49	.F.
001005	EXECUTIVE SUITE ENSEMBLE	1	CHICAGO	2125.79	.F.
001007	WOOD DESK-SINGLE PEDESTAL	29	NEW YORK	736.21	.F.
001008	WORKSTATION-STAND	22	LOS ANGELES	275.66	.F.
001009	CHAIR-ADJUSTABLE SWIVEL	124	CHICAGO	245.36	.T.
001015	CREDENZA-OAK SLIDING DOOR	15	NEW YORK	745.00	.T.
001019	TABLE-BOARD ROOM	12	NEW YORK	4250.00	.F.
001021	MANAGERS OFFICE ENSEMBLE	3	NEW YORK	2380.79	.F.
001022	TABLE-WALNUT OCCASIONAL	5	NEW YORK	414.95	.F.
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	.F.
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	.F.
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	.T.
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	.F.
001032	FILE CABINET-4 DRAWER	15	CHICAGO	134.69	.F.
001033	CHAIR-TRADITIONAL ARM	20	CHICAGO	125.00	.T.
001036	LAMP-DRAFTING SWING ARM	169	LOS ANGELES	149.59	.F.
•	•	•	•	•	•
•	•	•	•	•	•
•	•	•	•	•	•
001005	EXECUTIVE SUITE ENSEMBLE	0	NEW YORK	2125.79	.F.
001029	FILE CABINET-2 DRAWER	145	LOS ANGELES	89.95	.T.

Figure 26-3 Result table for a SQL query

The SELECT command contains the following *clauses:*

- A SELECT clause that identifies the column information to be displayed.
- A FROM clause that identifies the table that contains the data.
- An optional WHERE clause that limits the rows of the result table to those that satisfy the condition *On_hand > 50.*

Switching Between dBASE and SQL Modes

dBASE IV provides two modes in which to enter commands. In the default dBASE mode, you use only dBASE commands; in SQL mode, you use SQL commands and most dBASE commands and functions.

To use SQL data definition and data manipulation commands in place of the corresponding dBASE IV commands, use SQL mode. To use the full set of dBASE commands and functions, use dBASE mode.

You can easily switch back and forth between the two modes. To switch to SQL from the dBASE mode, enter the SET SQL ON command at the dot prompt. To switch back again, enter SET SQL OFF at the SQL prompt. You can use the SET SQL command only interactively.

In an embedded SQL application, you use a program file's extension to specify the mode (dBASE or SQL) in which you want the file to execute. Use the .prg extension for program files that contain only dBASE commands and the .prs extension for SQL program files. During execution, dBASE IV automatically switches modes, depending on the extension of the program file that is being called.

Using dBASE Commands and Functions in SQL Mode

As implemented in dBASE IV, a SQL table is a dBASE database file that is defined as a SQL table. Therefore, although you can't use SQL commands in dBASE mode, you can use most dBASE commands and functions in SQL mode to operate on SQL tables. This includes dBASE commands that perform functions similar to those of SQL commands, for example, BROWSE, EDIT, and APPEND.

SQL commands provide an alternative to using dBASE commands to define and manipulate data. You can use SQL or dBASE commands, depending on your familiarity with either language.

NOTE
Some of the dBASE commands that are available in SQL mode, such as CREATE/MODIFY STRUCTURE and CREATE/MODIFY QUERY, can be used only to define dBASE database files and queries in SQL mode, not to define SQL tables and queries.

Refer to the Using the Menu System section in the next chapter for information about using dBASE menu system functions from the Control Center. See Using dBASE Commands and Functions in SQL Mode in Chapter 30 for more information about combining dBASE and SQL commands.

Single-User and Network Operation

dBASE IV provides single-user and network support for both interactive and embedded SQL operations. SQL program files that you create in dBASE IV will automatically run on a network.

When more than one user is using dBASE IV on a network, table data is automatically *locked* during SQL operations that insert, update, or delete data. If an operation cannot be performed because the data is currently in use, you can instruct dBASE IV to retry the operation until it is successful.

Transaction Processing

dBASE IV also provides transaction processing with BEGIN TRANSACTION, END TRANSACTION, and ROLLBACK commands. If an operation cannot be completed, you can specify ROLLBACK to undo any changes to tables.

Summary

This chapter introduced you to SQL database concepts and to using SQL commands in dBASE IV. Some of the major concepts are summarized below.

SQL table:
A set of data arranged in rows and columns that is analogous to the records and fields in a database file. The intersection of a row and a column contains a data value. Also known as a *base* table.

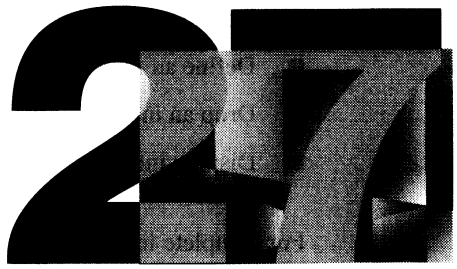
SQL view:
A *virtual* table that contains no data of its own, but draws on a subset of data from the rows and columns in one or more SQL base tables. Through a view, you can look at this subset of data. As data in the base tables changes, so does the data seen through the view. If a view is *updatable,* you can use it to insert and update in the base tables.

Nonprocedural, set-oriented data access:
You use a SQL command to specify what you want to do with data, not a procedure for doing it. Instead of operating on one record at a time, as a dBASE command does, a SQL command operates on sets of row data.

Interactive and embedded SQL:
In *interactive* SQL, you enter commands one by one at a SQL prompt on a command line. In *embedded* SQL, you use SQL commands within an application program written in the dBASE language. A single SQL command can do the work of a number of dBASE commands.

dBASE mode versus SQL mode:
In dBASE mode (at the dot prompt), you can enter only dBASE commands and functions. In SQL mode (at the SQL prompt), you can enter all SQL commands and most dBASE commands and functions. You can switch easily between the two modes. You can use SQL on a single-user computer as well as on a network, just as you can dBASE IV.

Starting SQL



This chapter shows you how to use SQL commands interactively, at the SQL prompt, to build and use SQL databases. You will learn how to:

- Display the SQL prompt
- Use a dBASE catalog file to maintain information about SQL tables
- Start a SQL database
- Enter SQL commands on the SQL prompt command line or in an editing window
- Use the dBASE IV history buffer to redisplay SQL commands for entry
- Display the Control Center to perform menu system functions on SQL database files
- Use the dBASE IV Help system to get information about SQL commands
- Create a new SQL database
- Display information about existing SQL databases
- Activate and deactivate a database
- Drop a database
- Create a new SQL table and add data to it
- Update and delete table data
- Restructure a table by adding columns or changing the table definition
- Drop a table
- Create a SQL view of the data in one or more SQL tables
- Use a view to restructure the information in a SQL table without changing the table's structure or definition
- Drop a view

- Define a synonym, or alternate name, for a table or view
- Define an index to speed access to table data
- Drop an index
- Display information about tables and views stored in the SQL system catalog tables

For complete information about any SQL command used in this manual, refer to Chapter 6 of *Language Reference.*

Preparing for This Chapter

Start dBASE IV:

1. Enter dbase at the operating system prompt.
2. If the Control Center is displayed, choose **Exit to dot prompt** from the **Exit** menu to access the dot prompt.

Samples Database

When you installed dBASE IV and its sample files, a SQL database named Samples was created in the \DBASE\SAMPLES directory. The Samples database contains the tables that you will be using in the exercises in this chapter and in the remaining chapters of this manual.

The sample tables, listed in Table 27-1, are for an order entry and invoicing system.

Table 27-1 Sample tables

SQL Table	Description
Customer	Information about customers
Staff	Information about salespeople who service customers
Inventry	Information about products for sale to customers
Assembly	Information about the parts required to build each product
Sales	Information about customer orders of products
Items	Information about the quantity of products needed for each order, and whether they have been shipped (.F. = no; .T. = yes)

All of the tables are related through values in common columns, as shown in Figure 27-1:

- Staff and Customer tables are related through the Sales table, which contains both Staff_no and Cust_no columns.
- Sales and Items tables are related through Order_no columns.
- Items, Assembly, and Inventry tables are related through Part_no columns and the Assembly column, which contains Part_no values.

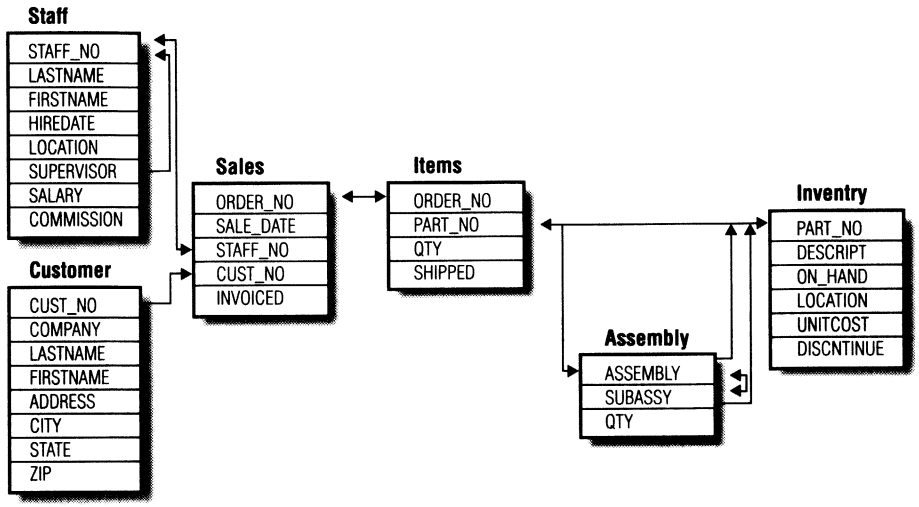


Figure 27-1 Table relationships

For example, the Staff table can be linked with the Inventry table through the values of Staff_no, Order_no, and Part_no.

The appendix lists the data in each table in the Samples database. Take a few minutes to become familiar with the tables. Keep a copy of the tables handy as you do the exercises in this chapter and later chapters.

Accessing the SQL Prompt

Before you can start entering SQL commands, you must switch to SQL mode.

1. Enter:

```
. SET SQL ON
SQL.
```

The SQL prompt appears, indicating that you have entered SQL mode. To re-enter dBASE mode and redisplay the dot prompt, enter SET SQL OFF

NOTE

- *You can enter SQL mode automatically when you start dBASE IV by including the SQL=ON command in the dBASE IV configuration file, Config.db. Refer to the Installation and Configuration section of* Getting Started with dBASE IV.
- *If the SQLHOME command is not set in your Config.db file, entering SET SQL ON displays an error message. Refer to the Installation and Configuration section of* Getting Started with dBASE IV.
- *When you switch to SQL mode, any work area that is open in dBASE mode remains open. To minimize the possibility of running out of work areas during the SQL session, close open work areas in dBASE mode before switching to SQL.*

2. To make Samples the default directory, enter:

```
SQL. SET DIRECTORY TO SAMPLES
C:\DBASE\SAMPLES
```

Using dBASE Catalogs

While in SQL mode, you can use dBASE catalog files to maintain information about database files that are defined as SQL tables. When SET CATALOG is ON, the following SQL commands affect the current dBASE catalog:

- START DATABASE — the database files for all tables in the database are added to the catalog.
- DROP DATABASE — the database files for all tables in the database are removed from the catalog.
- SELECT...SAVE TO TEMP — the temporary table is added to the catalog. When you change databases or end the SQL session, the table is removed from the file unless the KEEP option was specified. (Refer to the SAVE TO TEMP Clause section in Chapter 30.)
- CREATE TABLE — the database file for the new table is added to the catalog.
- DROP TABLE — the database file for the table is removed from the catalog.

Create a new dBASE catalog to track the tables in the Samples database for your order entry and invoicing application.

1. Enter:

```
SQL. SET CATALOG TO Orders
```

dBASE IV prompts you to confirm your decision to create a new catalog.

2. Respond with Yes.

   dBASE IV prompts you to enter a description of the new catalog.

3. Enter the following description for the new catalog:

```
SQL tables for order entry/invoicing application
```

   dBASE IV displays the message:

```
File catalog empty
```

Entering SQL Commands

In SQL mode, you can enter a SQL command directly on the command line or in an editing window.

Using the Command Line

You can type a SQL command of up to 254 characters on the command line. Terminate the command using a semicolon and press ↵ to enter the command.

Enter:

```
SQL. START DATABASE Samples;
Database SAMPLES started
```

dBASE IV opens the Samples database. You must start a database before you can use SQL commands to access data in any table or view.

NOTE
If you installed the sample files when you installed dBASE IV, the command SQLDATABASE=SAMPLES was included in your Config.db file. This command causes the Samples database to be started automatically each time that you enter SQL mode. Therefore, entering the START DATABASE command was not strictly necessary to start Samples. You can modify the SQLDATABASE command in Config.db so that another database is automatically started when you enter SQL mode, or remove the command altogether so that no database is started. Any database named in the SQLDATABASE or START DATABASE command must already exist. SQL databases are created using the CREATE DATABASE command, discussed later in this chapter.

While entering a command, use the navigation and editing keys listed in Table 27-2. These keys provide the same functions as when you're entering dBASE commands.

Table 27-2 SQL navigation and editing keys

Key	Function
→	Moves the cursor one character to the right.
←	Moves the cursor one character to the left.
↑	At the SQL prompt, displays previously entered commands stored in the history buffer. In the editing window, moves the cursor to the previous line.
↓	While viewing commands stored in the history buffer, displays previously entered commands, and finally a blank command line. In the editing window, moves the cursor to the next line.
Ctrl-←	Moves the cursor to the previous word.
Ctrl-→	Moves the cursor to the next word.
Home	At the SQL prompt, moves the cursor to the beginning of the command line. In the editing window, moves the cursor to the beginning of the current line.
End	At the SQL prompt, moves the cursor to the end of the command line. In the editing window, moves the cursor to the end of the current line.
Backspace	Deletes one character to the left of the cursor.
Del	Deletes one character at the cursor position.
Ins	Switches modes between inserting and overwriting characters.
Num Lock	Turns Numeric mode on and off (must be off for navigation keys on the numeric keypad to operate).
Caps Lock	Switches uppercase mode on and off.
F1 Help	At the SQL prompt, displays a Help screen for the current command. In the editing window, displays information about the editing window.
Ctrl-Home	Opens the full-screen editing window.
Ctrl-End	Enters the command currently displayed in the editing window and redisplays the SQL prompt.
↵	At the SQL prompt, executes the command currently entered on the command line. In the editing window, moves the cursor to the next line.

Using the Editing Window

In the editing window, you can enter a SQL command of up to 1,024 characters on more than one line. Press ↵ to end each line of the command and begin another. Terminate the command with a semicolon.

1. From anywhere on the SQL prompt command line, press **Ctrl-Home**.

   dBASE IV opens the editing window. If you had already entered a partial or complete command on the command line, this would be displayed in the window.

2. Type the following command to display data from the Inventry table of the Samples database:

```
SELECT Part_no, Descript, On_hand
  FROM Inventry;
```

3. Press **Ctrl-End** to enter the command.

   dBASE IV redisplays the command on the command line and then displays the *result table.* The result table contains part number, description, and quantity information for all the items in stock:

TIP

To display a long result table one page at a time, enter SET PAUSE ON at the SQL prompt or press ← to halt and resume scrolling of data. Specify SET HEADING OFF to remove column headings from the display.

```
PART_NO  DESCRIPT                      ON_HAND
001001   WORKSTATION-ELECTRONIC OFFICE       2
001002   HOME OFFICE SUITE                   2
001005   EXECUTIVE SUITE ENSEMBLE            1
    .              .                         .
    .              .                         .
    .              .                         .

001029    FILE CABINET-2 DRAWER              145
```

TIP

To simplify proofreading a complex command before entry, use **Tab** , **Spacebar**, *and ↵ to format the command in the editing window. If the command doesn't provide the result you wanted, press* **Ctrl-Home** *to redisplay it in the editing window exactly as you formatted it.*

SQL Command Syntax

A SQL command begins with a *keyword,* such as INSERT or SELECT, that names the basic operation to be performed. Many SQL commands also have one or more keyword phrases, or *clauses,* that let you tailor the command for a specific task.

Each SQL command must:

- Specify the data that you want (a set of rows in one or more tables).
- Indicate the action to be performed with the specified data.

For example, the basic syntax of SELECT, the command that you used earlier to retrieve data, is:

```
SELECT <columns>
FROM <tables>
[WHERE <condition>]...;
```

The clauses in this syntax are:

- SELECT — specifies the columns that are to appear in the result.
- FROM — specifies the tables or views from which rows and columns are to be retrieved.
- WHERE (optional) — limits result rows to those that satisfy a specified condition.

The following conventions are used in this section of the manual to specify SQL command syntax:

- Command and clause keywords are shown in capital letters. However, you can enter keywords in uppercase, lowercase, or a combination of the two.
- An item for which you supply a value is indicated by angle brackets (< >). Enter the value, but not the brackets themselves.
- An optional item is indicated by square brackets ([ ]). Enter the item, but not the brackets themselves.
- An item followed by an ellipsis (...) can be repeated as many times as needed.
- Items separated by a slash (/) are mutually exclusive.

Now, include a WHERE clause in the command that you entered earlier to limit the rows that appear in the result table.

Enter:

```
SQL. SELECT Part_no, Descript, On_hand
       FROM Inventry
       WHERE Location = "LOS ANGELES";

PART_NO     DESCRIPT                              ON_HAND
001002      HOME OFFICE SUITE                           2
001008      WORKSTATION-STAND                          22
001025      DESK-EXECUTIVE-5 FOOT                      63
001031      CHAIR-EXECUTIVE SWIVEL/TILT                79
001038      LAMP-DRAFTING SWING ARM                   169
001007      WOOD DESK-SINGLE PEDESTAL                  62
001013      CHAIR-MODERN PNEUMATIC                     35
001032      FILE CABINET-4 DRAWER                      71
001001      WORKSTATION-ELECTRONIC OFFICE               3
001029      FILE CABINET-2 DRAWER                     145
```

dBASE IV displays the same information as before, but only for items stocked in Los Angeles.

Figure 27-2 shows how rows and columns are selected using this command.

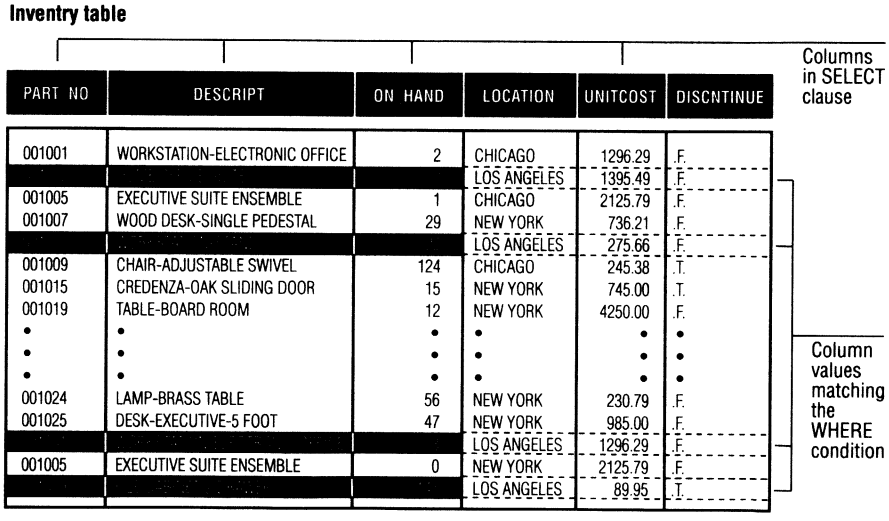


Figure 27-2 SELECT query

Re-entering SQL Commands

You can press ↑ to redisplay previously entered SQL commands that are stored in the dBASE IV history buffer. Once you have redisplayed a SQL command from the history buffer, you can:

- Press ↵ to re-enter it.
- Press **Ctrl-Home** to display it in the full-screen editing window and then use the edit keys to modify it before re-entering. The command is redisplayed in the same format as you originally entered it.

Using the Menu System

While in SQL mode, you can use menu system functions from the Control Center much as you can in dBASE mode.

1. At the SQL prompt, press **F2 Assist**.

   The Control Center appears. The catalog is set to Orders.cat and the tables in the Samples database are displayed in the **Data** panel.

2. Select one of the SQL tables from the panel by highlighting its name and pressing ↵.

   The prompt box appears, allowing you to open the file that contains the table, display and edit table data, and add rows.

3. Press **Esc** to remove the prompt box. To return to the SQL prompt, select **Exit to dot prompt** from the **Exit** menu.

NOTE
In SQL mode, you cannot use the menu system to create a SQL table or modify the structure of an existing table. These functions must be performed using the SQL commands described in this chapter. However, you can use the menu system to create and modify dBASE database files that can later be defined as SQL tables, as described in Chapter 30.

Using Control Center panels in SQL mode, you also can use table data to:

- Create, modify, and print reports and labels
- Create and run applications
- Create queries

Getting Help

You can get information about using SQL commands and the dBASE commands available in SQL mode by pressing **F1 Help** or entering the HELP command at the SQL prompt. This displays a Help screen similar to the one shown in Figure 27-3.

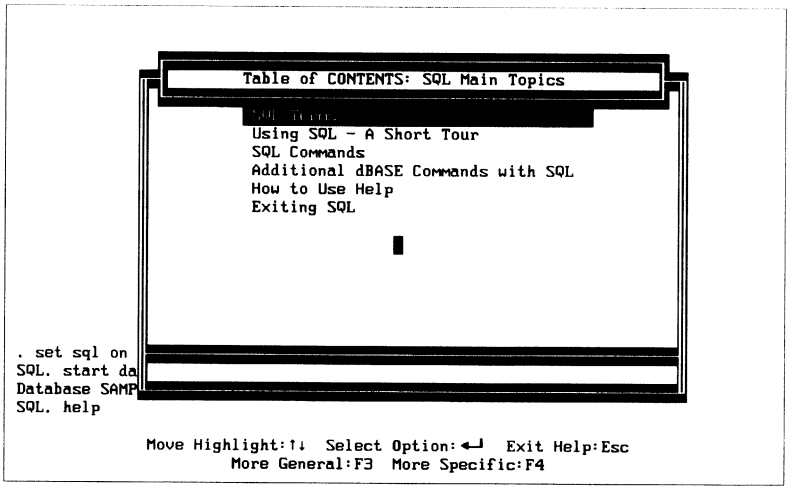


Figure 27-3 SQL Help screen

For information about navigating through Help screens, refer to the Getting Help sections in Chapters 8 and 13 of *Getting Started with dBASE IV.*

To get help for a specific SQL or dBASE command or keyword, type HELP followed by the name of the command or keyword.

1. Enter:

```
SQL. HELP SELECT
```

   A Help box describing the syntax of the SELECT command appears.

NOTE
When you enter a dBASE command (for example, HELP) in SQL mode, an error will occur if you terminate the command with a semicolon, as you do SQL commands.

2. Press **Esc** to return the cursor to the SQL prompt.

   The command syntax remains displayed to assist you in entering a command.

3. Press ↵ to clear the display.

If you've begun entering a command on the command line but haven't pressed ↵ to enter the command, you can get help in completing the command by pressing **F1 Help**.

4. Type:

```
SQL. SELECT
```

5. Press **F1 Help**.

   The Help box for the SELECT command syntax is displayed.

6. Press **Esc** twice to clear the display.

   If you type a command incorrectly and try to enter it, dBASE IV will display an error box with Help as one of its options.

7. Enter:

```
SQL. SELCT * FROM Staff;
```

   An error box similar to the one shown in Figure 27-4 appears. The error box provides the following options:

   - **Cancel** — discard the command and clear the command line.
   - **Edit** — correct the command.
   - **Help** — display information about the error or the command, or display a list of topics from which you can select.

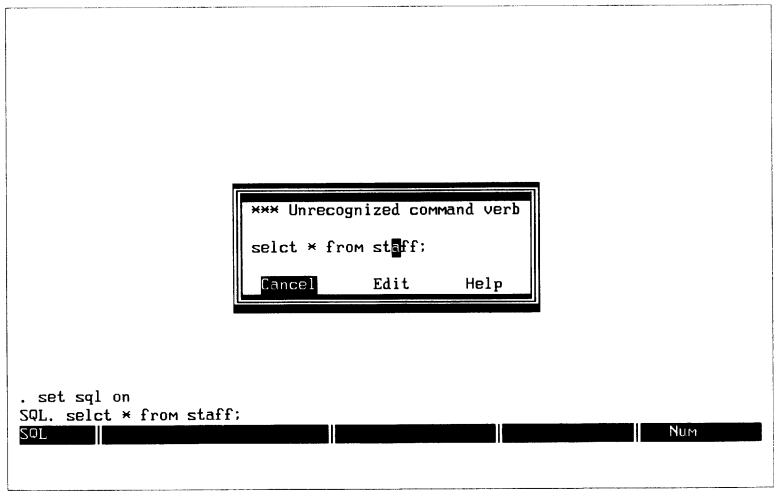


Figure 27-4 SQL error box

8. Press ↵ to cancel the command and clear the display.

SQL Databases

SQL uses a *database* to store all information related to an application. A SQL database contains the following types of SQL *objects:*

- Table — the basic structure for data in a SQL database (sometimes called a *base table).*
- View — a *virtual* table providing a *view* of data that consists of selected rows and columns from one or more base tables. The view is updated as data in the base tables changes.
- Synonym — an alternate name for a table or view.
- Index — a structure created for a base table that speeds access to data and helps maintain data integrity.
- Catalog — a set of system-controlled tables that describes each user database and its contents. A catalog differs from a dBASE .cat file, which is a listing of the database files needed for a particular job.

Before you can begin creating tables for a database, you must first create the database. Before you can use existing tables, you must start the database that contains the tables. In dBASE IV SQL, each database is assigned a directory on disk in which all of its tables and related files are stored.

In each database directory, a set of catalog tables is stored. A master catalog table, Sysdbs, stored in the SQLHOME directory, keeps track of all of the databases that you or other users create. The catalog tables and Sysdbs are discussed in the SQL System Catalogs section, later in this chapter.

SQL provides five commands for SQL databases: CREATE DATABASE, SHOW DATABASE, START DATABASE, STOP DATABASE, and DROP DATABASE.

NOTE
Designing an efficient relational database for an application is a complex process that is beyond the scope of this manual. Refer to a reference on relational database management systems.

Creating a Database

Use the CREATE DATABASE command to create a database. Enter:

```
SQL. CREATE DATABASE Myapp;
Database MYAPP created
```

dBASE IV creates a database named Myapp and starts it. The Samples database, started earlier, is closed.

NOTE

- *Each database name must be unique. The number of databases that you can create is limited only by the number of directories allowed by the operating system.*
- *You can specify an explicit operating system path (up to 64 characters) preceding the database name. Do not insert a space between the path and the name of the database. If a specified directory already exists, no new directory is created. Instead, that directory is defined as a SQL database and a complete set of catalog tables is copied into it. If you do not specify a path, the new database directory is created as a subdirectory of the default directory.*

Listing Current Databases

Use the SHOW DATABASE command to list information about all existing SQL databases.

To list the databases that have been created up to this point, enter:

```
SQL. SHOW DATABASE;

Existing databases are:
NAME      CREATOR    CREATED    PATH
SAMPLES   SYSTEM     09/30/88   C:\DBASE\SAMPLES
MYAPP                09/30/88   C:\DBASE\MYAPP
```

The **Creator** column contains an entry only for the Samples database, or for a database created after dBASE IV password protection is installed using PROTECT.

Activating a Database

The START DATABASE command activates an existing database. You can use only one database at a time. Enter:

```
SQL. START DATABASE SAMPLES;
Database SAMPLES started
```

dBASE IV closes Myapp and starts Samples. Any SQL command that you enter now applies to tables in the Samples database. You also can close a database using the STOP DATABASE command.

Stopping a Database

Use the STOP DATABASE command to close the current database. Enter:

```
SQL. STOP DATABASE;
Current database stopped
```

This command closes the Samples database, leaving no database open.

Dropping a Database

You use the DROP DATABASE command to delete a database that you no longer need. DROP DATABASE deletes all objects created in the database, along with associated database and index files, the SQL catalog files, and the entry for the database in the Sysdbs master catalog table. The database directory itself is not deleted.

WARNING
After you drop a database, you cannot recover its tables and related files.

You cannot drop a database if:

- The database is still active. You must first close it by entering the STOP DATABASE command or by switching to another database.
- Someone else on a network is currently using the database. If you try, an error message appears.

1. To drop the Myapp database, enter:

```
SQL. DROP DATABASE Myapp;
```

   dBASE IV displays a box warning you that all SQL tables in Myapp will be dropped.

2. Select **Proceed** to drop the database.

   The message **Database MYAPP dropped** is displayed.

NOTE
Entering DROP DATABASE removes entries for all tables in the database from an open dBASE IV catalog (.cat) file unless SET CATALOG is OFF.

SQL Tables

Tables are the basic components of a SQL database. Before you can manipulate data in a database, you must insert data in its tables.

SQL provides a number of commands that affect tables:

- CREATE TABLE — defines the structure of a new table
- INSERT — adds one or more rows to a table
- UPDATE — changes table data

- DELETE — removes one or more rows from a table
- ALTER TABLE — adds columns to an existing table
- DROP — deletes a table

Creating Tables

You use the CREATE TABLE command to specify the name of a new table and define its columns. A table name must be unique within the database, can be up to eight characters long, and must begin with a letter.

When you enter CREATE TABLE, dBASE IV creates a database (.dbf) file for the table. Thus, the limits on SQL table size are the same as for database files: 255 columns of up to 4,000 bytes per row.

Besides creating a database file, dBASE IV updates the database catalog tables to include the new table definition. A description of the table is entered in the Systabls table and descriptions of its columns are entered in the Syscols table.

1. Restart the Samples database by entering:

```
SQL. START DATABASE Samples;
Database SAMPLES started
```

2. Press **Ctrl-Home**.

   The editing window is opened.

3. Enter:

```
CREATE TABLE Staff1
   (Staff_no CHAR(6),
   Lastname CHAR(15),
   Firstname CHAR(10),
   Hiredate DATE,
   Location CHAR(15),
   Supervisor CHAR(6),
   Salary NUMERIC(6,0),
   Commission NUMERIC(4,1));
```

4. Press **Ctrl-End** to enter the command.

   The command is redisplayed on the command line and the message **Table Staff1 created** appears. The Staff1 table contains eight columns: Staff_no, Lastname, Firstname, Hiredate, Location, Supervisor, Salary, and Commission.

   In the command, you assign each column a *data* type that defines the kind of data the column can hold. For some data types (for example, the character data type), you must also specify a *column width,* the maximum number of characters or digits allowed in a column.

The data types that you can specify when you create a SQL table are listed in Table 27-3.

Table 27-3 SQL data types

Data Type	Description
SMALLINT	Holds an integer of up to six digits (including sign). Values entered can range from –99,999 to 999,999.
INTEGER	Holds an integer containing up to 11 digits (including sign). Values entered can range from –9,999,999,999 to 99,999,999,999.
DECIMAL(x,y)	Holds a signed fixed decimal point number with x total digits (including sign) and y decimal places (significant digits to the right of the decimal point). *x* can range from 1 to 19 and *y* can range from 0 to 18. For example, DECIMAL(6,2) specifies entry of values ranging between –999.99 and 9999.99. Precision for fixed decimal point numbers is specified using the dBASE SET PRECISION command.
NUMERIC(x,y)	Holds a signed fixed decimal point number with *x* total digits (including sign and decimal point) and *y* decimal places (significant digits to the right of the decimal point). *x* can range from 1 to 20 and *y* can range from 0 to 18. For example, NUMERIC(6,2) specifies entry of values ranging between –99.99 and 999.99. Precision for fixed decimal point numbers is specified using the SET PRECISION command.
FLOAT(x,y)	Holds a signed floating point number with *x* total digits (including sign and decimal point) and *y* decimal places (significant digits to the right of the decimal point). *x* can range from 1 to 20 and y can range from 0 to 18. The range of numbers you can store is $0.1*10^{-307}$ to $0.9*10^{+308}$. A number may be specified using scientific (exponential) notation, for example, –9.99E + 235.
CHAR(n)	Holds a character string of up to *n* characters. *n* can range from 1 to 254. Values can be entered from character columns, character type memory variables, or a character string.
DATE	Holds a date in the format specified by the SET DATE and SET CENTURY commands. The default format is *mm/dd/yy*. Values are entered from date-type columns, date-type memory variables, or DATE strings specified as { / / } or converted with the dBASE CTOD( ) function, for example, *CTOD("02/15/86").*
LOGICAL	Holds logical true or false values. *.T.* represents a true value and *.F.* represents a false value. Values are entered from dBASE LOGICAL memory variables or columns, or by the .T., .t., .Y., .y., .F., .f., .N., and .n. constants.

Figure 27-5 shows the Staff sample table, which is identical in structure to the Staff1 table you just created. The table contains a row of information for each salesperson.

STAFF_NO	LASTNAME	FIRSTNAME	HIREDATE	LOCATION	SUPERVISOR	SALARY	COMMISSION
000001	Zambini	Rick	02/15/80	LOS ANGELES	000000	6000	5.0
000003	Vidoni	Cheryl	03/06/80	NEW YORK	000000	5780	5.0
000004	Coudray	Sandy	06/06/80	LOS ANGELES	000001	6237	5.0
000006	Thomas	Pat	01/08/81	NEW YORK	000003	5875	5.0
000008	McLester	Debbie	04/12/81	LOS ANGELES	000001	4792	5.0
000011	Michaels	Delores	05/05/82	CHICAGO	000012	4927	7.0
000012	Charles	Ted	02/02/83	CHICAGO	000000	5945	5.0
000013	Marin	Mark	06/05/83	LOS ANGELES	000001	4802	11.0
000015	Roddick	Mary	02/13/84	NEW YORK	000003	5493	8.0
000016	Long	Nicole	08/18/84	NEW YORK	000003	5190	7.0
000019	Rolfes	Chuck	09/09/84	LOS ANGELES	000001	4586	6.0
000020	Sanders	Kathy	03/23/85	CHICAGO	000012	3783	5.0

Figure 27-5 Sample database table

NOTE
Entering CREATE TABLE adds the new table to an open dBASE IV catalog (.cat) file unless SET CATALOG is OFF.

Inserting Data

Use the INSERT command to add new rows to a table. Using INSERT, you can specify individual values to be inserted for each row, or you can SELECT values from another table. An inserted value must have the same data type as specified for its corresponding column.

Now, insert rows into the new Staff1 table.

1. Using the editing window or the command line, enter:

```
INSERT INTO Staff1
    VALUES ("000001", "Zambini", "Rick", {02/15/80},
    "LOS ANGELES","000000", 6000, 5.0);
```

The first row is added to the Staff1 table and the message **1 row(s) inserted** appears.

In the command, insert values are typed in the same order as their corresponding columns are defined in the CREATE TABLE command, separated by commas. Character constants are enclosed by quotes (") and dates by curly braces ({}).

If you are not specifying an insert value for every column in the table, you need to include a column list. A column list contains only the names of the columns in which you are inserting values.

For example, if you did not have complete information for salesperson Zambini, you could enter:

```
INSERT INTO Staff1
  (Firstname, Lastname, Supervisor,
  Hiredate, Location)
  VALUES ("Rick", "Zambini",
  "000000", [02/15/80], "LOS ANGELES");
```

Notice that when you include a column list, values need not be specified in the same order as columns were defined in CREATE TABLE. However, each value must be in the same order as its corresponding column name in the column list.

To insert data for remaining rows into Staff1, you can SELECT rows from an existing table.

2. Enter:

```
INSERT INTO Staff1
  SELECT *
  FROM Staff
  WHERE Lastname <> "Zambini";
```

dBASE IV inserts rows into Staff1 and displays the message **11 row(s) inserted**.

In the SELECT command, the asterisk (*) specifies that all columns in the Staff table be selected for insertion. The <> (does not equal) operator in the WHERE clause narrows the selection to rows whose Lastname values do not include *Zambini*.

3. To display the rows that you have inserted in Staff1, enter:

```
SELECT *
  FROM Staff1;
```

NOTE
Unlike tables created with the CREATE TABLE command, the result table returned by this command is not a permanent table that can be accessed using dBASE or SQL commands.

TIP
Enter the SET PAUSE ON command so that a SELECT result will display one screen at a time.

Updating Data

The SQL UPDATE command allows you to change the values in one or more rows of a table. To increase the commission of all salespeople hired before July 5, 1982 by 25 percent, enter:

```
UPDATE Staff1
  SET Commission = (Commission * 1.25)
  WHERE Hiredate < {07/05/82};
```

The message **6 row(s) updated** appears.

In this command, the WHERE clause specifies the UPDATE condition using the < (less than) operator.

You can use WHERE to specify a simple condition such as this or a more complex qualifier, such as a SELECT command. Chapters 28 and 29 describe WHERE qualifiers used in SELECT queries. These same qualifiers also can be used in UPDATE and other commands that use a WHERE clause.

Deleting Data

The SQL DELETE command allows you to delete table rows. To delete the row for salesperson Long, who has left the company, enter:

```
DELETE FROM Staff1
  WHERE Lastname = "Long";
```

The message **1 row(s) deleted** appears.

The row in the Staff1 table whose Lastname column contained the string "Long" is deleted.

WARNING
If you don't specify a WHERE clause in a DELETE command, all rows are deleted from your table. To prevent you from doing this unintentionally, a warning box is displayed when you enter DELETE without a WHERE clause. Pressing ↵ cancels command entry.

SET SAFETY must be ON for warning messages to be displayed. Refer to Chapter 3 of *Language Reference.*

Modifying Tables

As your SQL database grows, you will need to restructure tables. You can add columns to an existing table, but you cannot remove columns or modify column definitions.

Adding New Columns

Use the ALTER TABLE command to add one or more new columns to a table. To add a new column to the Staff1 table for salesperson telephone number, enter:

```
ALTER TABLE Staff1
  ADD (Phone CHAR(13));
```

The message **Column PHONE added to table STAFF1** appears.

The new column is added following Commission, the last column that you defined for the table using CREATE TABLE.

Restructuring Tables

To delete a column or modify the structure of a table, create a new table with the desired structure and insert rows from the old table into the new table.

1. To restructure Staff1, enter:

```
CREATE TABLE Staff2
  (Staff_no      CHAR(6),
  Fullname       CHAR(25),
  Hiredate       DATE,
  Salary         DECIMAL(6,0),
  Commission     DECIMAL(4,1));
```

   The message **Table STAFF2** created appears.

   In Staff2, the Lastname and Firstname columns are combined in a new column, Fullname, and the Location, Supervisor, and Phone columns are omitted. The new column has the same data type and the combined width of the Lastname and Firstname columns.

TIP

Use the history buffer and the editing window to redisplay the CREATE TABLE command for Staff1, and use the editing keys to modify the command as desired.

2. To insert data from Staff1 into Staff2, enter:

```
INSERT INTO Staff2
  SELECT Staff_no, TRIM(Firstname) + Lastname,
  Hiredate, Salary, Commission
  FROM Staff1;
```

   The message **11 Row(s) inserted** appears.

   You don't need to specify a column list in the INSERT command because the order of columns in the SELECT clause matches the structure of the new table. Data from Firstname and Lastname columns in Staff1 is transferred to the Fullname column of Staff2.

NOTE
Any column in a new table for which no data is inserted is initialized according to its data type:

- *Character — a null (empty) character string*
- *Numeric — zero*
- *Date — a blank date string (/ /)*
- *Logical — the value .F.*

Dropping Tables

To delete a table that is no longer needed, use the DROP TABLE command. When you drop a table, the data in it is lost and cannot be restored. All indexes, views, and synonyms defined for the table are also dropped.

To delete the Staff2 table, enter:

```
DROP TABLE Staff2;
```

The message **Table STAFF2** dropped appears.

NOTE
If the table is listed in an open dBASE IV catalog (.cat) file and SET CATALOG is ON, entering DROP TABLE removes the table from the catalog file.

SQL Views

A view (also called a virtual table because it does not actually contain stored data) combines data that is selected from one or more base tables. If a view is based on another view, it selects data from the base tables of the other view. As data in the base tables changes, so does the information displayed using the view.

You use the SELECT command to display view information. If the view is updatable (refer to CREATE VIEW in *Language Reference),* you can INSERT, UPDATE, or DELETE information from the base table using the view.

The simplest type of view selects rows and columns from a single base table. The first view illustrated in Figure 27-6 uses information from only certain columns of the base Staff table.

The second view in Figure 27-6 combines information from several tables: Sales, Staff, and Customer. View information is derived by matching information in the Staff_no columns of Sales and Staff and in the Cust_no columns of Sales and Customer.

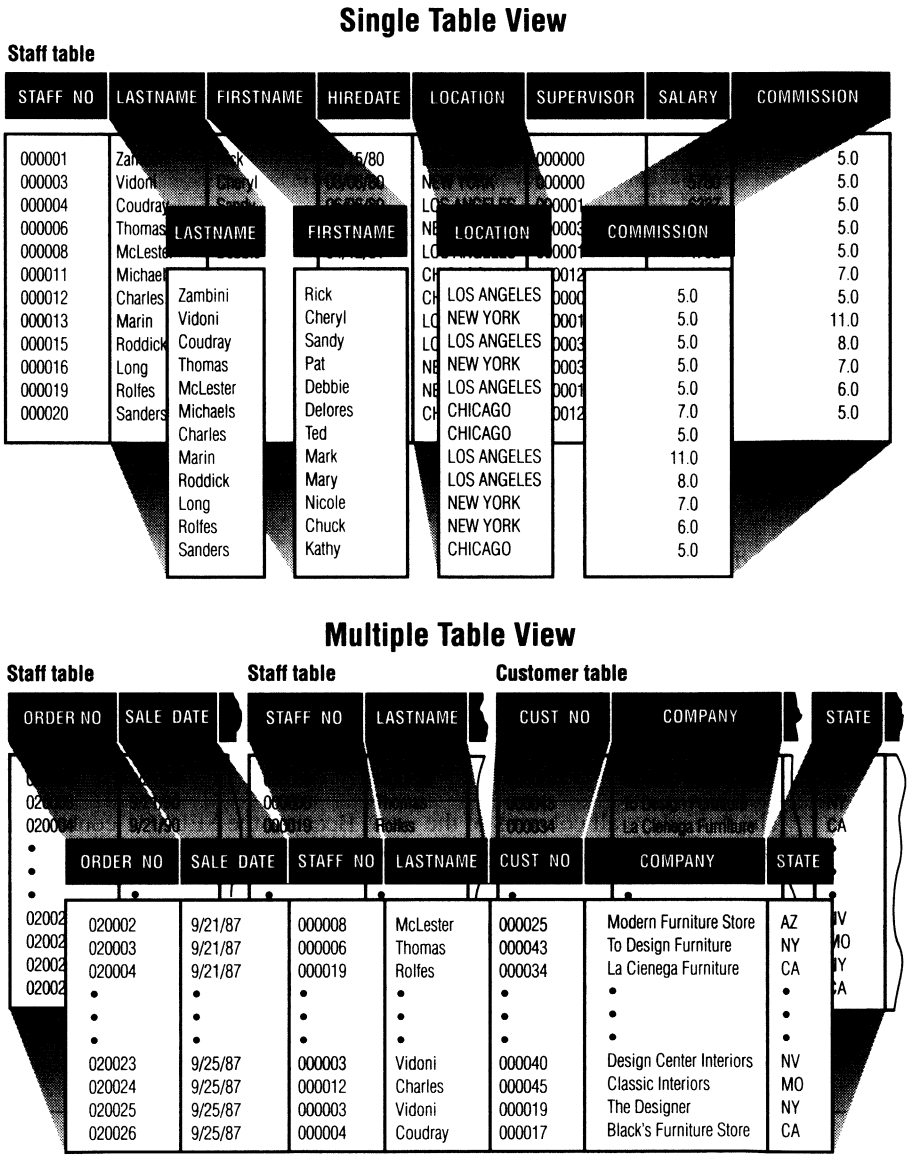


Figure 27-6 SQL views

You can use a view to let someone access just the rows and columns that the person needs from base tables. You can omit columns containing irrelevant or sensitive information, or rows that the person doesn't need to see.

Creating a View

You create a view using the CREATE VIEW command.

To create a view of the Staff1 table, enter:

```
CREATE VIEW La (Salesno, Name, Hired)
  AS SELECT Staff_no, Lastname, Hiredate
  FROM Staff1
  WHERE Location="LOS ANGELES";
```

The message **View LA created** appears.

The view includes only Staff_no, Lastname, and Hiredate data for salespeople based in Los Angeles. The clauses of the command are as follows:

- CREATE VIEW — names the view and optionally specifies column names to be used in place of those of the base table.
- AS — introduces the SELECT command that selects the data from the base tables that is to be included in the view.

To display information using the view, enter:

```
SELECT * FROM LA;

SALESNO    NAME       HIRED
000001     Zambini    02/15/80
000004     Coudray    06/06/80
000008     McLester   04/12/81
000013     Marin      06/05/83
000019     Rolfes     09/09/84
```

The * displays all columns of the view. Because the SELECT contains no WHERE clause, all rows specified by the view appear.

Using Views to Restructure a Table

You can use views to restructure information in a table without creating new tables with different column definitions.

To create a view with a column that combines Lastname and Firstname information from the Staff table, enter:

```
CREATE VIEW Staff2
  (Staff_no, Fullname, Hiredate, Salary, Commission)
  AS SELECT Staff_no, Firstname+Lastname, Hiredate, Salary, Commission
  FROM Staff;
```

The Staff2 view accomplishes the same purpose as the Staff2 table that you created earlier. However, using a view has the following advantages:

- Existing data is not duplicated.
- The same base table can be used for a number of different views.
- As you change data in a base table, view data is automatically updated.

Constructing a View for More than One Table

You can create a view that combines rows and columns from more than one table.

To define a view of data from Sales and Staff tables, enter:

```
CREATE VIEW Orders
  (Order_no, Sale_date, Seller)
  AS SELECT Order_no, Sale_date, Lastname
  FROM Sales, Staff
  WHERE Sales.Staff_no = Staff.Staff_no;
```

The message **View ORDERS created** appears.

This view links salespeople with their orders. It combines the salesperson's last name from the Staff table with the order number and sale date from the Sales table. The two tables are related using their common Staff_no column.

NOTE
The SELECT command used in this CREATE VIEW command creates a join of the Sales and Staff tables. Joins are described in Chapter 29.

To retrieve all information from the Orders view, enter:

```
SELECT *
  FROM Orders;

ORDER_NO  SALE_DATE    SELLER
020002    09/21/87     McLester
020003    09/21/87     Thomas
020004    09/21/87     Rolfes
020005    09/21/87     Zambini
  .          .            .
  .          .            .
  .          .            .
020024    09/25/87     Charles
020025    09/25/87     Vidoni
020026    09/25/87     Coudray
```

Dropping Views

A view is automatically dropped when its base tables are dropped. You also can drop a view using the DROP VIEW command.

To drop the Orders view, enter:

```
DROP VIEW Orders;
```

The message **View ORDERS dropped** appears.

Defining Table and View Synonyms

A synonym is an alternate name for a table or view that can be substituted for a table or view name in any SQL command. For example, you can use a synonym to shorten or simplify command entries.

1. To create a synonym for the Customer table, enter:

```
CREATE SYNONYM C1 FOR Customer;
```

   The message **Synonym C1 created for CUSTOMER** appears.

2. To drop synonym C1, enter:

```
DROP SYNONYM C1;
```

   The message **Synonym C1 dropped** appears.

SQL Indexes

You use indexes on SQL tables for the same reason that you use indexes on dBASE IV database files: to access data quickly. You create an index on the column or columns of a table that you specify most often in commands.

You can define an index on up to 10 columns. If you attempt to include more than 10 columns in an index, you receive an error message.

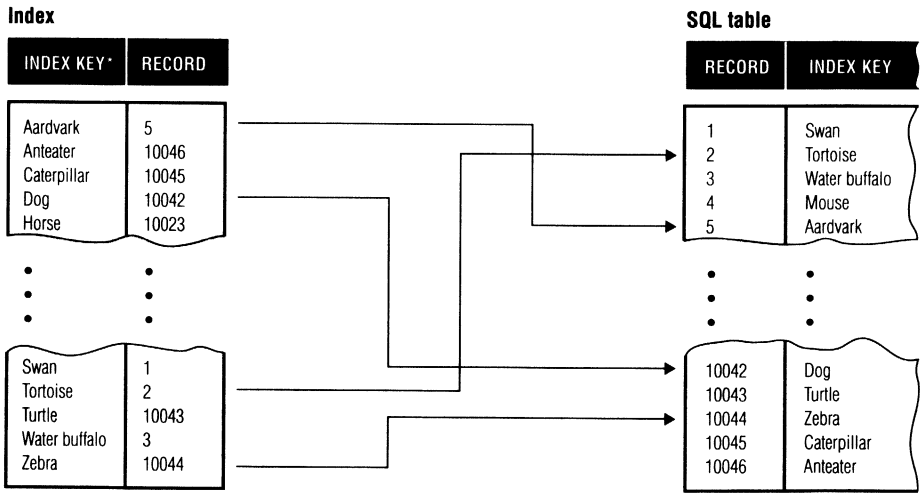
Unlike using indexes on dBASE database files, you don't prescribe the use of a specific index for a SQL operation. SQL automatically chooses the most efficient index that is available for an operation.

Indexes are defined for tables, not for views. However, when you use a view, SQL uses the indexes that you have defined on the view's base tables.

You can create up to 47 different indexes for a table. Each index is defined as a tag in a dBASE multiple index (.mdx) file that has the same name as the table. The name of a SQL index tag can be up to 10 characters long, must begin with a letter, and may contain letters, numbers, and underscores.

You can think of an index tag as a table with two columns. The first column contains the *index keys*. These are the row values for the columns that constitute the index, arranged in the order that you have specified. The second index column contains the record number of the corresponding row.

Figure 27-7 describes the operation of an index.



* The index key column contains the row values of columns in the SQL table for which the index is created. Index key values are arranged in ascending or descending order, depending on whether you specify ASCending (the default) or DEScending order when creating the index.

Figure 27-7 Indexing a table

The maximum combined length of the columns included in an index is 100 bytes. You can use any type of column in an index key except logical.

Although indexes speed up data manipulation, they hamper the performance of table updates because the index keys of each index tag also have to be updated. Also, each index tag takes up disk space.

Therefore, limit indexes to those that satisfy your most important data manipulation needs. Drop any index that has outlived its usefulness. Indexes are automatically dropped when you drop the table on which they are defined.

Creating Indexes

You create indexes using the CREATE INDEX command.

To create an index on the Lastname column of the Customer table, enter:

```
CREATE INDEX Lastname
  ON Customer (Lastname);
```

The message **Index LASTNAME created** appears.

By default, this command defines an ascending (alphabetical) order index called Lastname. To define a descending order index, you could have specified *(Lastname DESC).*

NOTE
If you use more than one column for an index key, you can't specify both ASC and DESC for the index.

Figure 27-8 illustrates how the index works.

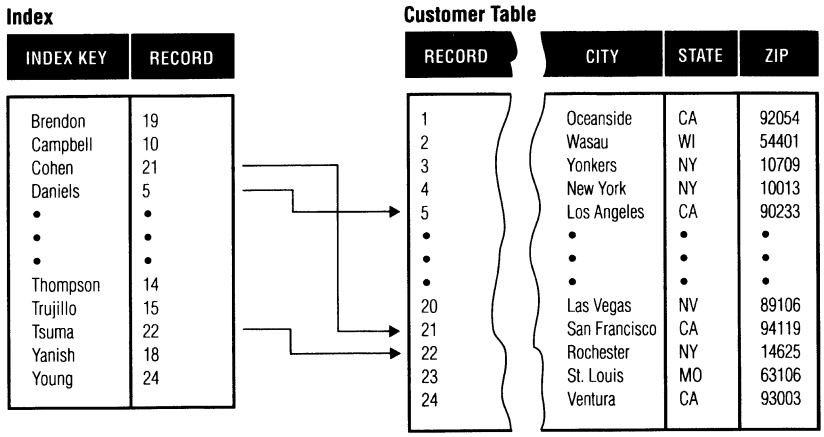


Figure 27-8 Indexing the Customer table

You use the UNIQUE keyword to specify an index in which each key value must be unique.

To create a unique index on the Sales table, enter:

```
CREATE UNIQUE INDEX Order_no
  ON Sales (Order_no);
```

The message **Index ORDER_NO created** appears.

Now, when you insert a new row in the Sales table, the row is accepted only if its value for Order_no is unique. This ensures that you do not inadvertently insert a duplicate row.

Likewise, if you mistakenly change an Order_no value for one row to that of another row, an error box appears. Also, using Order_no in the WHERE clause of a SELECT command is a very efficient way of retrieving Sales records.

Attempting to create a unique index on a column that already contains duplicate index values also causes an error.

A unique index affects performance for inserts and updates because the index key value of each inserted or updated row must be checked against existing index key values. Also, the database file for a table that has a unique index is available in dBASE mode only as a read-only file.

Therefore, use a unique index only when absolutely necessary.

Dropping Indexes

Drop an index using the DROP INDEX command.

To remove the Order_No index, enter:

```
DROP INDEX Order_no;
```

The message **Index ORDER_NO dropped** appears.

SQL System Catalogs

SQL maintains and references information about each table and view in its *system catalog tables.*

Catalog tables help ensure the reliability and security of SQL operations. Catalog table information includes characteristics of tables, views, columns, and indexes; dependencies between base tables and views; synonyms for table and view names; and user privileges granted on tables and views (if PROTECT is installed).

NOTE
A catalog table is different from a dBASE IV catalog file, *which contains name and directory information for the files used for an application.*

When you create a SQL database using the CREATE DATABASE command, a set of 10 catalog tables is automatically created for the new database. Refer to Chapter 7 of *Language Reference* for a description of each catalog table.

Displaying Catalog Information

You display catalog table information just as you do information from any other SQL table. For example, display the names of tables and views in the current database, stored in the Systabls catalog table.

1. Enter:

```
SELECT Tbname, Tbtype
  FROM Systabls;

TBNAME      TBTYPE
SYSTABLS    T
SYSCOLS     T
SYSIDXS     T
   .        .
   .        .
   .        .
ASSEMBLY    T
CUSTOMER    T
INVENTRY    T
ITEMS       T
SALES       T
STAFF       T
STAFF1      T
LA          V
STAFF2      V
```

The Tbtype column indicates whether files are base tables (T), views (V), or temporary tables (D or K). Note that the database catalog tables also are included in the listing.

Display the column definitions for the Customer table by querying the Syscols table.

2. Enter:

```
SELECT Colname, Coltype, Collen
  FROM Syscols
  WHERE Tbname="SALES";

COLNAME     COLTYPE  COLLEN
ORDER_NO    C              6
SALE_DATE   D              8
STAFF_NO    C              6
CUST_NO     C              6
INVOICED    L              1
```

The result lists the names, data types (C=character, D=date, L=logical, N=numeric), and lengths of columns in the Sales table.

Although you can view the contents of the catalog tables, you cannot update them unless you've installed PROTECT and are logged in with the SQLDBA user ID.

Master Catalog Table

A master catalog table, Sysdbs, is located in the SQL *home* directory, the directory in which the SQL system files are installed. Sysdbs contains the name of each database, the user ID (log-in name) of the person who created it (if PROTECT is installed), the date it was created, and the full operating system path location of the database directory.

Quitting dBASE IV

To exit from dBASE IV and return to the operating system, enter:

```
SQL. QUIT
```

Summary

This chapter showed you how to enter SQL commands interactively to create databases containing SQL objects. The following section summarizes the points covered in this chapter.

To display the SQL prompt:
At the dot prompt, enter the SET SQL ON command.

To enter a SQL command on the command line:
Type the command on the command line and terminate it with a semicolon. Use the editing keys to correct mistakes. Press ↵ to enter the command.

To enter a SQL command in the editing window:
Press **Ctrl-Home** to open the window. Type the command on a number of lines, pressing ↵ to end one line and start the next. Terminate the command with a semicolon. Use the navigation and editing keys to correct mistakes. Press **Ctrl-End** to enter the command.

To use the facilities of dBASE IV:
Use the ↑ key to display and re-enter SQL commands stored in the history buffer. Press **F2 Assist** to display the Control Center and use its panels to browse, edit, and work with SQL tables as database files. Press **F1 Help** or type help at the SQL prompt to use the dBASE IV Help system for SQL command entry.

To create a SQL database:
Enter the CREATE DATABASE <database name> command. A database is used to contain SQL objects: tables, views, synonyms (alternate names for tables and views), indexes, and catalogs (system-controlled tables that contain information about a database and its contents).

To list available databases:
Enter the SHOW DATABASE command.

To activate an existing database:
Enter the START DATABASE <database name> command.

To close the current database:
Enter the STOP DATABASE command.

To remove an existing database:
Enter the DROP DATABASE <database name> command.

To create a table:
Enter the CREATE TABLE command to define the structure of a table: the name of each column, its data type, and (if necessary) its width.

To insert rows in a table:
Enter the INSERT command to specify column values for a new row.

To change table data:
Enter the UPDATE command to change the column values of existing rows.

To remove table rows:
Enter the DELETE command to delete one or more rows that match a *WHERE clause* search condition.

To add columns to a table:
Enter the ALTER TABLE command to define one or more new columns to the right of existing columns.

To restructure a table:
Enter the CREATE TABLE command to define the new structure of the old table. Enter the INSERT command with a *SELECT clause* to copy data from the old table into the restructured table. You can use the CREATE VIEW command to define a view that shows a restructured version of the old table without physically restructuring the table.

To remove an unneeded table:
Enter the DROP TABLE <table name> command.

To create a view:
Enter the CREATE VIEW command. The command's SELECT clause defines the base table information that the view will include; as an option, different column headings can be defined for the view.

To remove a view:
Enter the DROP VIEW <view name> command.

To create and remove synonyms:
A synonym is an alternate, normally shorter name for a table or view that can be substituted for the actual name in any SQL command. To create a synonym, enter the CREATE SYNONYM <name> FOR <table or view name> command. To remove a synonym, enter the DROP SYNONYM <name> command.

To create an index:

A SQL index speeds access to table data, and is similar to a dBASE IV index tag. However, in SQL you do not prescribe use of an index for an operation; the SQL system itself determines the most efficient index.

To create an index, enter the CREATE [UNIQUE] INDEX <index name> ON <table name> (<column name>) command. The optional UNIQUE keyword is used to control the uniqueness of each index value to prevent insertion of duplicate rows or the erroneous update of an existing key value.

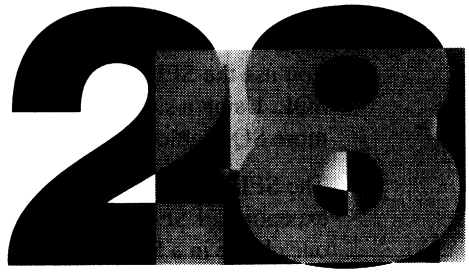
To remove an index:

Enter the DROP INDEX <index name> command.

To display catalog information:

To display information maintained in the SQL system catalog tables, enter the SELECT command. Information about SQL catalogs and the information that they contain is found in Chapter 7 of *Language Reference.* Note that a SQL catalog table differs from a dBASE catalog (.cat) file, which is a listing of information for files used for a particular application.

SQL Queries



This chapter shows you how to use the SELECT command to retrieve table data. You will learn how to:

- Retrieve data for specified table columns or for all columns
- Retrieve information without duplication
- Retrieve information only for rows that satisfy one or more search conditions
- Use expressions in a SELECT command
- Use dBASE functions and aggregate functions in command expressions
- Order a command result
- Group rows by category and limit the groups that appear in a result
- Combine SELECT queries

For complete information about any SQL command used in this manual, refer to Chapter 6 of *Language Reference.*

Preparing for This Chapter

Start dBASE IV:

1. Enter dbase at the operating system prompt.
2. If the Control Center is displayed, choose **Exit to dot prompt** from the **Exit** menu to access the dot prompt.
3. Enter SQL mode by entering:

```
. SET SQL ON
```

4. Start the Samples database:

```
SQL. START DATABASE Samples;
```

SELECT Overview

You use the SELECT command to retrieve data in both interactive and embedded SQL. Using a single SELECT command, you can retrieve any set of data from one or more SQL tables.

The SELECT command syntax, shown below, includes a number of clauses. With the exception of SELECT and FROM, most clauses are optional. When you use more than one clause in a SELECT command, combine them in the order shown in the syntax.

```
SELECT <clause>
[INTO <clause>]
FROM <clause>
[WHERE <clause>]
[GROUP BY <clause>]
[HAVING <clause>]
[UNION subquery]...
[ORDER BY <clause>/FOR UPDATE OF <clause>]
[SAVE TO TEMP <clause>];
```

NOTE
The INTO and FOR UPDATE OF clauses are used only for queries in embedded SQL applications. Refer to Chapter 31.

Simple Queries

The simplest form of the SELECT command includes only SELECT and FROM clauses.

To display the Company, Firstname, and Lastname columns from the Customer table, enter:

```
SELECT Company, Firstname, Lastname
  FROM Customer;
```

The following result is displayed:

```
COMPANY                   FIRSTNAME       LASTNAME
Leonard Design Services   Rick            Leonard
Ace Furniture             Lisa            Martin
Custom Furniture          Daniel          Pollock
The Office                Dominique       LeClerc
      .                      .               .
      .                      .               .
      .                      .               .
Classic Interiors         Eric            Lawson
Commercial Interiors LTD  Sandy           Young
```

The SELECT clause specifies the column names that are to appear in the result, and their order. The FROM clause identifies the table that contains the columns.

The result table is not stored, like a base table, nor can you reference it after it is displayed. However, you can:

- Temporarily save the result table for use during the current session using the SAVE TO TEMP clause.
- Permanently save the result table as a database file using SAVE TO TEMP with the KEEP option.

SAVE TO TEMP and KEEP are described in Chapter 30.

Selecting All Columns

Use the asterisk (*) symbol to select all columns in a table.

To display the information in all columns of the Staff table, enter:

```
SELECT *
  FROM Staff;

STAFF_NO  LASTNAME  FIRSTNAME HIREDATE  LOCATION     SUPERVISOR SALARY COMMISSION
000001    Zambini   Rick      02/15/80  LOS ANGELES  000000       6000       5.0
000003    Vidoni    Cheryl    03/06/80  NEW YORK     000000       5780       5.0
000004    Coudray   Sandy     06/06/80  LOS ANGELES  000001       6237       5.0
000006    Thomas    Pat       01/08/81  NEW YORK     000003       5875       5.0
000008    McLester  Debbie    04/12/81  LOS ANGELES  000001       4792       5.0
000011    Michaels  Delores   05/05/82  CHICAGO      000012       4927       7.0
000012    Charles   Ted       02/02/83  CHICAGO      000000       5945       5.0
000013    Marin     Mark      06/05/83  LOS ANGELES  000001       4802      11.0
000015    Roddick   Mary      02/13/84  NEW YORK     000003       5493       8.0
000016    Long      Nicole    08/18/84  NEW YORK     000003       5190       7.0
000019    Rolfes    Chuck     09/09/84  LOS ANGELES  000001       4586       6.0
000020    Sanders   Kathy     03/23/85  CHICAGO      000012       3783       5.0
```

SELECT with DISTINCT

Because any inventory item can be stocked in three different locations — Chicago, Los Angeles, and New York — the result of SELECT * FROM Inventry might contain as many as three rows for the same item. To display each part number just once, use the *DISTINCT* keyword.

NOTE
The following rules apply to using the DISTINCT keyword in a SELECT command. This applies to the outer query and inner queries. (Refer to the Subqueries section in Chapter 29.)

- *DISTINCT can be used only once in any SELECT clause of a SELECT command.*
- *DISTINCT can apply only to the first column named in a SELECT clause.*
- *DISTINCT can be used in a HAVING clause to operate on the values of any column in the table (for example: SELECT Location, MAX(Salary) FROM Staff GROUP BY Location HAVING MAX(DISTINCT Salary) > 50000;).*

Enter:

```
SELECT DISTINCT Part_no, Descript
  FROM Inventry;

PART_NO    DESCRIPT
001001     WORKSTATION-ELECTRONIC OFFICE
001002     HOME OFFICE SUITE
001005     EXECUTIVE SUITE ENSEMBLE
001007     WOOD DESK-SINGLE PEDESTAL
  .
  .
  .
001033     CHAIR-TRADITIONAL ARM
001038     LAMP-DRAFTING SWING ARM
```

Figure 28-1 illustrates how DISTINCT works.

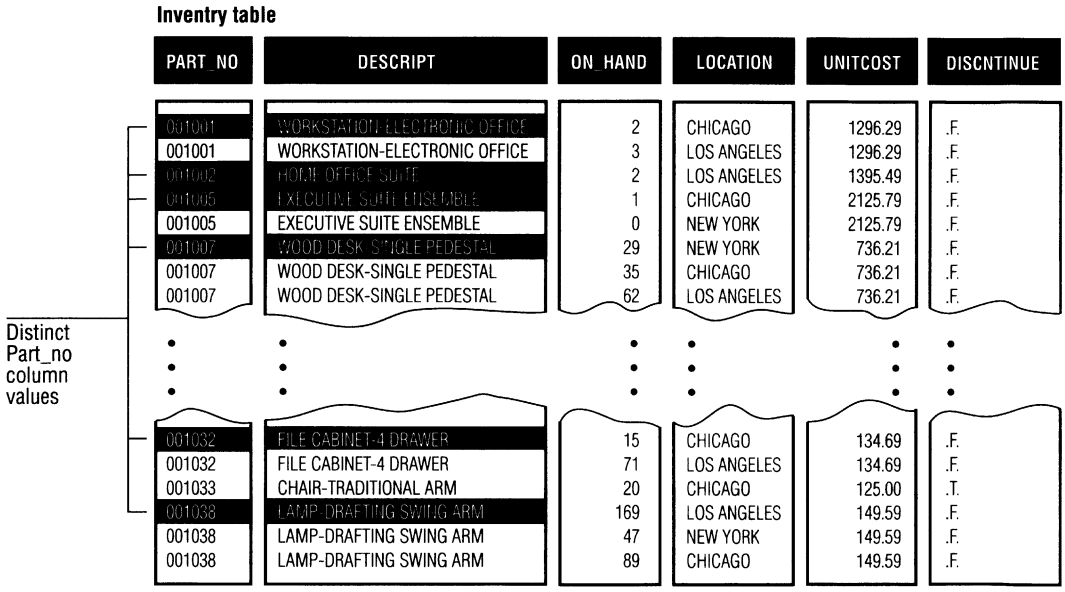


Figure 28-1 Selection of DISTINCT rows

WHERE Clause

You query specific rows by including a *search condition* in a WHERE clause.

To list all the inventory items stocked in Chicago, enter:

```
SELECT Part_no, Descript, Location, On_hand
  FROM Inventry
  WHERE Location = "CHICAGO";
```

```
PART_NO    DESCRIPT                        LOCATION    ON_HAND
001001     WORKSTATION-ELECTRONIC OFFICE   CHICAGO           2
001005     EXECUTIVE SUITE ENSEMBLE        CHICAGO           1
001009     CHAIR-ADJUSTABLE SWIVEL         CHICAGO         124
001024     LAMP-BRASS TABLE                CHICAGO         140
001032     FILE CABINET-4 DRAWER           CHICAGO          15
001033     CHAIR-TRADITIONAL ARM           CHICAGO          20
001007     WOOD DESK-SINGLE PEDESTAL       CHICAGO          35
001013     CHAIR-MODERN PNEUMATIC          CHICAGO         115
001038     LAMP-DRAFTING SWING ARM         CHICAGO          89
001027     DESK-EXECUTIVE-6 FOOT           CHICAGO          20
001031     CHAIR-EXECUTIVE SWIVEL/TILT     CHICAGO          44
```

The WHERE search condition (Location = "CHICAGO") compares the Location value in each row with the value "CHICAGO". If the search condition is true for the row, the row is included in the result table.

Figure 28-2 shows the selection process.

Inventry table

PART NO	DESCRIPT	ON HAND	LOCATION	UNITCOST	DISCNTINUE
				1296.29	.F.
001002	HOME OFFICE SUITE	2	LOS ANGELES	1395.49	.F.
				2125.79	.F.
001007	WOOD DESK-SINGLE PEDESTAL	29	NEW YORK	736.21	.F.
001008	WORKSTATION-STAND	22	LOS ANGELES	275.66	.F.
				245.38	.T.
001015	CREDENZA-OAK SLIDING DOOR	15	NEW YORK	745.00	.T.
001019	TABLE-BOARD ROOM	12	NEW YORK	4250.00	.F.
001021	MANAGERS OFFICE ENSEMBLE	3	NEW YORK	2380.79	.F.
001022	TABLE-WALNUT OCCASIONAL	5	NEW YORK	414.95	.F.
				230.79	.F.
001025	DESK-EXECUTIVE-5 FOOT	63	LOS ANGELES	985.00	.F.
001029	FILE CABINET-2 DRAWER	200	NEW YORK	89.95	.T.
001031	CHAIR-EXECUTIVE SWIVEL/TILT	79	LOS ANGELES	420.00	.F.
				134.69	.F.
				125.00	.T.
001038	LAMP-DRAFTING SWING ARM	169	LOS ANGELES	149.59	.F.
•	•	•	•	•	•
•	•	•	•	•	•
•	•	•	•	•	•
001005	EXCUTIVE SUITE ENSEMBLE	0	NEW YORK	2125.79	.F.
001029	FILE CABINET-2 DRAWER	145	LOS ANGELES	89.95	.T.

Figure 28-2 Selection of rows

The result table needn't include the column used to specify the WHERE condition. Enter:

```
SELECT  Part_no,  Descript,  On_hand
  FROM Inventry
  WHERE Location = "CHICAGO";
```

Notice that the result includes the same rows, but omits the Location column.

Using Comparison Operators

You can use the comparison operators listed in Table 28-1 in WHERE search conditions.

Table 28-1 Comparison operators

Operator	Description
=	Equals
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
<>	Not equal (# is also supported for dBASE compatibility)
!	Negation of <, >, and = operators (for example, !< evaluates to "not less than")

Values that are compared must be of compatible data types. A character value (or constant) must be compared with another character value, a numeric value with a numeric value, and so on. The SMALLINT, INTEGER, DECIMAL, NUMERIC, and FLOAT data types are compatible with one another.

Character values must be enclosed in single or double quotes, or in brackets ([]). Also, character values are compared using their ASCII representation. Thus, lowercase *a* is not the same as uppercase *A*, and so on.

NOTE

- *A logical type column can be specified in a WHERE clause with or without a comparison operator. For example, the following queries are equivalent:*

```
SELECT * FROM Orders WHERE SHIPPED = .T.;
SELECT * FROM Orders WHERE Shipped;
```

- *You can use dBASE functions to convert search condition values for comparison. For example, use the UPPER( ) function to convert character column values to uppercase and the CTOD( ) function to convert a date string for comparison with date column values.*

Combining Conditions

You can combine search conditions in a WHERE clause using SQL logical operators. Enter:

```
SELECT Part_no, Descript, On_hand, Location
  FROM Inventry
  WHERE Location = "LOS ANGELES" AND On_hand < 20;

PART_NO  DESCRIPT                          ON_HAND  LOCATION
001002   HOME OFFICE SUITE                       2  LOS ANGELES
001001   WORKSTATION-ELECTRONIC OFFICE           3  LOS ANGELES
```

The result lists inventory items stocked in Los Angeles with fewer than 20 on hand. The AND logical operator specifies that the rows displayed satisfy both WHERE conditions.

SQL logical operators are listed in Table 28-2, in the order in which they are evaluated in a WHERE clause.

Table 28-2 SQL logical operators

Operator	Description
NOT	Selects rows that do not meet the search condition
AND	Both conditions must be true to qualify the row
OR	One or both of the conditions must be true

When you specify more than one WHERE condition, each condition is evaluated as true or false for every row. The logical operator then combines the rows for which search conditions evaluate to True to produce a result.

The precedence of logical operators determines the order in which WHERE conditions are evaluated. If you use the same logical operator to combine conditions, the conditions are evaluated from left to right.

You can use parentheses to determine the order in which search conditions are evaluated. Suppose you change the previous command to list inventory items in Los Angeles or New York, excluding items in short supply that cost less than $1,000. Enter:

```
SELECT Part_no, Descript, Location, On_hand, Unitcost
  FROM Inventry
  WHERE Location = "LOS ANGELES" OR Location = "NEW YORK"
  AND On_hand < 20 AND Unitcost < 1000;

PART_NO  DESCRIPT                      LOCATION        ON_HAND   UNITCOST
001002   HOME OFFICE SUITE             LOS ANGELES           2    1395.49
001008   WORKSTATION-STAND             LOS ANGELES          22     275.66
001015   CREDENZA-OAK SLIDING DOOR     NEW YORK             15     745.00
001022   TABLE-WALNUT OCCASIONAL       NEW YORK              5     414.95
   .            .                         .                .        .
   .            .                         .                .        .
   .            .                         .                .        .
001032   FILE CABINET-4 DRAWER         LOS ANGELES          71     134.69
001001   WORKSTATION-ELECTRONIC OFFICE LOS ANGELES           3    1296.29
001029   FILE CABINET-2 DRAWER         LOS ANGELES         145      89.95
```

The result is clearly not what you intended. SQL first evaluates *Location = "NEW YORK" AND On_hand < 20.* Then it evaluates *AND Unitcost < 1000.* Finally, it evaluates *Location = "LOS ANGELES".*

To get the correct result, use parentheses. Enter:

```
SELECT Part_no, Descript, Location, On_hand, Unitcost
  FROM Inventry
  WHERE (Location = "LOS ANGELES" OR Location = "NEW YORK")
  AND (On_hand < 20 AND Unitcost < 1000);

PART_NO  DESCRIPT                      LOCATION        ON_HAND   UNITCOST
001015   CREDENZA-OAK SLIDING DOOR     NEW YORK             15     745.00
001022   TABLE-WALNUT OCCASIONAL       NEW YORK              5     414.95
```

In this case, SQL first evaluates the expression *(Location = "LOS ANGELES" OR Location = "NEW YORK").* Then it evaluates the expression *(On_hand < 20 AND Unitcost < 1000).* Finally, it combines the rows that test True for both sets of conditions.

Defining and Using Expressions

You can use expressions in the SELECT, WHERE, and HAVING clauses of a SELECT command. An expression is one or more data values, sometimes joined by operators.

For example, "Los Angeles" is a constant expression and *Unitcost*10* is a variable expression. When evaluated, an expression always returns a single value: a condition (true or false), a numeric value, or a character string.

An expression can contain the following elements:

- Column names
- Arithmetic operators
- Constants (numeric, logical, date, and character)
- Memory variables
- dBASE functions
- The USER() function, which returns the user ID of the current user if PROTECT is installed

Table 28-3 lists the arithmetic operators that you can use with SQL.

Table 28-3 Arithmetic operators

Operator	Description
** and ^	Exponentiation
+ and –	Unary operator (+ is the default; – for negative values)
* and /	Multiplication and division
+ and –	Addition and subtraction

You can use all of these operators with numeric values. However, you can use only the addition and subtraction operators with character strings, for concatenation.

The expressions that you use in SQL are similar to those that you use in dBASE commands. Figure 28-3 shows how elements are used in constructing an expression.

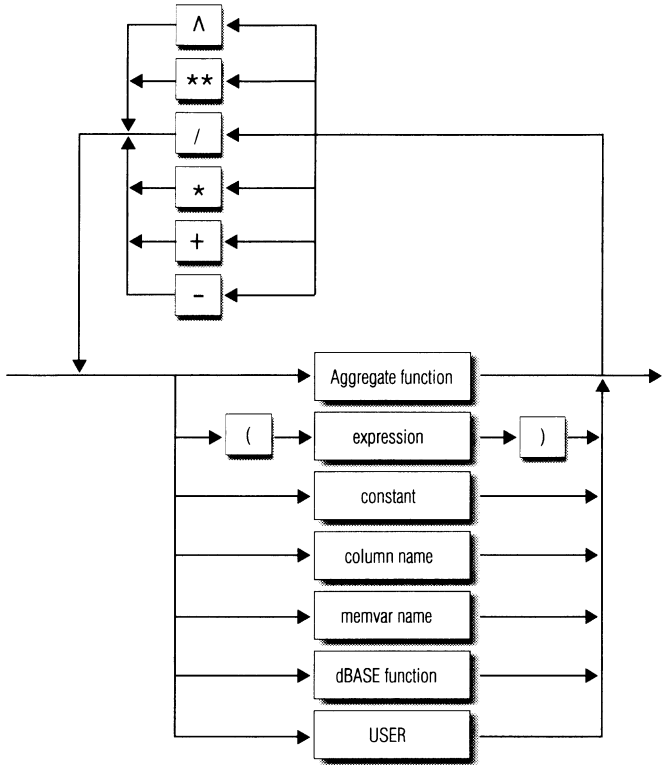


Figure 28-3 Components of SQL expressions

Expressions in the SELECT Clause

You can use expressions in a SELECT clause in the following ways:

- To define a result column that describes information in an adjacent result column
- To define a *calculated result column*

To display yearly salaries calculated from the monthly salaries in the Staff table, enter:

```
SELECT Staff_no, Firstname, Lastname, Salary*12, "(Yearly Salary)"
  FROM Staff;

STAFF_NO  FIRSTNAME        LASTNAME     EXP1      EXP2
000001    Rick             Zambini     72000      (Yearly Salary)
000003    Cheryl           Vidoni      69360      (Yearly Salary)
000004    Sandy            Coudray     74844      (Yearly Salary)
000006    Pat              Thomas      70500      (Yearly Salary)
000008    Debbie           McLester    57504      (Yearly Salary)
000011    Delores          Michaels    59124      (Yearly Salary)
000012    Ted              Charles     71340      (Yearly Salary)
000013    Mark             Marin       57624      (Yearly Salary)
000015    Mary             Roddick     65916      (Yearly Salary)
000016    Nicole           Long        62280      (Yearly Salary)
000019    Chuck            Rolfes      55032      (Yearly Salary)
000020    Kathy            Sanders     45396      (Yearly Salary)
```

EXP1, containing the yearly salary figures, is a calculated column. The yearly salary figures are calculated from the monthly Salary column figures in the Staff table using the expression Salary*12. The expression "(Yearly Salary)" is used to define an informative column (EXP2) describing information in EXP1.

Notice that result columns defined using expressions have EXP headings numbered according to their left-to-right occurrence in the SELECT clause.

Expressions in the WHERE Clause

In a WHERE clause, you use expressions to define search conditions.

1. To display information for salespeople whose annual salary is more than $50,000, enter:

```
SELECT Staff_no, Firstname, Lastname, Salary*12
  FROM Staff
  WHERE Salary*12 > 50000;

STAFF_NO  FIRSTNAME           LASTNAME             EXP1
000001    Rick                Zambini             72000
000003    Cheryl              Vidoni              69360
000004    Sandy               Coudray             74844
000006    Pat                 Thomas              70500
000008    Debbie              McLester            57504
000011    Delores             Michaels            59124
000012    Ted                 Charles             71340
000013    Mark                Marin               57624
000015    Mary                Roddick             65916
000016    Nicole              Long                62280
000019    Chuck               Rolfes              55032
```

The entire WHERE condition, Salary*12 > 50000, is an expression composed of two expressions, Salary*12 and 50000.

You could substitute another expression, in the form of a memory variable, for the expression 50000. This would enable the SELECT command to provide varying results, depending on the value assigned to the memory variable.

2. Enter:

```
SQL. msalary = 50000
    50000
SELECT Staff_no, Firstname, Lastname, Salary*12
  FROM Staff
  WHERE Salary*12 > msalary;
```

The result is the same.

NOTE
Being able to use memory variables in SELECT commands is useful in SQL programs, as discussed in Chapter 31.

dBASE Functions in Expressions

dBASE IV provides more than 100 functions that you can use in SQL expressions.

For example, you can use the DATE() function, which returns your computer's current system date, to display the names of all salespeople with more than five years of service.

1. Enter:

```
SELECT Firstname, Lastname, Hiredate
  FROM Staff
  WHERE (DATE()-Hiredate) > 365*5;

FIRSTNAME      LASTNAME       HIREDATE
Rick           Zambini        02/15/80
Cheryl         Vidoni         03/06/80
Sandy          Coudray        06/06/80
Pat            Thomas         01/08/81
Debbie         McLester       04/12/81
Delores        Michaels       05/05/82
Ted            Charles        02/02/83
Mark           Marin          06/05/83
```

   In this example, the Hiredate value for each salesperson is subtracted from the system date (September 30, 1988, for this example). If the DATE() – Hiredate interval for a row is greater than 365*5, the row is selected for the result.

   The UPPER(), LOWER(), and SUBSTR() functions are useful in defining character string comparisons.

2. To display all customers located in San Francisco, enter:

```
SELECT Company, City, State
  FROM Customer
  WHERE UPPER(City) = "SAN FRANCISCO";

COMPANY                   CITY           STATE
Black's Furniture Store   San Francisco  CA
A1 Office Supply Store    San Francisco  CA
Cohen's Furniture         San Francisco  CA
```

   The UPPER() function transforms all City values to uppercase so that they can be compared with the value "SAN FRANCISCO". This is useful if you aren't certain about whether City data in Customer is entered in uppercase or a combination of uppercase and lowercase.

3. For example, enter this command a different way:

```
SELECT Company, City, State
  FROM Customer
  WHERE City = "SAN FRANCISCO";
```

No result is displayed because City data is actually entered in lowercase with initial capitals. (Remember, ASCII uppercase and lowercase characters aren't equivalent.)

4. Now, enter:

```
SELECT UPPER(Company), City, State
  FROM Customer
  WHERE UPPER(City) = "SAN FRANCISCO";
```

Notice that company names are displayed in uppercase.

NOTE
When using a dBASE function in a SQL command, you cannot abbreviate its name, as you can in a dBASE command.

Using SQL Aggregate Functions in Expressions

You can use the following SQL *aggregate functions* in expressions in a SELECT clause:

- AVG() — Calculates the average of the values in a numeric column
- COUNT() — Counts the number of rows selected
- MAX() — Finds the maximum value in a character, date, or numeric column
- MIN() — Finds the minimum value in a character, date, or numeric column
- SUM() — Sums the values of a numeric column

NOTE
For character columns, MIN() and MAX() return the lowest (A to Z) and highest (a to z) ASCII values, respectively. Values are compared from left to right.

Figure 28-4 illustrates the use of aggregate functions in SELECT clause expressions.

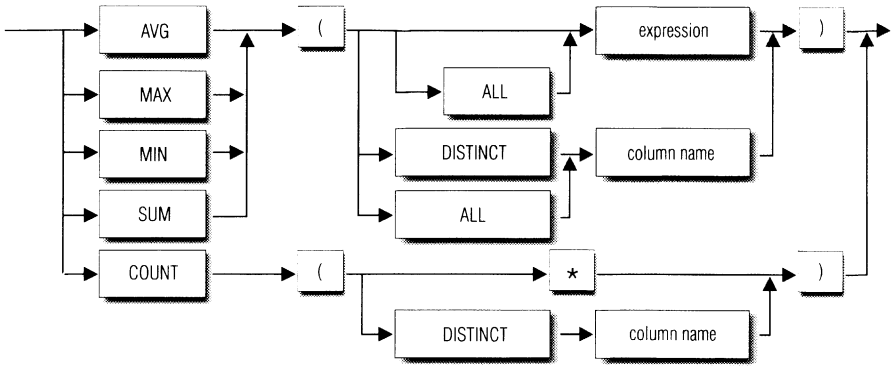


Figure 28-4 SQL aggregate functions

NOTE
You can use aggregate functions only in expressions that address column values. You cannot perform an aggregate operation on a constant value, a memory variable, or the values in a logical column.

Aggregate Expressions in a SELECT Clause

In a SELECT clause, you use aggregate expressions instead of column names. The aggregate functions operate on all selected values to return a single summary value.

Because an aggregate function operates on all selected column values to produce one value, you cannot use an aggregate expression in a SELECT clause that contains a column name expression that displays all selected values.

NOTE
If the SELECT clause contains both column names and aggregate functions, the command must include a GROUP BY clause. Refer to the Grouping Rows section later in this chapter.

1. To count the number of rows in the Staff table, enter:

```
SELECT COUNT(*)
  FROM Staff;

                COUNT1
                    12
```

NOTE

- *The asterisk symbol (*) specifies that COUNT() operate on the values of all columns of the table.*
- *Result table headings for aggregate functions use the function name plus a number indicating its left-to-right position in the SELECT clause.*

2. To use the SUM() function to calculate the total monthly payroll for salespeople, enter:

```
SELECT SUM(Salary)
  FROM Staff;

            SUM1
           63410
```

3. To find the highest, the lowest, and the average salary for salespeople, enter:

```
SELECT MAX(Salary), MIN(Salary), AVG(Salary)
  FROM Staff;

           MAX1                 MIN2                 AVG3
           6237                 3783              5284.17
```

You can use an aggregate function with the DISTINCT keyword to operate only on unique column values.

4. To count the number of individual cities in which there are customers, enter:

```
SELECT COUNT(DISTINCT City)
  FROM Customer;

         COUNT1
             15
```

Aggregate Expressions Qualified by a WHERE Clause

In a SELECT command that contains a WHERE clause, an aggregate function in the SELECT clause applies only to the column values that are qualified by the WHERE condition.

To find the number of customers located in New York, enter:

```
SELECT COUNT(*)
  FROM Customer
  WHERE State = "NY";

          COUNT1
               5
```

The SELECT command retrieves the rows whose State column contains "NY". The COUNT() function then counts the number of rows retrieved. Figure 28-5 illustrates this process.

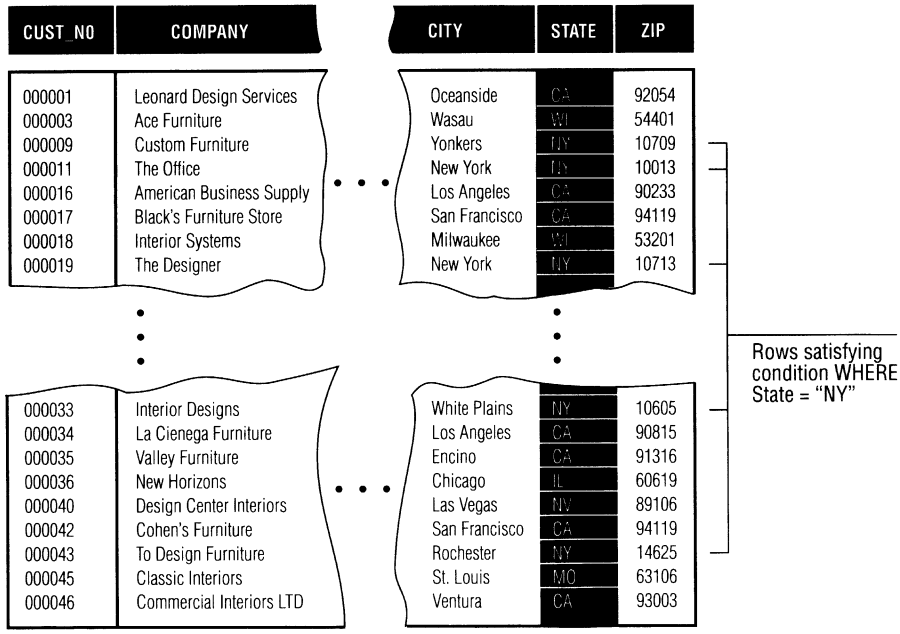


Figure 28-5 Aggregate function with a WHERE clause

SELECT with BETWEEN, IN, and LIKE Predicates

You can use the following *predicates* in a WHERE clause to qualify rows for inclusion in a result:

- BETWEEN — selects column values that fall within a range.
- IN — selects column values that match one of a number of values.
- LIKE — selects column values that contain a character string.

Predicates are so called because the SELECT result is predicated on their operation.

BETWEEN Predicate

The BETWEEN predicate tests for column values that fall within a range bounded by two values.

1. To find salespeople whose monthly salaries are between $5,000 and $6,000, enter:

```
SELECT Lastname, Salary from Staff
  WHERE Salary >= 5000 AND Salary <= 6000;

LASTNAME        SALARY
Zambini           6000
Vidoni            5780
Thomas            5875
Charles           5945
Roddick           5493
Long              5190
```

2. Now, enter:

```
SELECT Lastname, Salary from Staff
  WHERE Salary BETWEEN 5000 AND 6000;
```

   The result is the same. Notice that the range specified using BETWEEN is *inclusive;* rows whose values match either of the range values also are selected for the result.

   You can also use BETWEEN with character and date type columns.

3. To recall the names and locations of salespeople whose last names fall alphabetically between Sanders and Zambini, enter:

```
SELECT Lastname, Location from Staff
  WHERE Lastname BETWEEN "Sanders" AND "Zambini";

LASTNAME        LOCATION
Zambini         LOS ANGELES
Vidoni          NEW YORK
Thomas          NEW YORK
Sanders         CHICAGO
```

IN Predicate

The IN predicate tests for column values that match one of a list of values.

To list information for customers located in Arizona, Missouri, or Wisconsin, enter:

```
SELECT Company, Address, City, State, Zip
  FROM Customer
  WHERE State IN ("AZ","MO","WI");
```

```
COMPANY                  ADDRESS              CITY         STATE   ZIP
Ace Furniture            1960 Lindley Ave.    Wasau        WI      54401
Interior Systems         899 Kenwood St.      Milwaukee    WI      53201
Baker Furniture          6700 Tyler St.       Phoenix      AZ      85012
Modern Furniture Store   366 Shirley Ave.     Phoenix      AZ      85004
Al's Furniture & Supplies 40555 Brentwood     St. Louis    MO      63121
Contemporary Designs     5670 Colorado Blvd.  Milwaukee    WI      53220
Classic Interiors        2015 Edmonton        St. Louis    MO      63106
```

An equivalent WHERE condition would be:

```
WHERE State = "AZ" OR State = "MO" OR State = "WI"
```

You can use IN with character, date, numeric, and logical conditions. An IN value list can include dBASE functions, memory variables, column names, and the USER keyword.

LIKE Predicate

The LIKE predicate selects result rows by matching column values with a character string or a character-type memory variable. You can use the following wildcard characters in a LIKE string:

- Underscore (_) — represents any single character
- Percent symbol (%) — represents no characters or a number of characters

1. To display information for customers located in states whose initial letter is N, enter:

```
SELECT Company, City, State, Zip
  FROM Customer
  WHERE State LIKE "N_";
```

```
COMPANY                  CITY          STATE  ZIP
Custom Furniture         Yonkers       NY     10709
The Office               New York      NY     10013
The Designer             New York      NY     10713
Las Vegas Furniture      Las Vegas     NV     89106
Accent Furniture Designs Las Vegas     NV     89108
Interior Designs         White Plains  NY     10605
Design Center Interiors  Las Vegas     NV     89106
To Design Furniture      Rochester     NY     14625
```

2. To display inventory items whose descriptions contain "DESK", enter:

```
SELECT DISTINCT Part_no, Descript
  FROM Inventry
  WHERE Descript LIKE "%DESK%";

PART_NO   DESCRIPT
001007    WOOD DESK-SINGLE PEDESTAL
001025    DESK-EXECUTIVE-5 FOOT
001027    DESK-EXECUTIVE-6 FOOT
```

Ordering Results

Up to now, SELECT result rows have been listed in *natural* order, the order in which their base table rows were inserted. You can use the ORDER BY clause in a SELECT command to display a result in alphabetic or numeric order by the values in one or more columns.

Ordering on a Single Column

Using ORDER BY, you can list a result in ascending (lowest to highest) or descending (highest to lowest) order.

To display inventory items in ascending order by the first letter of their descriptions, enter:

```
SELECT Part_no, Descript, Location, On_hand
  FROM Inventry
  ORDER BY Descript ASC;

PART_NO     DESCRIPT                       LOCATION          ON_HAND
001009      CHAIR-ADJUSTABLE SWIVEL        CHICAGO               124
001031      CHAIR-EXECUTIVE SWIVEL/TILT    LOS ANGELES            79
001031      CHAIR-EXECUTIVE SWIVEL/TILT    CHICAGO                44
  .               .                            .                 .
  .               .                            .                 .
  .               .                            .                 .
001032      FILE CABINET-4 DRAWER          LOS ANGELES            71
001002      HOME OFFICE SUITE              LOS ANGELES             2
001024      LAMP-BRASS TABLE               CHICAGO               140
  .               .                            .                 .
  .               .                            .                 .
  .               .                            .                 .
001001      WORKSTATION-ELECTRONIC OFFICE  CHICAGO                 2
001001      WORKSTATION-ELECTRONIC OFFICE  LOS ANGELES             3
001008      WORKSTATION-STAND              LOS ANGELES            22
```

The ORDER BY column, Descript, must be included in the SELECT clause. The ASC keyword, which specifies ascending (alphabetical) order, is optional in this command because ascending order is the default for ORDER BY.

TIP

To speed retrieval, create indexes on columns by which you most often order results.

Ordering on More than One Column

Now, order the previous result first by location and then by description.

Enter:

```
SELECT Part_no, Descript, Location, On_hand
  FROM Inventry
  ORDER BY Location, Descript;

PART_NO  DESCRIPT                        LOCATION     ON_HAND
001009   CHAIR-ADJUSTABLE SWIVEL         CHICAGO          124
001031   CHAIR-EXECUTIVE SWIVEL/TILT     CHICAGO           44
001013   CHAIR-MODERN PNEUMATIC          CHICAGO          115
   .            .                          .              .
   .            .                          .              .
   .            .                          .              .
001031   CHAIR-EXECUTIVE SWIVEL/TILT     LOS ANGELES       79
001013   CHAIR-MODERN PNEUMATIC          LOS ANGELES       35
001025   DESK-EXECUTIVE-5 FOOT           LOS ANGELES       63
   .            .                          .              .
   .            .                          .              .
   .            .                          .              .
001001   WORKSTATION-ELECTRONIC OFFICE   LOS ANGELES        3
001008   WORKSTATION-STAND               LOS ANGELES       22
001031   CHAIR-EXECUTIVE SWIVEL/TILT     NEW YORK          76
   .            .                          .              .
   .            .                          .              .
   .            .                          .              .
001019   TABLE-BOARD ROOM                NEW YORK          12
001022   TABLE-WALNUT OCCASIONAL         NEW YORK           5
001007   WOOD DESK-SINGLE PEDESTAL       NEW YORK          29
```

Inventory item rows are ordered first by Location values. Within each location group, rows are then ordered by Descript values. ORDER BY columns are separated by a comma.

To list items in ascending order by location and in descending order by number of items on hand, enter:

```
SELECT Part_no, Descript, Location, On_hand
  FROM Inventry
  ORDER BY 3, 4 DESC;

PART_NO   DESCRIPT                      LOCATION     ON_HAND
001024    LAMP-BRASS TABLE              CHICAGO          140
001009    CHAIR-ADJUSTABLE SWIVEL       CHICAGO          124
001013    CHAIR-MODERN PNEUMATIC        CHICAGO          115
   .             .                         .              .
   .             .                         .              .
   .             .                         .              .
001038    LAMP-DRAFTING SWING ARM       LOS ANGELES      169
001029    FILE CABINET-2 DRAWER         LOS ANGELES      145
001031    CHAIR-EXECUTIVE SWIVEL/TILT   LOS ANGELES       79
   .             .                         .              .
   .             .                         .              .
   .             .                         .              .
001022    TABLE-WALNUT OCCASIONAL       NEW YORK           5
001021    MANAGERS OFFICE ENSEMBLE      NEW YORK           3
001005    EXECUTIVE SUITE ENSEMBLE      NEW YORK           0
```

Notice that the Location and On_hand columns are represented in the ORDER BY clause by the numbers 3 and 4. These numbers represent the left-to-right positions of these columns in the SELECT clause. When you are ordering by a column derived using an expression, such as *Salary*12* or *SUM(Salary)*, you must use a number to represent the expression in ORDER BY.

ORDER BY with a WHERE Clause

ORDER BY orders a result that is qualified by a WHERE condition.

To restrict an Inventry result to rows for items stocked in Chicago, enter:

```
SELECT Part_no, Descript, Location, On_hand
  FROM Inventry
  WHERE Location = "CHICAGO"
  ORDER BY Part_no;

PART_NO    DESCRIPT                        LOCATION        ON_HAND
001001     WORKSTATION-ELECTRONIC OFFICE    CHICAGO              2
001005     EXECUTIVE SUITE ENSEMBLE         CHICAGO              1
001007     WOOD DESK-SINGLE PEDESTAL        CHICAGO             35
001009     CHAIR-ADJUSTABLE SWIVEL          CHICAGO            124
001013     CHAIR-MODERN PNEUMATIC           CHICAGO            115
001024     LAMP-BRASS TABLE                 CHICAGO            140
001027     DESK-EXECUTIVE-6 FOOT            CHICAGO             20
001031     CHAIR-EXECUTIVE SWIVEL/TILT      CHICAGO             44
001032     FILE CABINET-4 DRAWER            CHICAGO             15
001033     CHAIR-TRADITIONAL ARM            CHICAGO             20
001038     LAMP-DRAFTING SWING ARM          CHICAGO             89
```

Grouping Rows

The GROUP BY and HAVING clauses let you arrange rows in groups on which you can perform aggregate operations. Using GROUP BY and HAVING, you can change the scope of an aggregate operation from all values in a column to just the values contained in the group.

GROUP BY Clause

The GROUP BY clause groups a result by common values in one or more columns. Each group is summarized as a single row in the result table. Result table headings either identify groups (G_) or aggregate values for each group.

Each column specified in the SELECT clause must also be included in the GROUP BY clause unless it is used in an aggregate operation. The inverse is also true. Also, you cannot specify a calculated column in a GROUP BY clause.

The total width of columns specified in a GROUP BY clause can be up to 100 bytes.

1. To get an in-stock total for each inventory item for all locations, enter:

```
SELECT Part_no, Descript, SUM(On_hand)
  FROM Inventry
  GROUP BY Part_no, Descript;

G_PART_NO   G_DESCRIPT                          SUM1
001001      WORKSTATION-ELECTRONIC OFFICE          5
001002      HOME OFFICE SUITE                      2
001005      EXECUTIVE SUITE ENSEMBLE               1
   .                .                           .
   .                .                           .
   .                .                           .
001032      FILE CABINET-4 DRAWER                 86
001033      CHAIR-TRADITIONAL ARM                 20
001038      LAMP-DRAFTING SWING ARM              305
```

Part_no and Descript information is displayed for each inventory item as if you had entered *SELECT DISTINCT Part_no, Descript FROM Inventry;*. However, GROUP BY allows you to display the inventory quantity of each part for all three locations. Notice that the result is ordered by the values in Part_no, the first column specified in the GROUP BY clause.

The GROUP BY process is illustrated in Figure 28-6.

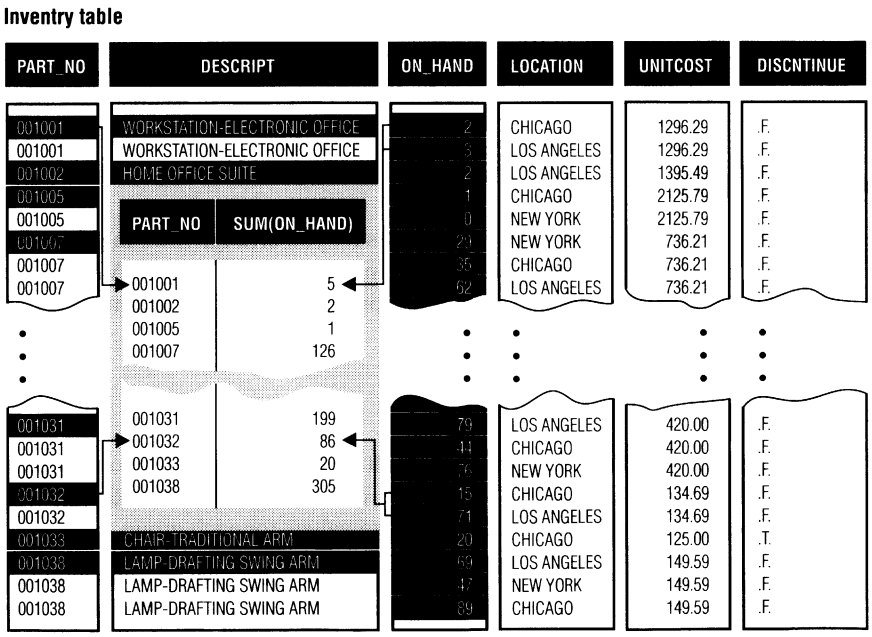


Figure 28-6 Grouping rows for aggregation

2. To count the number of salespeople in each office location, enter:

```
SELECT Location, COUNT(*)
  FROM Staff
  GROUP BY Location;

G_LOCATION     COUNT1
CHICAGO             3
LOS ANGELES         5
NEW YORK            4
```

Use an ORDER BY clause with a GROUP BY clause to order a result differently from the GROUP BY default.

3. To list inventory items by in-stock quantity rather than by part number, enter:

```
SELECT Part_no, Descript, SUM(On_hand)
  FROM Inventry
  GROUP BY Part_no, Descript
  ORDER BY 3;

G_PART NO   G_DESCRIPT                      SUM1
001005      EXECUTIVE SUITE ENSEMBLE          1
001002      HOME OFFICE SUITE                 2
001021      MANAGERS OFFICE ENSEMBLE          3
001022      TABLE-WALNUT OCCASIONAL           5
001001      WORKSTATION-ELECTRONIC OFFICE     5
001019      TABLE-BOARD ROOM                 12
001015      CREDENZA-OAK SLIDING DOOR        15
001033      CHAIR-TRADITIONAL ARM            20
  .                  .                       .
  .                  .                       .
  .                  .                       .
```

Note that using its position number in the SELECT clause (3) to identify the ORDER BY column is mandatory because SUM(On_hand) is a column derived using an expression.

HAVING Clause

You use the HAVING clause to qualify the groups that appear in a GROUP BY result. HAVING qualifies a group result in the same way that WHERE qualifies an ordinary result — by specifying a condition that each group must satisfy. The condition usually includes the aggregate operation specified in the SELECT clause.

To list inventory items costing more than $500 of which there are more than 10 in stock, enter:

```
SELECT Descript, Part_no, SUM(On_hand), Unitcost
  FROM Inventry
  WHERE Unitcost > 500
  GROUP BY Descript, Part_no, Unitcost
  HAVING SUM (On_hand) > 10;

G_DESCRIPT                    G_PART_NO   SUM1      G_UNITCOST
CREDENZA-OAK SLIDING DOOR     001015        15          745
DESK-EXECUTIVE-5 FOOT         001025       110          985
DESK-EXECUTIVE-6 FOOT         001027        76         1475
TABLE-BOARD ROOM              001019        12         4250
WOOD DESK-SINGLE PEDESTAL     001007       126          736
```

NOTE
The HAVING clause is normally used with the GROUP BY clause. If you use HAVING without GROUP BY, HAVING operates on the entire result as if it were a single group.

UNION Clause

You use the UNION clause to combine queries. UNION combines the result tables of two or more SELECT commands and removes duplicate rows.

To display the sales office locations that also stock inventory, enter:

```
SELECT Location
  FROM Staff
      UNION
  SELECT Location
  FROM Inventry;

LOCATION
CHICAGO
LOS ANGELES
NEW YORK
```

As the result shows, the salespeople and what they sell are located in the same places. Figure 28-7 illustrates a UNION operation using Customer and Staff tables.

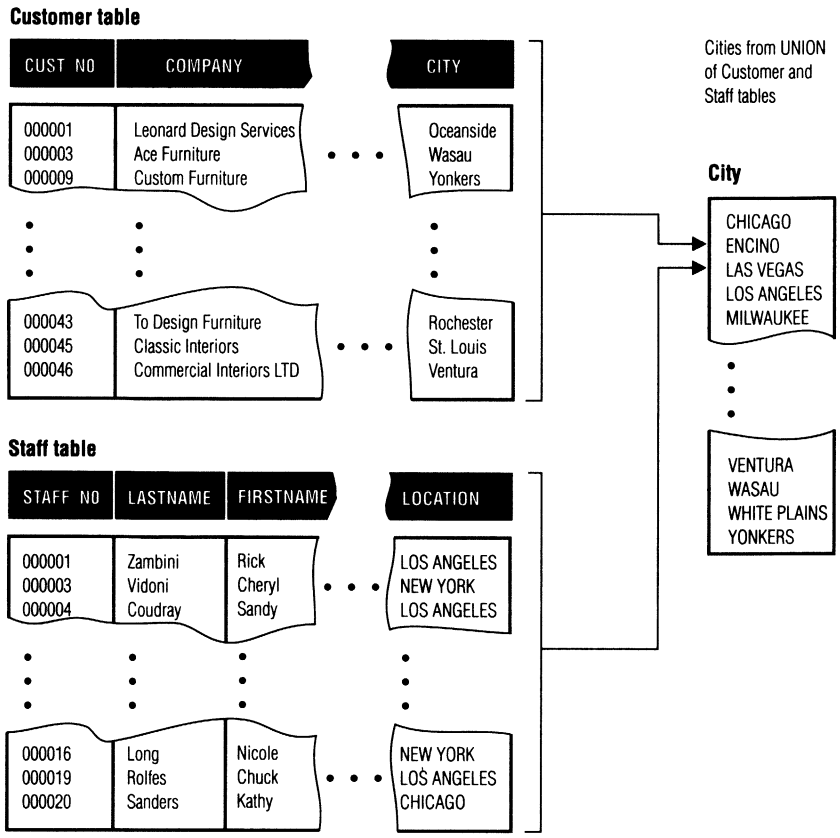


Figure 28-7 Combining queries with the UNION clause

The following rules govern the SELECT clauses used to specify a union:

- Each SELECT clause must have the same number of columns, and the data types of corresponding columns must match. However, corresponding column names needn't match.
- The widths of corresponding character columns must match.
- Corresponding numeric columns must be of the same data type (fixed or floating point), must contain the same number of digits, and, if decimal, must have the same number of decimal point digits.
- SELECT clauses cannot contain a column with a LOGICAL data type, or a dBASE function or memory variable.

A UNION result is ordered by the values of all columns specified in each SELECT clause. If you want the result ordered by the values of a different column, use ORDER BY as the last clause of the SELECT command. Use a number indicating the left-to-right position of the ORDER BY column, for example, *ORDER BY 2*.

To display the names of all the cities in which you have staff, inventory, or customers, enter:

```
SELECT City
  FROM Customer
      UNION
  SELECT Location
  FROM Staff
      UNION
  SELECT Location
  FROM Inventry;

CITY
CHICAGO
Chicago
Encino
LOS ANGELES
Las Vegas
Los Angeles
Milwaukee
NEW YORK
New York
Oceanside
Phoenix
Rochester
San Francisco
St. Louis
Ventura
Wasau
White Plains
Yonkers
```

NOTE
Apparent duplicates in this result (for example, CHICAGO and Chicago) reflect the fact that City values (entered in initial caps and lowercase) and Location values (entered in uppercase) do not have the same ASCII representation.

Quitting dBASE IV

To exit from dBASE IV and return to the operating system, enter:

```
SQL. QUIT
```

Summary

In this chapter, you used SELECT commands to retrieve data from a single table. The essentials of entering SELECT commands are summarized below.

To display all data from specific columns:
Enter the SELECT <column name,...> FROM <table name> command.

To display all data in a table:
Enter the SELECT * FROM <table name> command.

To display only one of a number of identical rows:
Enter the SELECT DISTINCT <column name,...> FROM <table name> command.

To display data from specific rows:
Enter the SELECT <column name,...> FROM <table name> WHERE <search expression> command.

To combine search conditions:
Combine search expressions in a WHERE <search expression> clause using NOT, AND, and OR logical operators. Use parentheses to group expressions for evaluation, for example: WHERE (Location = "Los Angeles" OR Location = "New York") AND (On_hand < 20 AND Unitcost < 1000).

Using expressions:
Expressions can consist of column names, arithmetic operators, constants (numeric, logical, date, and character), memory variables, dBASE functions, and the USER keyword. Expressions can be used in the following clauses of the SELECT command:

SELECT — to define a result column describing data in an adjacent result column, for example, "(Yearly Salary)"; to define a calculated result column, for example, Salary*12. You can use constants or memory variables in an expression.

WHERE — to define search conditions. For example, in the clause WHERE Salary*12 > 50000, the expression Salary*12 > 50000 is an expression composed of two expressions, Salary*12 and 50000.

To use dBASE functions in expressions:
Use dBASE functions such as UPPER() and CTOD() to convert search expression values for comparison. For example: DATE() – Hiredate, to determine the number of days between a person's hire date and the current system date on your computer; UPPER(City), to convert City column values to uppercase for comparison with an uppercase value.

To use SQL aggregate functions in expressions:
Use aggregate functions in a SELECT clause to summarize column values in a result. For example: COUNT(*), to count the number of rows returned; SUM(Salary), to add Salary column values in returned rows.

To use predicates in search conditions:

Use the BETWEEN, IN, and LIKE predicates in the WHERE clause to qualify result rows. For example: WHERE Salary BETWEEN 5000 AND 6000 (inclusive); WHERE State IN ("AZ", "MO", "WI"); WHERE State LIKE "N_" or WHERE Descript LIKE "%DESK%".

To order results:

Use the ORDER BY clause to sequence a SELECT result by the values in one or more SELECT clause columns. ASCending (lowest to highest) is the default, but can be redundantly specified using the ASC keyword; DESCending (highest to lowest) can be specified using the DESC keyword. For example: ORDER BY Descript ASC; ORDER BY Location, On_hand (DESC).

You can represent a column name by its numeric position in the SELECT clause, for example: ORDER BY 3, 4 DESC.

To group result rows:

Use the GROUP BY clause to group a result by common values in one or more columns. Each group is summarized as a single result row. Each column in the SELECT clause must be included in GROUP BY unless it is used to specify an aggregate operation. For example: SELECT Location, COUNT(*) FROM Staff GROUP BY Location;.

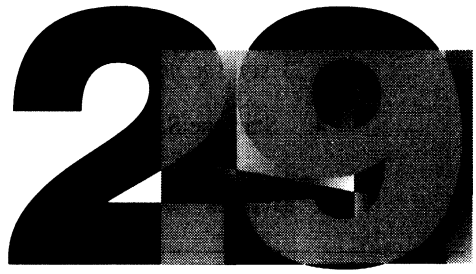
To qualify groups for inclusion in the result:

Use the HAVING clause with GROUP BY. HAVING normally includes the aggregate operation specified in the SELECT clause. For example: SELECT Part_no, Descript, SUM(On_hand) FROM Inventry WHERE Unitcost > 500 GROUP BY Part_no, Descript HAVING SUM(On_hand) > 10;.

To combine queries:

Use the UNION clause to combine the result tables of two or more SELECT commands and remove duplicate rows. Each SELECT clause must have the same number of columns and corresponding columns must have the same data type (though not the same name); the widths of corresponding character columns must be the same; corresponding numeric columns must have the same data type (fixed or floating point) and the same number of digits (if decimal, the same number of decimal point digits); SELECT clauses cannot contain logical-type columns, dBASE functions, or memory variables.

Joins and Subqueries



In the Samples database, each table contains a different type of information. Except for common columns that allow you to relate tables, the sample tables do not contain redundant information. Therefore, you'll often want to combine data from different tables.

This chapter shows you how to use SELECT queries to do this. You will learn how to:

- Combine, or join, information from more than one table
- Limit joined rows to those that satisfy a search condition
- Order a join result
- Group joined rows and limit the groups displayed
- Join a table with a copy of itself
- Use a nested SELECT command, or subquery, to supply one or more values to the search condition of a SELECT command
- Nest more than one subquery

For complete information about any SQL command used in this manual, refer to Chapter 6 of *Language Reference.*

Preparing for This Chapter

Start dBASE IV:

1. Enter dbase at the operating system prompt.
2. If the Control Center is displayed, choose **Exit to dot prompt** from the **Exit** menu to access the dot prompt.

3. Enter SQL mode by entering:

```
. SET SQL ON
```

4. Start the Samples database:

```
SQL. START DATABASE Samples;
Database SAMPLES started
```

Joins

You can use a single SELECT command to combine information from several tables. This operation is called a *join*. In the SELECT command that defines a join, you can:

- Specify the columns in each table that you want to appear in a result.
- Use the asterisk (*) symbol to display all columns from all the joined tables.
- Use the WHERE clause to specify *join conditions* that limit the rows that appear in a result.
- Use GROUP BY and HAVING clauses to group a result.
- Use the ORDER BY clause to order a result.
- Use SAVE TO TEMP...KEEP to create a database file.

To join the Sales and Staff tables, enter:

```
SELECT Order_no, Staff.Staff_no, Lastname
  FROM Sales, Staff
  WHERE Sales.Staff_no = Staff.Staff_no;

SALES->ORDER_NO   STAFF->STAFF_NO   STAFF->LASTNAME
020002            000008            McLester
020003            000006            Thomas
020004            000019            Rolfes
   .                 .                 .
   .                 .                 .
   .                 .                 .

020023            000003            Vidoni
020024            000012            Charles
020025            000003            Vidoni
020026            000004            Coudray
```

This command joins data from the Order_no column of the Sales table with data from the Staff_no and Lastname columns of the Staff table. Because the Staff_no column occurs in both Sales and Staff tables, you use the prefix *Staff.* in the SELECT clause to specify that this information is to be retrieved from the Staff table.

The WHERE clause contains the join condition *Sales.Staff_no = Staff.Staff_no* to specify that each Sales row be joined with the Staff row that contains the same Staff_no value.

NOTE
If no join condition is supplied in a WHERE clause, every row in each table specified in the FROM clause is joined with every row in the other tables.

Enter:

```
SELECT Order_no, Sales.Staff_no, Staff.Staff_no, Lastname
  FROM Sales, Staff
  WHERE Sales.Staff_no = Staff.Staff_no;

SALES->ORDER_NO  SALES->STAFF_NO  STAFF->STAFF_NO  STAFF->LASTNAME
020002           000008           000008           McLester
020003           000006           000006           Thomas
020004           000019           000019           Rolfes
  .                .                .                .
  .                .                .                .
  .                .                .                .

020023           000003           000003           Vidoni
020024           000012           000012           Charles
020025           000003           000003           Vidoni
020026           000004           000004           Coudray
```

This result shows you more graphically how columns from the Sales table are joined with Staff table columns.

Each of these commands defines an *equijoin*, so called because the join condition uses the equals operator. The first example is also called a *natural join* because only one of the columns used in the join condition is displayed in the result.

The names of the columns used in a join condition need not be the same. However, the columns must have compatible data types.

How dBASE IV SQL Joins Tables

SQL first combines the rows that are to be joined and then applies the join condition to produce the result. Figure 29-1 illustrates the process.

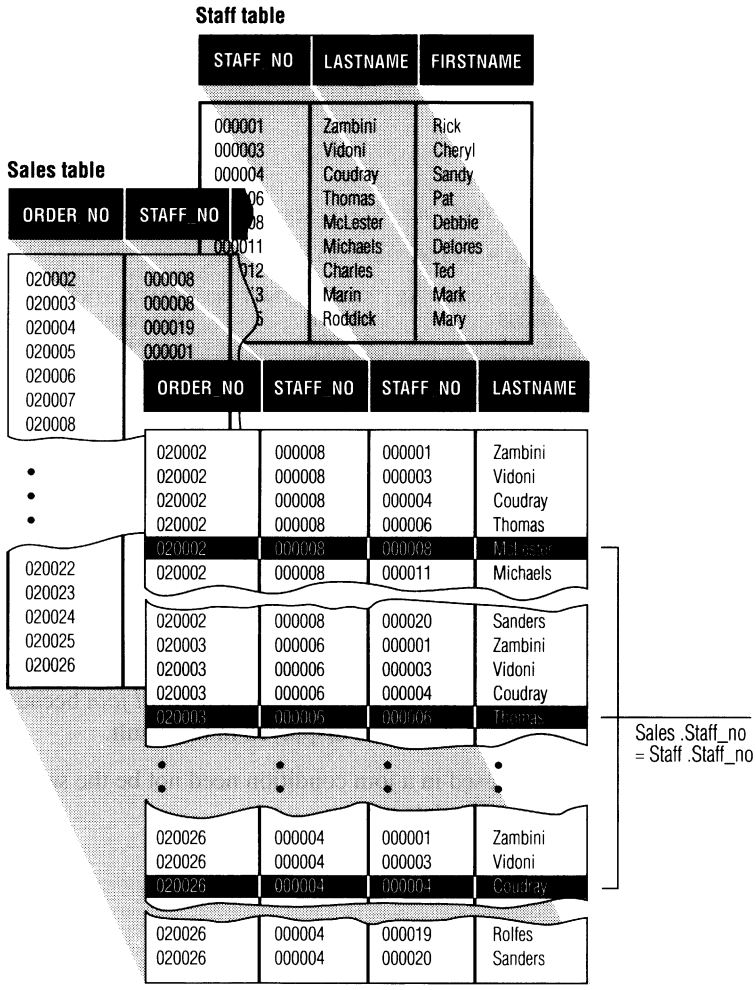


Figure 29-1 Equijoin

Selecting Data from a Join

Besides the join condition, the WHERE clause can contain search conditions.

To display orders only for salesperson Charles, enter:

```
SELECT Order_no, Staff.Staff_no, Lastname
  FROM Sales, Staff
  WHERE Sales.Staff_no = Staff.Staff_no
    AND Lastname = "Charles";

SALES->ORDER_NO  STAFF->STAFF_NO  STAFF->LASTNAME
020006           000012           Charles
020009           000012           Charles
020013           000012           Charles
020024           000012           Charles
```

To combine information from the Inventry and Items tables that you might use for printing an invoice form for order 020021, enter:

```
SELECT DISTINCT Items.Part_no, Qty, Unitcost, (Qty*Unitcost)
  FROM Items, Inventry
  WHERE Inventry.Part_no = Items.Part_no
    AND Order_no = "020021";

ITEMS->PART_NO  ITEMS->QTY  INVENTRY->UNITCOST        EXP1
001013                   8              275.80     2206.40
001024                   6              230.79     1384.74
001025                   8              985.00     7880.00
```

For each item ordered, the result includes part number and quantity information from the Items table and unit cost information from the Inventry table. The calculated column *Qty*Unitcost* displays total cost for each item. The DISTINCT keyword eliminates duplicate rows, as shown in Figure 29-2.

PART NO	QTY	UNITCOST	QTY·UNITCOST
		275.80	2206.40
001013	8	275.80	2206.40
		230.79	1384.74
001024	6	230.79	1384.74
		985.00	7880.00
001025	8	985.00	7880.00

Figure 29-2 DISTINCT keyword used in a join

Greater-Than, Less-Than Join Operators

Besides the equals operator, you can also use a greater-than (>) or less-than (<) operator in a join condition.

To locate items in sufficient quantity to fill an order, enter:

```
SELECT Items.Part_no, Qty, On_hand, Location
  FROM Items, Inventry
  WHERE Items.Part_no = Inventry.Part_no
    AND On_hand > Qty AND Items.Order_no = "020021";

ITEMS->PART_NO  ITEMS->QTY  INVENTRY->ON_HAND   INVENTRY->LOCATION
001013                   8                115   CHICAGO
001013                   8                 35   LOS ANGELES
001025                   8                 63   LOS ANGELES
001025                   8                 47   NEW YORK
001024                   6                140   CHICAGO
001024                   6                 56   NEW YORK
```

ORDER BY Clause

You can use the ORDER BY clause to order a join result. Values can be arranged in ascending (ASC) or descending (DESC) order or a combination of the two.

To list orders and their dates for each salesperson, enter:

```
SELECT Order_no, Sale_date, Lastname
  FROM Sales, Staff
  WHERE Sales.Staff_no = Staff.Staff_no
  ORDER BY Sale_date DESC, Lastname;

SALES->ORDER_NO  SALES->SALE_DATE  STAFF->LASTNAME
020024           09/25/87          Charles
020026           09/25/87          Coudray
020022           09/25/87          Coudray
020021           09/25/87          McLester
020025           09/25/87          Vidoni
020023           09/25/87          Vidoni
020020           09/24/87          Marin
020019           09/24/87          Marin
020018           09/24/87          Michaels
020017           09/24/87          Thomas
020013           09/23/87          Charles
020016           09/23/87          Roddick
  .                 .                 .
  .                 .                 .
  .                 .                 .

020008           09/22/87          Vidoni
020002           09/21/87          McLester
020004           09/21/87          Rolfes
020003           09/21/87          Thomas
020005           09/21/87          Zambini
```

Rows are displayed by sale date, from most recent to earliest. For each date, rows are displayed alphabetically by salesperson's last name. Enter:

```
SELECT Order_no, Sale_date, Lastname
  FROM Sales, Staff
  WHERE Sales.Staff_no = Staff.Staff_no
  ORDER BY Lastname, Sale_date DESC;
```

Notice that result rows are in alphabetical order by salesperson's last name. For each salesperson, rows are in descending order by date.

GROUP BY and HAVING Clauses

Use the GROUP BY clause to display a single row for each group in a join result.

To display number of sales for each salesperson, enter:

```
SELECT Staff.Staff_no, Lastname, COUNT(*)
  FROM Staff, Sales
  WHERE Staff.Staff_no = Sales.Staff_no
  GROUP BY Staff.Staff_no, Lastname;

G_STAFF_NO  G_LASTNAME          COUNT1
000001      Zambini                  2
000003      Vidoni                   3
000004      Coudray                  2
000006      Thomas                   2
000008      McLester                 3
000011      Michaels                 2
000012      Charles                  4
000013      Marin                    2
000015      Roddick                  4
000019      Rolfes                   1
```

The COUNT(*) aggregate function counts the number of rows in the Sales table (each row representing a sale) for each salesperson "group."

Use the HAVING clause to qualify the rows displayed in the result, and ORDER BY to order result table rows differently from the default GROUP BY ordering.

To display information for any salesperson who has more than two sales, enter:

```
SELECT Staff.Staff_no, Lastname, COUNT(*)
  FROM Staff, Sales
  WHERE Staff.Staff_no = Sales.Staff_no
  GROUP BY Staff.Staff_no, Lastname
  HAVING COUNT(*) > 2
  ORDER BY 3;

G_STAFF_NO  G_LASTNAME          COUNT1
000003      Vidoni                   3
000008      McLester                 3
000012      Charles                  4
000015      Roddick                  4
```

Figure 29-3 shows the operation of GROUP BY and HAVING clauses in this example.

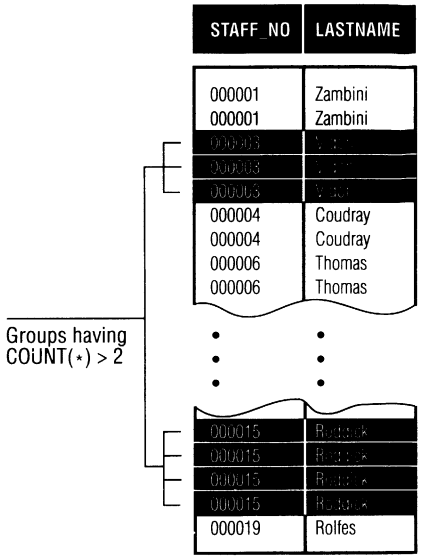


Figure 29-3 GROUP BY and HAVING in a join

Joining More than Two Tables

When you join more than two tables, each additional table normally contains an additional join condition, connected by the AND or OR logical operator.

To join the Staff, Sales, and Customer tables to relate customer orders to salespeople, enter:

```
SELECT Staff.Lastname, Order_no, Company
  FROM Staff, Sales, Customer
  WHERE Staff.Staff_no = Sales.Staff_no
    AND Sales.Cust_no = Customer.Cust_no
  ORDER BY Staff.Lastname;

STAFF->LASTNAME  SALES->ORDER_NO  CUSTOMER->COMPANY
Charles          020006           New Horizons
Charles          020009           Interior Systems
Charles          020013           New Horizons
Charles          020024           Classic Interiors
Coudray          020022           A1 Office Supply Store
Coudray          020026           Black's Furniture Store
   .                .                      .
   .                .                      .
   .                .                      .

Thomas           020017           Contemporary Designs
Vidoni           020008           The Office
Vidoni           020023           Design Center Interiors
Vidoni           020025           The Designer
Zambini          020005           American Business Supply
Zambini          020014           Leonard Design Services
```

The Staff and Sales tables are joined on common values in their Staff_no columns. The Sales and Customer tables are joined on common values in their Cust_no columns. The result is ordered by salespersons' last names.

Joining a Table with Itself

You can join a table with a copy of itself to perform a *self-join*. A self-join allows you to relate information from different rows of the same table.

In a self-join, you define different names, or *aliases*, for the table. SQL treats each alias as if it referred to a different table.

To join the Staff table with itself to list the supervisor for each salesperson, enter:

```
SELECT Super.Lastname,"Supervises:", Emp.Lastname
  FROM Staff Emp, Staff Super
  WHERE Super.Staff_no = Emp.Supervisor
  ORDER BY Super.Lastname DESC;

SUPER->LASTNAME    EXP1         EMP->LASTNAME
Zambini            Supervises:  Rolfes
Zambini            Supervises:  Marin
Zambini            Supervises:  McLester
Zambini            Supervises:  Coudray
Vidoni             Supervises:  Long
Vidoni             Supervises:  Roddick
Vidoni             Supervises:  Thomas
Charles            Supervises:  Sanders
Charles            Supervises:  Michaels
```

In this command, Super and Emp are defined as aliases for the Staff table. The alias tables are joined on the Staff_no column and the Supervisor column, which contains the Staff_no value for each salesperson's supervisor.

Figure 29-4 illustrates the operation of this self-join.

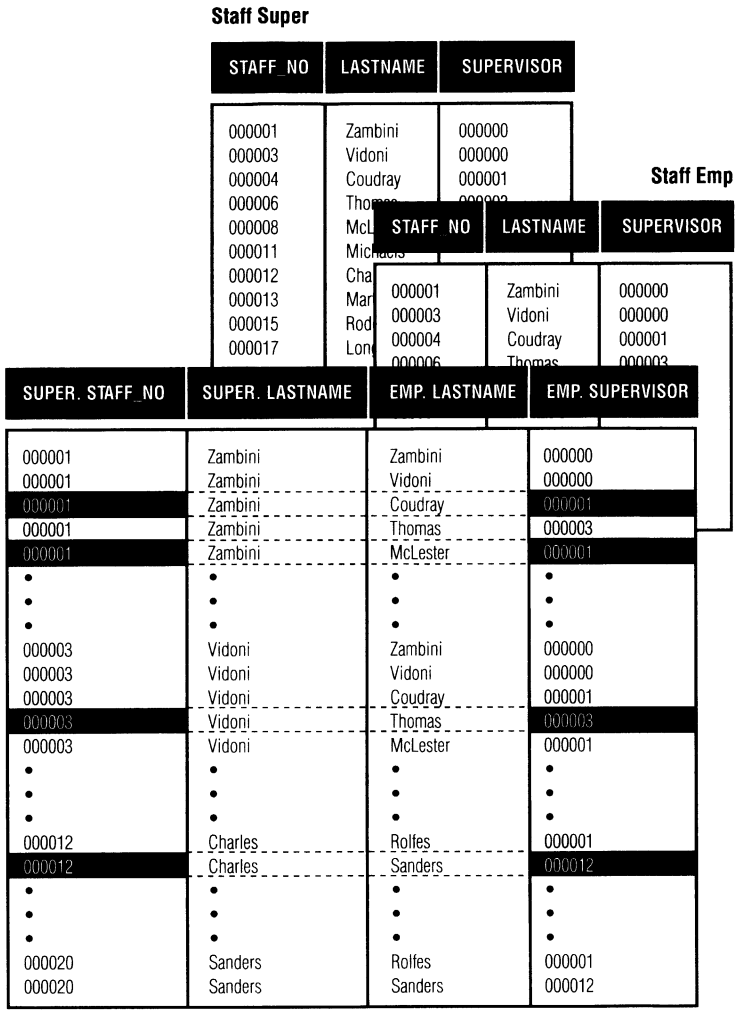


Figure 29-4 Self-join

Subqueries

A subquery is a SELECT command that is *nested* within the WHERE clause or HAVING clause of another SELECT command. The subquery, also called an *inner* query, supplies values for the search condition of the *outer* query, the command that contains the inner query.

The SELECT clause of an inner query can contain only one column name. If the EXISTS predicate, discussed later in this chapter, is used in the WHERE clause of the outer query, the * symbol also can be used.

There are two types of SELECT subquery commands:

- Simple — the inner query is evaluated first and its result used to evaluate the outer query.
- Correlated — the outer query is evaluated first and its result used to evaluate the inner query. The inner query is evaluated once for each row returned by the outer query.

In a simple subquery command, the construction of the outer query depends on whether the inner query returns a single value or multiple values.

Inner Query Returning a Single Value

When the inner query of a subquery command returns a single value, the value is supplied to the WHERE clause of the outer query as if you had typed it in.

1. To display information about orders for American Business Supply when you don't recall its customer number, enter:

```
SELECT Order_no, Sale_date
  FROM Sales
  WHERE Cust_no =
    (SELECT Cust_no
    FROM Customer
    WHERE Company = "American Business Supply")
  ORDER BY Sale_date;

ORDER_NO  SALE_DATE
020005    09/21/87
020019    09/24/87
```

   The inner query returns the customer number value for American Business Supply to the WHERE clause of the outer query. The outer query matches this value with Cust_no values in the Sales table to select the result rows. The result is the same as if you had used the clause *WHERE Cust_no = "000016"* in the outer query.

NOTE

If, in a simple subquery, the WHERE condition of the outer query uses a comparison operator and the inner query returns more than one value, an error occurs. However, if you combine the outer comparison operator with the ANY or ALL predicate, discussed later in this chapter (for example, > ANY, = ALL), the subquery is valid. If you use = as the outer comparison operator without ANY or ALL and the inner query returns more than one row, a simple subquery fails while a correlated subquery succeeds. Refer to the Correlated Subqueries section later in this chapter.

2. To list salespeople who are earning higher than average salaries, enter:

```
SELECT Lastname, Salary
  FROM Staff
  WHERE Salary >
    (SELECT AVG(Salary)
    FROM Staff)
  ORDER BY Lastname;

LASTNAME        SALARY
Charles           5945
Coudray           6237
Roddick           5493
Thomas            5875
Vidoni            5780
Zambini           6000
```

The inner query uses the AVG() aggregate function to compute the average monthly salary from Salary values in the Staff table. The outer query uses the average value to return rows for salespeople with higher salaries.

You can use the logical operators AND, OR, and NOT in the outer WHERE clause to specify more than one search condition requiring an inner query.

3. To view information about orders for American Business Supply or for orders taken by Zambini when you don't recall the company's customer number or the salesperson's staff number, enter:

```
SELECT Order_no, Sale_date
  FROM Sales
  WHERE Cust_no =
    (SELECT Cust_no
    FROM Customer
    WHERE Company = "American Business Supply")
      OR Staff_no =
    (SELECT Staff_no
    FROM Staff
    WHERE Lastname = "Zambini")
  ORDER BY Order_no;

ORDER_NO  SALE_DATE
020005    09/21/87
020014    09/23/87
020019    09/24/87
```

The first inner query supplies the customer number for American Business Supply from the Customer table, and the second inner query supplies the staff number for Zambini from the Staff table. The two inner queries are linked by the OR operator to express a WHERE condition that could be typed as: *WHERE Cust_no = "000016" OR Staff_no = "000001".*

Inner Query Returning Multiple Values

For an inner query to be able to return more than one value to the outer query, the outer WHERE clause must contain the IN, ANY, or ALL predicate. If not, an error occurs.

IN Predicate

You use the IN predicate in an outer WHERE clause to select rows whose values match the values returned by the inner query. These values are used in place of a value list that you might type.

To retrieve information about orders for customers located in California, enter:

```
SELECT Order_no, Cust_no, Sale_date
  FROM Sales
  WHERE Cust_no IN
    (SELECT Cust_no
    FROM Customer
    WHERE Zip LIKE "9%")
  ORDER BY Order_no;

ORDER_NO  CUST_NO  SALE_DATE
020004    000034   09/21/87
020005    000016   09/21/87
020012    000027   09/22/87
020014    000001   09/23/87
020019    000016   09/24/87
020021    000046   09/25/87
020022    000027   09/25/87
020026    000017   09/25/87
```

The inner query selects customer numbers from Customer table rows whose Zip values begin with 9 (those for California). The customer numbers are returned to the outer WHERE clause and used to look up orders in the Sales table.

Figure 29-5 illustrates the operation of this subquery.

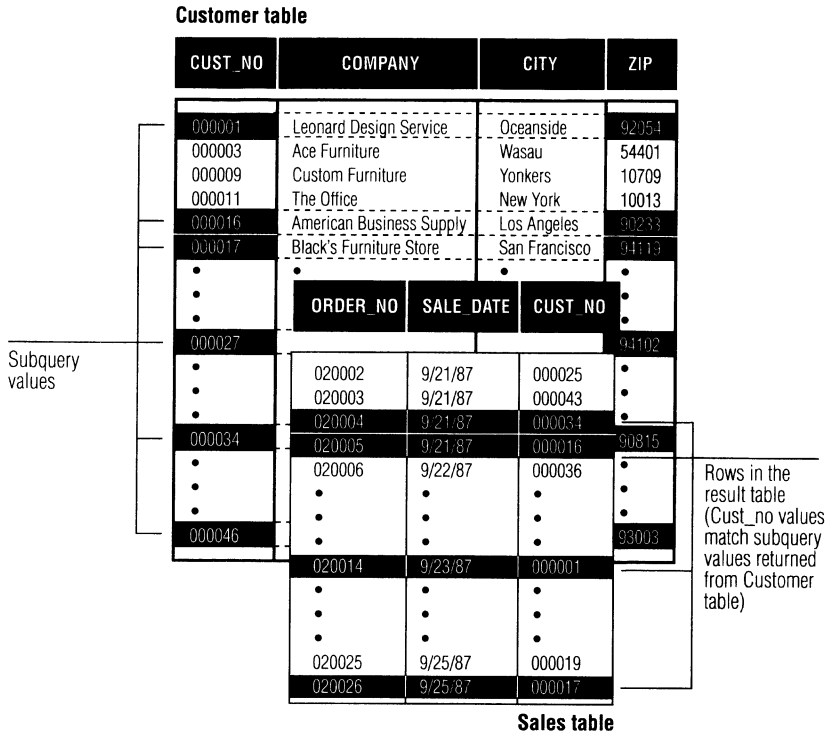


Figure 29-5 Subquery returning multiple values

NOTE
An inner query cannot contain an ORDER BY or a UNION clause.

ANY Predicate

Use the ANY keyword in an outer WHERE condition to select rows whose values match any of the values returned by the inner query.

To display information for salespeople whose commission rate is less than that of any salesperson in Chicago, enter:

```
SELECT Staff_no, Firstname, Lastname, Hiredate, Location,
  Commission
  FROM Staff
  WHERE Commission < ANY
    (SELECT Commission
    FROM Staff
    WHERE Location = "CHICAGO")
  ORDER BY Staff_no;

STAFF_NO  FIRSTNAME  LASTNAME  HIREDATE  LOCATION     COMMISSION
000001    Rick       Zambini   02/15/80  LOS ANGELES         5.0
000003    Cheryl     Vidoni    03/06/80  NEW YORK            5.0
000004    Sandy      Coudray   06/06/80  LOS ANGELES         5.0
000006    Pat        Thomas    01/08/81  NEW YORK            5.0
000008    Debbie     McLester  04/12/81  LOS ANGELES         5.0
000012    Ted        Charles   02/02/83  CHICAGO             5.0
000019    Chuck      Rolfes    09/09/84  LOS ANGELES         6.0
000020    Kathy      Sanders   03/23/85  CHICAGO             5.0
```

The inner query supplies Commission values for Chicago salespeople to the outer query. The outer query compares each value with the Commission value in each Staff row, and selects rows whose values are lower than any one of these values. Because the highest commission rate for a Chicago salesperson is 7.0, most Staff rows are selected.

You can use any comparison operator with ANY in an outer WHERE clause.

NOTE
The expression !=ANY is not the opposite of =ANY.

ALL Predicate

You use the ALL predicate in an outer WHERE condition to select rows whose values have some relation to all of the values returned by the inner query.

To display information for salespeople whose commission is higher than the highest rate earned at the Chicago location, enter:

```
SELECT Staff no, Firstname, Lastname, Hiredate, Location,
  Commission
  FROM Staff
  WHERE Commission > ALL
    (SELECT Commission
    FROM Staff
    WHERE Location = "CHICAGO")
  ORDER BY Staff_no;

STAFF_NO  FIRSTNAME  LASTNAME  HIREDATE  LOCATION     COMMISSION
000013    Mark       Marin     06/05/83  LOS ANGELES        11.0
000015    Mary       Roddick   02/13/84  NEW YORK            8.0
```

Nesting One Inner Query within Another

You can nest an inner query within another inner query in a subquery command.

1. To display information for all customers who have ordered a certain item, enter:

```
SELECT Cust_no, Company, Firstname, Lastname, City
  FROM Customer
  WHERE Cust_no IN
   (SELECT Cust_no
   FROM Sales
   WHERE Order_no IN
        (SELECT Order_no
        FROM Items
        WHERE Part_no = "001025"))
  ORDER BY Cust_no;
```

```
CUST_NO  COMPANY                  FIRSTNAME   LASTNAME   CITY
000011   The Office               Dominique   LeClerc    New York
000019   The Designer             Luke        Hobbs      New York
000025   Modern Furniture Store   Robert      Hamilton   Phoenix
000040   Design Center Interiors  Chuck       Gilbert    Las Vegas
000046   Commercial Interiors LTD Sandy       Young      Ventura
```

   The second inner query supplies order numbers for the specified part to the WHERE clause of the first inner query. The first inner query supplies the customer numbers associated with the order numbers to the WHERE clause of the outer query. The outer query uses the customer numbers to select Customer rows.

2. To express this query as a join, enter:

```
SELECT Customer.Cust_no, Company,
  Firstname, Lastname, City
  FROM Customer, Sales, Items
  WHERE Customer.Cust_no = Sales.Cust_no
    AND Sales.Order_no = Items.Order_no
    AND Items.Part_no = "001025"
  ORDER BY 1;
```

   Any subquery command can be expressed as a join. However, a subquery command is often easier to conceptualize than the equivalent join. Not all joins can be expressed as subqueries.

EXISTS Predicate

The EXISTS predicate determines whether the outer WHERE condition is true or false. If the inner query returns any value, the WHERE condition is true. Otherwise, the condition is false.

A simple subquery that uses EXISTS returns all rows if the outer WHERE condition is true; otherwise, it returns no rows. For example, enter:

```
SELECT Staff_no, Lastname
  FROM Sales
  WHERE EXISTS
    (SELECT *
    FROM Staff
    WHERE Staff.Staff_no = Sales.Staff_no);
```

This command retrieves Staff_no and Lastname information for all salespeople listed in the Sales table, just as if you had entered the outer query without a WHERE condition.

Almost all subqueries that use EXISTS are correlated. For example, you could use the following correlated subquery to display information for all customers who have placed orders within the past week:

```
SELECT Cust_no, Company, City, State
  FROM Customer
  WHERE EXISTS
    (SELECT *
    FROM Sales
    WHERE Customer.Cust_no = Sales.Cust_no)
  ORDER BY Cust_no;
```

```
CUST_NO  COMPANY                    CITY           STATE
000001   Leonard Design Services    Oceanside      CA
000011   The Office                 New York       NY
000016   American Business Supply   Los Angeles    CA
000017   Black's Furniture Store    San Francisco  CA
000018   Interior Systems           Milwaukee      WI
000019   The Designer               New York       NY
000025   Modern Furniture Store     Phoenix        AZ
000027   A1 Office Supply Store     San Francisco  CA
000031   Al's Furniture & Supplies  St. Louis      MO
000032   Contemporary Designs       Milwaukee      WI
000034   La Cienega Furniture       Los Angeles    CA
000036   New Horizons               Chicago        IL
000040   Design Center Interiors    Las Vegas      NV
000043   To Design Furniture        Rochester      NY
000045   Classic Interiors          St. Louis      MO
000046   Commercial Interiors LTD   Ventura        CA
```

In this command, EXISTS is true for customers whose customer numbers appear in the Sales table (that is, customers who have placed an order within the past week). Note that you could use WHERE NOT EXISTS to display information for customers who do not have a current order.

NOTE
*EXISTS is the only predicate whose inner query can contain SELECT *.*

Correlated Subqueries

In a correlated subquery, the outer query supplies the inner query with a value from each of its result rows. The inner query is evaluated once for each of these values.

If both the inner and outer queries access the same table, you need to assign aliases for the table. This is because the inner and outer queries need to access different rows within the same table simultaneously.

To display information for the salesperson at each location who has the highest salary, enter:

```
SELECT Outer.Firstname, Outer.Lastname, Outer.Location,
  Outer.Hiredate, Outer.Salary
  FROM Staff Outer
  WHERE Salary =
    (SELECT MAX(Salary)
    FROM Staff Inner
    WHERE Inner.Location = Outer.Location)
  ORDER BY Outer.Lastname;
```

```
OUTER>FIRSTNAME  OUTER->LASTNAME  OUTER->LOCATION  OUTER->HIREDATE  OUTER->SALARY
Ted              Charles         CHICAGO          02/02/83          5945
Sandy            Coudray         LOS ANGELES      06/06/80          6237
Pat              Thomas          NEW YORK         01/08/81          5875
```

The outer query supplies the Location value of each row of the Staff table (alias Outer), in turn, to the inner query. The inner query retrieves the maximum Salary value for that location from Staff (alias Inner) and supplies it to the outer query.

The outer query then compares the Salary value in the current row with the maximum value returned by the inner query. If the current-row value matches the maximum, the row is selected for the result table.

Figure 29-6 illustrates this selection process.

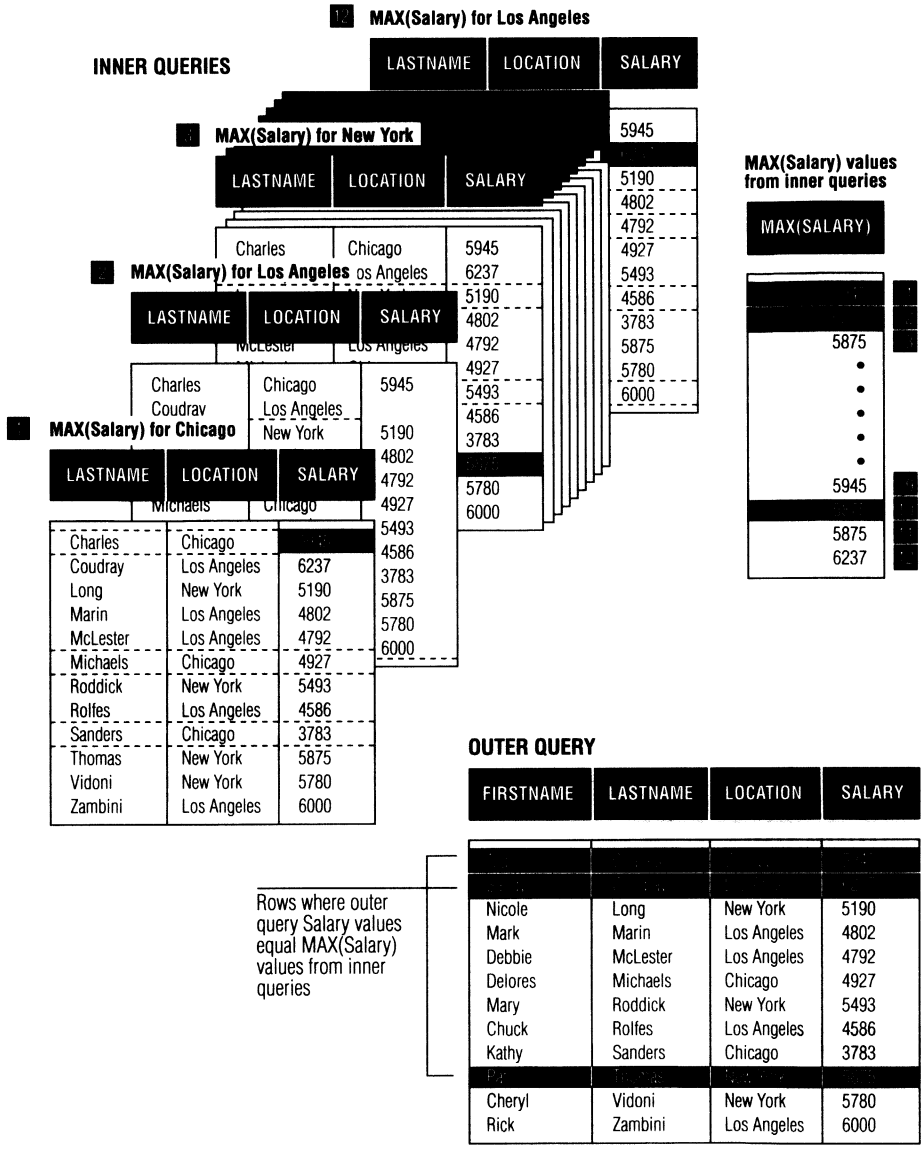


Figure 29-6 Correlated subquery

To display customer number and order number for all orders for customers who have more than one current order, enter:

```
SELECT S1.Cust_no, S1.Order_no
  FROM Sales S1
  WHERE EXISTS
    (SELECT *
    FROM Sales S2
    WHERE S1.Order_no <> S2.Order_no
    AND S1.Cust_no=S2.Cust_no)
  ORDER BY S1.Cust_no, S1.Order_no;
```

```
S1->CUST_NO   S1->ORDER_NO
000011        020008
000011        020016
000016        020005
000016        020019
000019        020007
000019        020015
000019        020025
000027        020012
000027        020022
000031        020010
000031        020020
000036        020006
000036        020013
000040        020011
000040        020023
```

The outer query supplies the Cust_no value of each Sales (alias S1) row to the inner query, which looks for rows in Sales (alias S2) that contain the same value. If it finds more than one row, the current row in S1 is chosen for the result.

Quitting dBASE IV

To exit from dBASE IV and return to the operating system, enter:

```
SQL. QUIT
```

Summary

In this chapter, you used joins and subqueries to combine data from different tables. The principles of combining table data are as follows:

To join two tables (equijoin):

In the SELECT command used to specify a join, use the SELECT clause to name the columns in each table that you want to include in the result. If columns have the same name in both tables, use the form <table name>.<column name> to specify from which table you want the data. Use an asterisk (*) to display all columns from both tables. For example: SELECT Order_no, Staff.Staff_no, Lastname.

Use the FROM clause to name the two tables. For example: FROM Sales, Staff.

Use the WHERE clause to specify the *join condition*. The join condition names a column in each table whose values are to be paired to join rows. The names of these columns need not be the same, but their data types must be compatible. The clause also can include a search expression for qualifying result rows. For example: WHERE Sales.Staff_no = Staff.Staff_no AND Lastname = "Charles".

To perform a natural join:

In the SELECT clause of an equijoin, name only one of the columns used to specify the join condition.

To order a join result:

Use the ORDER BY clause to order a join result. Values used for ordering can be arranged in ascending (ASC) or descending (DESC) order, or a combination of the two. For example, ORDER BY Sale_date DESC, Lastname.

To group a join result:

Use the GROUP BY clause to display a single row for each group in a join result. Use the HAVING clause to qualify the rows that are included in the result. For example: GROUP BY Staff.Staff_no, Lastname HAVING Count(*) > 2.

To join more than two tables:

For each additional table, use the AND or OR logical operator to specify another join condition. For example: Where Staff.Staff_no = Sales.Staff_no AND Sales.Cust_no = Customer.Cust_no.

To join a table with itself (self-join):

To join a table with a copy of itself to relate information from different rows, define an *alias* name for both the table and its copy. SQL treats each alias as if it referred to a different table. For example: SELECT Super.Lastname, Emp.Lastname FROM Staff Emp, Staff Super WHERE Super.Staff_no = Emp.Supervisor;.

To create a subquery:

Nest a SELECT command (inner query) within the WHERE clause of another SELECT command (outer query). If the subquery returns a single value, it is supplied to the WHERE clause of the outer query just as if you had typed it in. For example: SELECT Lastname, Salary FROM Staff WHERE Salary > (SELECT AVG(Salary) FROM Staff);.

More than one subquery can be nested. In this case, the innermost query supplies a value to the next inner query, and so on. The outermost query supplies a value to the outer WHERE condition.

If the inner query returns more than one value:

Use the IN predicate in the outer WHERE condition to select rows whose values match the value returned by the inner query. For example: SELECT Order_no, Cust_no, Sale_date FROM Sales WHERE Cust_no IN (SELECT Cust_no FROM Customer WHERE Zip LIKE "9%");.

Use the ANY predicate in the outer WHERE condition to select rows whose values match any value returned by the inner query. For example: SELECT Staff_no, Lastname, Commission FROM Staff WHERE Commission < ANY (SELECT Commission FROM Staff WHERE Location = "Chicago");.

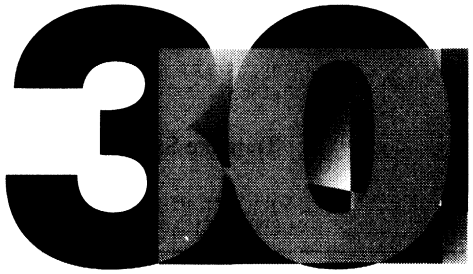
Use the ALL predicate in the outer WHERE condition to select rows whose values match all of the values returned by the inner query. For example: SELECT Staff_no, Lastname, Commission FROM Staff WHERE Commission > ALL (SELECT Commission FROM Staff WHERE Location = "Chicago");.

Use the EXISTS predicate to determine whether the outer WHERE condition is true or false. If the inner query returns any value, the outer WHERE condition is true; otherwise, it is false. For example: SELECT Cust_no, Company FROM Customer WHERE EXISTS (SELECT * FROM Sales WHERE Customer.Cust_no = Sales.Cust_no);.

To create a correlated subquery:

In a correlated subquery, the outer query supplies the inner query with a value from each of its rows. The inner query is evaluated once for each of these values. If both outer query and inner query access the same table, assign aliases for the table. For example: SELECT Outer.Lastname, Outer.Location, Outer.Salary FROM Staff Outer WHERE Salary = (SELECT MAX(Salary) FROM Staff Inner WHERE Inner.Location = Outer.Location);.

Combining SQL and dBASE IV



Using SQL interactively prepares you for designing dBASE IV application programs that contain embedded SQL commands, discussed in Chapter 31. You can design your SQL database interactively and then test the operations that you plan to use in your application. dBASE IV provides a text editor, a debugger, and a compiler that you can use interactively to build your application.

This chapter shows you how to supplement SQL operations by using dBASE IV commands and functions interactively in SQL mode. You will learn how to:

- Enter dBASE commands and functions in SQL mode
- Use the SAVE TO TEMP clause of the SELECT command to monitor the operation of a complex join or subquery, or to save query results as database files
- Define dBASE database files as SQL tables
- Update SQL catalog table statistics as a database changes
- Import and export data to and from SQL tables
- Control access to SQL objects on a password-protected dBASE IV system

For complete information about any SQL command used in this manual, refer to Chapter 6 of *Language Reference*.

Preparing for This Chapter

Start dBASE IV:

1. Enter dbase at the operating system prompt.
2. If the Control Center is displayed, choose **Exit to dot prompt** from the **Exit** menu to access the dot prompt.
3. Enter SQL mode by entering:

```
. SET SQL ON
```

4. Start the Samples database:

```
SQL. START DATABASE Samples;
Database SAMPLES started
```

5. To make Samples the default directory, enter:

```
SQL. SET DIRECTORY TO \DBASE\SAMPLES
C:\DBASE\SAMPLES
```

6. Activate the Orders catalog file by entering:

```
SQL. SET CATALOG TO Orders
```

Using dBASE Commands and Functions in SQL Mode

You can use dBASE commands and functions to complement the operation of SQL commands. For example, you can:

- Use dBASE commands to process and format SQL results for printing reports and labels.
- Use SET commands to define the SQL environment.
- Use functions to influence how SQL commands are processed and how results are displayed.

Using SQL commands eliminates the need to use many dBASE commands and functions in SQL mode. However, you can still use redundant dBASE commands and functions in SQL mode to perform tasks the dBASE way.

You can use:

- SET CATALOG and SET TITLE to record and maintain information about SQL database files in dBASE catalog (.cat) files.
- BROWSE, EDIT or CHANGE, and APPEND (without the [BLANK] option) to display and update data in SQL tables. While using these commands, you can use SET INSTRUCT to control display of prompts and SET WINDOW to edit memo fields of non-SQL database files.
- ASSIST to display the Control Center and use its panels.
- CREATE/MODIFY APPLICATION, CREATE/MODIFY LABEL, CREATE/MODIFY QUERY/VIEW, CREATE/MODIFY REPORT, CREATE/MODIFY SCREEN, REPORT FORM, and LABEL FORM to initiate Control Center design and print functions.

- SET DESIGN to prevent an application user from accessing the dot prompt or one of the Control Center design screens.
- USE and SELECT to access SQL and non-SQL database files.
- CREATE/MODIFY STRUCTURE to define and update non-SQL database files.
- LIST/DISPLAY STRUCTURE to display the field definitions of SQL and non-SQL database files.
- PROTECT to implement a security scheme.
- SET to display and change the current values of SET commands.
- ALIAS() to return the alias name of a specified work area.
- SELECT() to return the number of the highest unused work area.

Unlike other dBASE functions, you cannot use ALIAS() or SELECT() within a SQL command.

Entering dBASE Commands

When entering a dBASE command at the SQL prompt or in the editing window, enter the command on one line *without* a semicolon terminator.

1. To change the way dates are displayed, enter:

```
SQL. SET DATE TO FRENCH
```

   When you retrieve date information, it will appear in French date format until you enter a new SET DATE command.

2. To route SQL commands and results to the printer, enter:

```
SQL. SET PRINTER ON
```

   Everything you type will now be echoed to the printer until you enter SET PRINTER OFF.

3. To record the SQL commands that you type and their results in a text file, enter:

```
SQL. SET ALTERNATE TO Session
SQL. SET ALTERNATE ON
```

   Everything you enter and display will now be recorded in a text file named Session.txt until you enter SET ALTERNATE OFF and then CLOSE ALTERNATE.

Using dBASE Functions

You can use dBASE functions in a SELECT command to transform result data and define search conditions.

Enter:

```
SELECT RTRIM(Firstname) + SPACE(1) + Lastname
  FROM Staff
  WHERE MONTH(Hiredate) = 6;

EXP1
Sandy Coudray
Mark Marin
```

The RTRIM() function in the SELECT clause removes trailing blanks following Firstname values in the result. The SPACE() function adds a single space between Firstname and Lastname. The + string operators concatenate Firstname, space, and Lastname.

The MONTH() function in the WHERE clause scans the number of the month in the Hiredate column of each salesperson row. If the number equals 6, the row is included in the result.

NOTE
In SQL mode you cannot abbreviate function names as you can in dBASE mode. For example, SQL does not recognize RTRI() or SPAC().

SAVE TO TEMP Clause

SAVE TO TEMP, used as the last clause in a SELECT query, is used to save a result in a temporary table so that you can use it during the current SQL session. Adding the KEEP keyword saves the result permanently as a database file in the current directory.

You can use SAVE TO TEMP...KEEP in the following ways:

- To monitor the operation of a complex join or subquery. Perform each part of the query and save the result for further study. Or, save the result of the full query performed on a smaller table.
- To save a result in a database file that can be processed by dBASE commands or defined as a SQL table using the DBDEFINE command, discussed later in this chapter.

Unless you specify KEEP, a temporary table is discarded as soon as you stop the current database or return to dBASE mode. If created by a SQL application, the temporary table is discarded when the program returns control to the SQL prompt or the dot prompt.

NOTE

Using SAVE TO TEMP adds an entry for the temporary table to an open dBASE IV catalog (.cat) file unless SET CATALOG is off. If you don't specify KEEP, the entry is removed from the catalog when the temporary table is discarded.

Now, use the SAVE TO TEMP clause to compile information about customer orders for use on invoices.

1. Enter:

```
SELECT DISTINCT Order_no, Items.Part_no, Descript, Qty, Unitcost, (Qty * Unitcost)
  FROM Items, Inventry
  WHERE Inventry.Part_no = Items.Part_no
  ORDER BY Items.Order_no
  SAVE TO TEMP Detail (Order_no, Part_no, Descript, Qty, Unitcost, Total);
```

   This command joins Items and Inventry tables to generate detail lines for each ordered item. The SAVE TO TEMP clause renames the calculated column and saves the result as a temporary table named Detail.

2. To verify the result, enter:

```
SELECT *
  FROM Detail;
```

   To complete the information needed to produce invoices, join the Detail, Sales, and Customer tables.

3. Enter:

```
SELECT DISTINCT Lastname, Sale_date, Detail.Order_no,
    Part_no, Descript, Qty, Unitcost, Total
  FROM Customer, Sales, Detail
  WHERE Sales.Cust_no = Customer.Cust_no
  AND Detail.Order_no = Sales.Order_no
  ORDER BY Detail.Order_no
  SAVE TO TEMP Invoice (Lastname, Inv_Date, Order_no,
    Part_no, Descript, Qty, Unitcost, Total) KEEP;
```

   This query saves invoice information in a database file named Invoice.

4. To verify that the information is complete, enter:

```
SELECT *
  FROM Invoice;
```

To create and print invoices using information from the Invoice file, use the dBASE IV report generator:

1. Enter USE Invoice to open the Invoice database file.
2. Press **F2 Assist** to display the Control Center.
3. If you want, use the **Reports** panel to specify the layout of the invoice report, adding sales tax, invoice price total, and any other desired detail.

Defining Database Files as SQL Tables

You may want to use database files created in dBASE as tables in a SQL database. Before you can use SQL commands to access database files as tables, you must use the DBDEFINE command to include information about the files in the SQL catalog tables.

When entered with a <filename> specification, DBDEFINE creates catalog table entries for the specified database file and its associated indexes (.mdx files only). Without a <filename> specification, DBDEFINE creates catalog entries for all undefined database files and associated indexes in the current database directory.

1. To define the sample file Vendors.dbf as a SQL table in the Samples database, enter:

```
DBDEFINE Vendors;
```

   The messages **Table(s) DBDEFINEd: VENDORS** and **DBDEFINE successful** appear.

   To view catalog information for the new Vendors table, query the Systabls catalog table. Systabls contains one row for each table in a database.

2. Enter:

```
SELECT Tbname, Tbtype, Colcount, Indxcount
  FROM Systabls
  WHERE Tbname = "VENDORS";

TBNAME  TBTYPE  COLCOUNT  INDXCOUNT
VENDORS T              12          1
```

NOTE
All table, index, and synonym names are stored in the system catalog tables in uppercase.

To display information about the columns of the Vendors table, query the Syscols table. Syscols contains information about each column of every table and view in the database.

3. Enter:

```
SELECT Colno, Colname, Coltype, Collen
  FROM Syscols
  WHERE Tbname = "VENDORS";

COLNO COLNAME    COLTYPE COLLEN
    1 VENDOR_ID  C            4
    2 VENDOR     C           30
    3 ADDRESS1   C           30
    .    .       .           .
    .    .       .           .
    .    .       .           .

   10 PHONE_EXT  C            4
   11 TERMS      C           10
   12 DISCOUNT   N            2
```

For a description of each of the SQL catalog tables, refer to Chapter 7 of *Language Reference.*

NOTE
If you've installed PROTECT for your dBASE IV system (refer to the SQL Security and Authorization section, later in this chapter), DBDEFINE encrypts the file using a SQL encryption key so that it is no longer accessible by dBASE commands.

Unencrypting Database Files

If PROTECT is installed, database files are encrypted by dBASE IV. Before using DBDEFINE on an encrypted database file, you must first unencrypt it:

- Enter SET ENCRYPTION OFF.
- Use the dBASE COPY command to create an unencrypted version of the file.

NOTE
You must have the privilege of accessing a file before you can unencrypt it.

Moving Database Files to a SQL Database

To move the database files for a dBASE application into a SQL database before executing an embedded SQL program (refer to Chapter 31), use the following procedure:

1. Create a SQL database for the application using the CREATE DATABASE command. (To add the files to an existing SQL database, skip this step.)
2. Use the dBASE COPY FILE command in SQL mode to copy database, memo, and index files into the database directory.

3. Activate the database (if it is not already started) using the START DATABASE command.
4. Enter the DBDEFINE <filename>; command to define individual dBASE database files as SQL tables. Enter the DBDEFINE; command (without a <filename> parameter) to define all dBASE database files as SQL tables.

Notes on DBDEFINE

You cannot use DBDEFINE to update SQL catalogs for:

- A dBASE view. Use the CREATE VIEW command to re-create a view once you have defined its base tables.
- Single-index (.ndx) files. Convert a single-index file to an .mdx tag before executing DBDEFINE.
- An index that was created in dBASE mode using the UNIQUE option of the INDEX command.
- An index that references more than 10 columns.

If you attempt to DBDEFINE a database file with memo fields whose memo (.dbt) files are not present in the database directory, you will receive a **File Open** error message. Before DBDEFINEing such a database file, you must do either of the following:

- Copy the memo file into the database directory.
- Enter the USE command for the database file to create an empty memo file.

DBDEFINE ignores the memo fields associated with a database file. Once you define the database file as a table, you cannot access its memo fields using SQL commands. Although a memo field remains with its database file, no catalog table entry is made for it.

If the multiple index files for a database file are present in the SQL database when the file is defined, DBDEFINE updates the catalog tables to reflect the statistics of the most recent REINDEX operation. To ensure that these statistics reflect the current state of the database file, enter the REINDEX command for the file before entering DBDEFINE, or enter the RUNSTATS command after using DBDEFINE.

Verifying SQL Catalog Entries

Use the DBCHECK command to verify catalog entries for SQL tables and indexes in the current database. DBCHECK compares the catalog definition of each table and index against the actual file structure. For every table whose .dbf or .mdx file structure does not match its catalog definition, DBCHECK displays an error message.

When DBCHECK displays an error message for a table, use the following procedure:

1. Use the dBASE COPY FILE command to copy the table's database and index files to another directory.
2. Use the DROP TABLE command to remove information for the table from the catalog tables.
3. Use the COPY FILE command to copy the database and index files for the table back to the database directory.
4. Enter the DBDEFINE command to redefine the database file as a SQL table.

Updating Catalog Statistics

When you create tables and indexes, statistics about these objects are recorded in the SQL catalog tables. Catalog table statistics help SQL determine the most efficient way of performing any operation on a table.

As you change the tables in a database, use the RUNSTATS command to update their statistics.

For example, enter:

```
RUNSTATS Staff;
```

to update the Staff table after using ALTER TABLE to change its structure, after changing more than 10 percent of its rows using INSERT, DELETE, or UPDATE, and after creating a new index for the table.

To update statistics for all tables and indexes in the Samples database, enter:

```
RUNSTATS;
```

without specifying a table name.

Importing and Exporting Data

Use the following commands to transfer data between SQL tables in the current database and external data files:

- LOAD DATA — imports data from an external file into a SQL table
- UNLOAD DATA — exports data from a SQL table to an external file

These commands support transfer to and from the same file types as the dBASE IV COPY commands:

- dBASE II database files (DBASEII)
- dBASE III®, dBASE III PLUS, and dBASE IV database files

- RapidFile database files (RPD)
- Framework II, Framework III, and Framework IV database and spreadsheet files (FW2, FW3, and FW4)
- Delimited format ASCII files (DELIMITED)
- System Data Format ASCII files (SDF)
- VisiCalc format files (DIF)
- MultiPlan spreadsheet format files (SYLK)
- Lotus 1-2-3 (version 1.0) format files (WKS)
- Lotus 1-2-3 (versions 1.A, 2.0, 2.1) format files (WK1)
- Lotus 1-2-3 (version 3.0) format files (WK2)

LOAD DATA and UNLOAD DATA can be executed in both interactive and embedded SQL modes. The external file can be in the current directory or in a different directory, specified by path.

For example, to append data from a RapidFile file to the Staff table, enter:

```
LOAD DATA FROM \RAPID\RFDATA\Salstaff.rpd
  INTO TABLE Staff
  TYPE RPD;
```

Note that Salstaff must have the same number of fields as Staff has columns, and that the data type of each Salstaff field must match that of its corresponding column. You can specify path information for both source and target files.

To export data from Staff to a dBASE database file in another directory, enter:

```
UNLOAD DATA TO C:\BACKUP\Staff2
  FROM TABLE Staff;
```

dBASE IV displays the message **12 row(s) inserted**. The fields in the new database file are created using the column names and definitions of the Staff table.

NOTE
If PROTECT is installed:

- *Data is automatically encrypted by SQL when it is loaded using LOAD DATA. (File data created using UNLOAD DATA is not encrypted.)*
- *You cannot use LOAD DATA to import data from a database file that has been encrypted by SQL or by dBASE IV — refer to the Data Encryption section later in this chapter.*

SQL Security and Authorization

Security refers to how SQL protects data from unauthorized access, modification, or destruction. dBASE IV provides security through a password-protected log-in system, assignment of access privileges to users, and encryption of database files.

Database protection in dBASE IV is set up by a system administrator using the dBASE PROTECT command. During installation of PROTECT, the administrator creates:

- Group names, user IDs (log-in names), and passwords.
- The SQLDBA (SQL Database Administrator) user ID, which controls all privileges for performing operations on SQL objects.

Once PROTECT is installed:

- A user must enter a password to *log in* and gain access to dBASE IV.
- In dBASE mode, the operations that the user can perform are dictated by privileges assigned by PROTECT.
- In SQL mode, user operations are controlled by the GRANT and REVOKE commands, not by PROTECT.

For complete information about the PROTECT command, refer to Chapter 2 of *Language Reference* and to Chapter 14 of *Using dBASE IV.*

GRANT and REVOKE

The owner of a table or view or the SQLDBA user ID uses the GRANT command to extend to other users privileges for the table or view. One of these privileges is the ability to grant the same privileges to other users (WITH GRANT OPTION), or to revoke the privileges.

GRANT privileges are cumulative; you can add to the privileges a user already has by granting further privileges. The following privileges can be granted:

- ALTER — ability to add columns to a table
- DELETE — ability to delete rows from a table or updatable view
- INDEX — ability to create indexes for a table
- INSERT — ability to add rows to a table or updatable view
- SELECT — ability to display rows from a table or view
- UPDATE [(<column list>)] — ability to update all columns or specific columns in a table or updatable view
- ALL — all privileges

The REVOKE command is used to remove any privilege granted. All privileges controlled using GRANT and REVOKE commands are recorded in the catalog tables of each SQL database.

The following guidelines apply to GRANTed privileges:

- Only the creator of a table, view, index, or synonym (or the SQLDBA user ID) has the privilege of dropping the object.
- You can't grant the ALTER, INDEX, or ALL privilege on a view.
- If you grant privileges using WITH GRANT OPTION, the grantee can grant the same privileges to other users. If you later revoke the privileges, they are likewise revoked from those to whom the grantee granted them.

For example, to grant SELECT, INSERT, UPDATE, and DELETE privileges on the Staff and Sales tables to users Janice and Greg and allow them to grant these privileges to other users, enter:

```
GRANT SELECT, INSERT, UPDATE, DELETE
  ON Staff, Sales TO Janice, Greg
  WITH GRANT OPTION;
```

If you later enter:

```
REVOKE ALL ON Staff, Sales
  FROM Janice, Greg;
```

the privileges are also revoked from any user to whom Janice and Greg granted them.

Data Encryption

Once PROTECT is installed, SQL table data is *encrypted* after a table is created. Data is stored on disk in an encoded form so that only privileged users can read it.

NOTE
To unencrypt SQL table data for use in dBASE mode, enter SQL mode and use the UNLOAD command or the SELECT command with the SAVE TO TEMP...KEEP option to copy table data to a database file. To unencrypt a database file for use in SQL mode, refer to the Unencrypting Database Files section earlier in this chapter.

Using SQL on a Network

On a network, where access to database files is shared, users may attempt to change the same information at the same time. If allowed to occur, such conflicts could be disastrous.

dBASE IV uses the following methods of resolving possible conflicts:

- Exclusive use of files
- File or record locking for the duration of an operation

Exclusive use of a file or file locking is used to control an entire file, for example, for updating multiple records. Record locking is used for random update of individual records.

In dBASE mode, you specify explicit or automatic locking of files and records using such commands and functions as SET EXCLUSIVE, SET LOCK, FLOCK(), and RLOCK() or LOCK(). In SQL mode, however, file and record locking is automatic. dBASE IV determines which type of lock you need by the operation you're performing on a table or view.

When dBASE IV is unable to complete a lock operation because a resource is already locked, use the SET REPROCESS command to retry the command until the lock is released.

For optimal access to data on a network:

- SET EXCLUSIVE to OFF (the default setting).
- Use the dBASE BEGIN/END TRANSACTION commands to ensure uninterrupted access during related update operations.

The next chapter illustrates how transaction processing is used in an embedded SQL application program.

Quitting dBASE IV

To exit from dBASE IV and return to the operating system, enter:

```
SQL. QUIT
```

Summary

This chapter showed you how to combine the features of SQL and dBASE IV. The major points are summarized below.

To enter a dBASE command in SQL mode:
Type the command on one line without a semicolon terminator and press ↵.

To use dBASE functions in SQL commands:
Use functions to transform result data and define search conditions. For example: SELECT RTRIM(Firstname) + SPACE(1) + Lastname FROM Staff WHERE MONTH(Hiredate) = 6;. Do not abbreviate function names.

To save a result:
Use the SAVE TO TEMP clause of the SELECT command to save a result as a temporary table for use during the current SQL session. Adding the KEEP keyword saves the temporary table as a dBASE database file at the close of the database session. You can use this file in dBASE mode or define it as a table using the DBDEFINE command. For example: SELECT DISTINCT Order_no, Items.Part_no, Descript, Unitcost FROM Items, Inventry WHERE Inventry.Part_no = Items.Part_no ORDER BY Items.Order no SAVE TO TEMP Detail (Order_no, Part_no, Descript, Unitcost) KEEP;.

To USE a database file or SQL table in SQL mode:
At the SQL prompt, enter the USE command to open the file. Press **F2 Assist** to display the Control Center or enter dBASE commands for the file.

To define a database file as a SQL table:
Use the DBDEFINE command to create SQL catalog table information for a specific database file and associated indexes, or for all undefined database files in the current directory. For example, to define Vendors.dbf as a table, enter: DBDEFINE Vendors;.

To view catalog table information for a table:
Query one of the SQL catalog tables (refer to Chapter 7 of *Language Reference*) using the SELECT command. For example: SELECT Tbname, Tbtype, Colcount, Indxcount FROM Systabls WHERE Tbname = "VENDORS";.

To unencrypt a database file:
To unencrypt a PROTECTed database file before DBDEFINEing it, enter SET ENCRYPTION OFF and use the COPY command to create an unencrypted version of the file.

To move database files to a SQL database:
Use the CREATE DATABASE command to create a database for the files or use an existing database. Use the dBASE COPY FILE command in SQL mode to copy database, memo, and index files into the database directory. Use the START DATABASE command to start the database (if it is not already started). Enter DBDEFINE to define the database files as SQL tables.

To verify catalog entries for database files:

Use the DBCHECK command to compare catalog definitions of tables and indexes with their file structures.

To update catalog statistics:

Use the RUNSTATS <table name> command to update statistics for an individual table; use RUNSTATS without a parameter to update statistics for all tables in the current database.

To import and export data:

Use the LOAD DATA command to import data from an external file to a SQL table. Use the UNLOAD DATA command to export data from a SQL table to an external file.

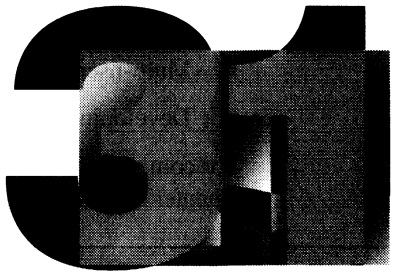
To control privileges on database objects:

If PROTECT is installed, SQL tables are encrypted after creation so that only authorized users can use their data. Use the GRANT and REVOKE commands to extend or remove user privileges for using tables and views.

To maximize use of SQL data on a network:

SET EXCLUSIVE to OFF (the default). Use the dBASE BEGIN/END TRANSACTION commands to ensure uninterrupted access to data during related update operations.

Embedding SQL Commands



In a mainframe or minicomputer environment, SQL is used to write database applications with host programming languages such as COBOL, FORTRAN, or C. With dBASE IV, you use SQL with dBASE IV as the programming language.

dBASE IV provides the following tools for building an application:

- Commands and functions for creating menus and data entry forms, and for printing reports
- Program memory variables
- Constructs such as DO WHILE, DO CASE, IF...ELSE, and ON ERROR, to control program processing and flow
- Commands for processing keyboard input and for setting up the environment, so that, for example, you can customize options for displaying data

In a dBASE IV SQL program, SQL commands are *embedded* in the dBASE application program. You use the same SQL commands in a program that you use at the SQL prompt, plus other SQL commands that are used exclusively for programs.

This chapter guides you in:

- Using SQL program files (which contain embedded SQL commands) with dBASE program files (which contain only dBASE commands) in an application
- Using dBASE functions and user-defined functions in SQL program files
- Using system memory variables to determine program status for error handling
- Controlling use of dBASE IV work areas
- Creating, compiling, and executing programs
- Embedding data definition commands
- Embedding SELECT commands to display data, transfer data from one or more result rows into SQL memory variables, or update or delete selected rows using cursor commands

- Embedding INSERT, UPDATE, and DELETE commands
- Multi-user programming and transaction processing
- Developing SQL applications

For complete information about any SQL command used in this manual, refer to Chapter 6 of *Language Reference*.

Preparing for This Chapter

The purpose of this chapter is to introduce you to programming with SQL. You are shown examples of dBASE programming code without being asked to write a program or enter commands.

If you are familiar with writing applications in the dBASE language, you may want to use the dBASE IV internal editor to experiment with writing simple SQL programs.

Embedding SQL Commands

In a dBASE IV application, you can use pure dBASE commands or you can combine dBASE and SQL commands. However, because of the differences in the way the two languages access data, you need to distinguish between program files that use SQL for data access and those that use dBASE IV.

dBASE and SQL Program Files

Program files that use embedded SQL commands for accessing data must have a .prs file extension. Program files that use dBASE commands for access must have a .prg file extension.

You can use both types of program files in the same application. A .prg file can call a .prs file, and vice versa, using the DO command. Thus, you can use the best data access method for any programming situation.

The following restrictions apply to using .prg and .prs files:

- You cannot include SQL commands in program files that use dBASE commands to access database files.
- In program files that use SQL, you cannot use dBASE commands and functions that cannot be used in SQL mode. (Refer to Appendix H of *Language Reference* for a listing of the dBASE commands and functions that you can use with SQL.)
- In a .prg file, the semicolon is used only as a *continuation* character, to end a command line containing a dBASE command that is continued on the next line. In a .prs file, the semicolon is used both as a terminator for each SQL command *and* as a continuation character.

NOTE

*Be careful when using the dBASE NOTE command to exclude a SQL command from being executed (that is, to "comment out" the command line by preceding it with NOTE, *, or &&). dBASE IV may interpret the commented SQL command's semicolon terminator as continuing the NOTE command on the next line, and the next line of code may be ignored.*

dBASE Memory Variables

You can use dBASE memory variables in SQL program files for:

- Specifying WHERE clause search conditions
- Transferring data to table columns using the INSERT command
- Receiving column values conveyed using the FETCH command
- Receiving row values conveyed using the SELECT INTO command
- Passing data between dBASE and SQL program files

User-Defined Functions

You can use a dBASE UDF (user-defined function) in a SQL program file, with the following restrictions:

- A UDF can contain only the dBASE IV commands and functions that are available in SQL mode. A UDF cannot contain any SQL command.
- A UDF cannot be called from within a SQL command.

A UDF in a .prs file can call a UDF in a .prg file, and vice versa, just as .prs and .prg files can call one another.

SQL Status and Error Handling

You can use the dBASE ON ERROR command to create specific error-handling routines that are triggered by the error numbers your SQL program returns.

You can use the *Sqlcode* and *Sqlcnt* memory variables to signal the status of each SQL operation as it is performed. Sqlcode indicates whether the operation succeeded, failed, or returned no data. Sqlcnt indicates the number of rows affected by the operation.

Table 31-1 summarizes Sqlcode and Sqlcnt values for SQL data definition commands. These commands include: ALTER TABLE; CREATE or DROP DATABASE, INDEX, TABLE, or VIEW; SHOW, STOP, or START DATABASE; RUNSTATS; CLOSE and DECLARE CURSOR. (For a complete listing, see Table 31-4.)

Table 31-1 Data definition Sqlcode and Sqlcnt values

Status	Sqlcode	Sqlcnt
Succeed	0	0
Fail	–1	0

Table 31-2 lists Sqlcode and Sqlcnt values for SQL data retrieval and data manipulation commands. These commands include SELECT, FETCH, OPEN CURSOR, INSERT, UPDATE, and DELETE.

Table 31-2 Data retrieval or manipulation Sqlcode and Sqlcnt values

Status	Sqlcode	Sqlcnt
Succeed	0	Number of rows affected (for SELECT INTO, Sqlcnt = 1)
Fail	–1	0
No data affected; FETCH fetched no rows	100	0

NOTE
For Fail, the value of Sqlcode may be the error number.

Table 31-3 summarizes Sqlcode and Sqlcnt values for SQL cursor commands. Cursor commands include UPDATE WHERE CURRENT OF and DELETE WHERE CURRENT OF.

Table 31-3 Cursor control Sqlcode and Sqlcnt values

Status	Sqlcode	Sqlcnt
Succeed	0	1
Fail	–1	0

You can use the value of Sqlcode to determine whether a FETCH command returned any rows that need to be processed. You can use the Sqlcnt value following an UPDATE operation to determine whether you should use the RUNSTATS command to update catalog statistics for a table.

dBASE IV Work Areas

dBASE IV can use a total of 40 work areas. When a program switches to SQL mode to execute a .prs file, any work areas that were being used in dBASE mode remain open. To minimize the possibility of running out of work areas, your program should close unused open files before switching to SQL mode.

SQL requires one work area for:

- Each database table or catalog table referenced in a SELECT query, subquery, or self-join.
- Each open cursor used by a program (refer to the Transferring Multiple Rows section later in this chapter). Work areas used by an open cursor remain open until the cursor is closed, even if the program switches to dBASE mode.
- Each SAVE TO TEMP clause.

NOTE
A database file is opened during execution of a SQL command, and is closed after execution.

Creating and Running SQL Programs

You create an embedded SQL program in one or more ASCII text files. The dBASE and embedded SQL program files are stored separately from the database files and tables referenced by the program.

Creating Program Files

Use the dBASE internal editor to create program files in dBASE IV. (Refer to Chapter 15 of *Using dBASE IV* for a discussion of the internal editor.)

NOTE
You can use a different editor in place of the dBASE internal editor by including a TEDIT configuration command in your Config.db file. (Refer to Chapter 2 of the Getting Started with dBASE IV *manual.)*

The same programming techniques that you use to create dBASE program files also apply to designing SQL program files. Embedded SQL programs have the following similarities to dBASE programs:

- They have the same structure.
- They use nearly the same commands and functions.
- They define and use memory variables in the same way.
- They use similar methods of error handling and transaction processing.
- They operate the same way on networks.

Compiling and Executing Programs

Once you've created your application program files, you can run your application using the dBASE DO command. DO compiles the program files, loads the specified main program into memory, and begins executing its commands.

During execution of an application program, dBASE program files call SQL program files, and vice versa. Depending on the file extension it encounters, dBASE IV automatically switches between dBASE and SQL mode. When switching modes, dBASE IV restores the environment (active SQL database, work areas, and open files) previously established for that mode.

Using Program Procedures

Both SQL and dBASE program files can contain named program procedures. If you don't include a file extension when executing a named procedure, dBASE IV searches as follows:

- First, for a procedure within the currently executing program file
- Second, for a procedure within any other active program file
- Finally, for a dBASE (.prg) or SQL (.prs) program file

Using dBASE Programming Facilities

Your application can take advantage of the following dBASE IV facilities:

- SET TRAP and DEBUG — to handle execution errors
- COMPILE — to compile a program into *object* code without executing it (as DO does)
- DBLINK — to link program object modules into a single file

- RunTime program — to package a run-time version of dBASE IV with your application object files
- BUILD — to create an executable version of your application and copy the files to a floppy disk

Embedding Data Definition Commands

It is not a good practice to use data definition commands in a SQL program to define tables, views, synonyms, and indexes that are used by the program. Never execute data definition commands conditionally, using the IF...ELSE program construct.

Instead, define objects before program execution, either interactively or in a .prs file that is run before your program is executed. Once objects are defined, you can use data retrieval and data manipulation commands in your program to query and update them.

Table 31-4 lists the data definition commands.

Table 31-4 Data definition commands

Command	Description
ALTER TABLE	Add columns to a table
CREATE DATABASE	Create a database to hold tables and related files
CREATE INDEX	Create an index based on the columns of a table
CREATE SYNONYM	Create an alternate name for a table or view
CREATE TABLE	Create a base table
CREATE VIEW	Create a view based on one or more tables or views
DBDEFINE	Create a SQL table from an existing database file
DROP DATABASE	Remove an existing database
DROP INDEX	Remove an index
DROP SYNONYM	Remove a synonym
DROP TABLE	Remove a table and any views or synonyms based on the table
DROP VIEW	Remove a view
RUNSTATS	Update catalog statistics for a single table or all tables in the current database

The syntax of any of these commands in an embedded SQL program is the same as in interactive mode. Refer to Chapter 6 of *Language Reference*.

If PROTECT is installed, after defining objects you can use the authorization commands, GRANT and REVOKE, to define privileges on the objects.

NOTE
If you plan on using SQL data definition commands in programs, refer to the Developing SQL Applications section later in this chapter for restrictions on their use.

Embedding SELECT Commands

The most versatile command in embedded SQL is SELECT. An embedded SELECT command can:

- Display data from a table or view
- Transfer values from a single row to dBASE memory variables
- Transfer values from selected rows to dBASE memory variables, one row at a time, using SQL cursor commands
- Update or delete selected rows using SQL cursor commands

Embedding SELECT to Display Data

The following program example shows how to use an embedded SELECT command to display data on the screen:

```
DO CASE
      *
      CASE mchoice = 1
   *
   WAIT "Press any key to display the employee with highest salary"
   *
   SELECT Lastname, Salary
   FROM Staff
   WHERE Salary =
      (SELECT MAX(Salary)
      FROM Staff);
      CASE mchoice = 2
         .
         .
         .
ENDCASE
```

The user is prompted to press any key on the terminal keyboard. When a key is pressed, the embedded SELECT command displays the last name and salary of the salesperson with the highest salary.

The following embedded SELECT displays rows matching the value stored in a memory variable:

```
mlocation = "NY"
    .
    .
    .
  SELECT *
  FROM Customer
  WHERE State = mlocation
  ORDER BY Company;
```

Transferring a Single Row

The SELECT command uses the INTO clause to transfer values from a single row into corresponding dBASE memory variables. If more than one row is selected, only the first row's values are stored.

The following program segment prompts a user to enter a customer name, then retrieves information for the name entered into memory variables:

```
ACCEPT "Enter customer's last name: " TO mcustomer
  SELECT Firstname, Lastname, Address, City, State, Zip
  INTO mfname, mlname, maddr, mcity, mstate, mzip
  FROM Customer
  WHERE UPPER(Lastname) = UPPER(mcustomer);
  *
  IF Sqlcnt > 0
  DO Subprg1 WITH mfname, mlname, maddr, mcity, mstate, mzip
```

The ACCEPT command stores the customer name in a memory variable named *mcustomer*. The new value of *mcustomer* is then used in the WHERE clause of the SELECT command to locate the Customer row whose Lastname value equals the *mcustomer* value. The information in that row is stored in the command's INTO memory variables.

NOTE
If a SELECT...INTO does not return any rows, memory variables named in the INTO clause are not created. Therefore, before processing INTO memory variables, you may want to initialize their values or check the value of Sqlcnt as shown in the example.

Transferring Multiple Rows

When an embedded SELECT command uses memory variables to return data from a number of rows, it employs a *cursor*. The cursor is used to transfer data to the program one row at a time.

A cursor is similar to a dBASE record pointer. However, the cursor points to each row in the SELECT result table, one row at a time. Also, the SQL cursor can only move in a forward direction.

The following SQL commands are used for cursors:

- DECLARE CURSOR — defines the cursor and an embedded SELECT command that returns the result rows to which the cursor will point.
- OPEN — executes the cursor's SELECT command and positions the cursor just before the first result row. If the SELECT returns no rows, the cursor is positioned at the end of the file.
- FETCH — advances the cursor to each result row, in turn, and transfers row values into corresponding program memory variables.
- CLOSE — closes the cursor and releases its work area and the memory that it uses. If you reopen the cursor, its SELECT command is re-executed to produce a new result table.

In a SQL program, a cursor normally operates in the following way:

1. The program DECLAREs the cursor.
2. The program OPENs the cursor, executes its SELECT command to produce a result table, and positions the cursor just before the first result row.
3. The program encounters the FETCH command, which positions the cursor to the first result row, creates the corresponding memory variables, and transfers row data into the variables.
4. The program loops back to FETCH, which positions the cursor to the next result row and transfers its values into the memory variables.
5. The process in step 4 is repeated until there are no more result rows to process.
6. The program CLOSEs the cursor.

NOTE
In a program, you cannot execute a DECLARE CURSOR command conditionally. A CLOSE command must be placed after any command that references the cursor or an error will occur during compilation.

Once opened, a cursor remains open until the program closes it, leaves SQL mode, or finishes executing. Thus, a cursor remains open during execution of lower-level subroutines, whether they are dBASE program files or SQL program files.

A cursor created in one SQL program file cannot be referenced in another SQL program file. If a cursor created by one procedure is referenced by other procedures in the same program file, the creating procedure must occur before the other procedures.

You could use a cursor to identify uninvoiced orders and transfer information about each order to an invoicing subprogram. For example:

```
DECLARE Inv CURSOR FOR
     SELECT Order_no, Sale_date, Staff_no, Cust_no
     FROM Sales
     WHERE NOT Invoiced;
        .
        .
        .
     OPEN Inv;
        .
        .
        .
     DO WHILE .T.
        FETCH Inv INTO morder_no, msale_date, mstaff_no, mcust_no;
        IF SQLCODE = 0
                .
                .
                .
           DO Invoice WITH morder_no, msale_date, mstaff_no, mcust_no
                .
                .
                .
        ELSE
          EXIT
        ENDIF
     ENDDO
     CLOSE Inv;
```

The same principles that apply to using cursors to transfer row data also apply to using cursors to update and delete row data, discussed below.

Embedding UPDATE Commands

You can use the UPDATE command in a program to change selected row values in a table or view. You can use the interactive form of UPDATE to update selected rows or you can use UPDATE with cursor commands.

Updating Selected Rows

The following program code could be used to update the Commission column of the Staff table for salespeople located in New York:

```
ACCEPT "Enter percent commission increase" TO mcomm
    UPDATE Staff
    SET Commission = Commission + mcomm
    WHERE State = "NY";
```

NOTE
You can use a dBASE memory variable in an UPDATE command wherever you can use a constant. In the previous WHERE clause, for example, you could update each row containing a value that matched the value of a memory variable.

Using a Cursor to Update

Use the WHERE CURRENT OF form of the UPDATE command to update the table rows that correspond to the result rows pointed to by a cursor. For example:

```
DECLARE Inv CURSOR FOR
   SELECT Order_no, Sale_date, Staff_no, Cust_no, Invoiced
   FROM Sales
   WHERE NOT Invoiced
   FOR UPDATE OF Invoiced;
      .
      .
      .
   OPEN Inv;
   DO WHILE .T.
       FETCH Inv INTO morder_no, msale_date, mstaff_no, mcust_no, minvoiced;
       IF SQLCODE = 0
          * Print an invoice for order number in the current row
          * If successful, change minvoiced memory variable to .T.
          *
          DO Invoices WITH morder_no, msale_date, mstaff_no, mcust_no, minvoiced
          *
          IF minvoiced
             UPDATE Sales
             SET Invoiced = .T.
             WHERE CURRENT OF Inv;
          *
          ENDIF
       ELSE
         EXIT
       ENDIF
  ENDDO
  CLOSE Inv;
```

The Invoice program generates an invoice for the order row pointed to by the cursor and updates the Invoiced column in the corresponding Sales table row to true (.T.). *WHERE CURRENT OF Inv* identifies the Sales table row that is to be updated.

Embedding DELETE Commands

Use the DELETE command in a program to delete selected rows of a table or view. As with UPDATE, you can use the interactive form of DELETE to delete selected rows or you can use DELETE with a SQL cursor.

Deleting Selected Rows

You could use the following command in a program to delete all rows for invoiced orders that were entered before September 22, 1987:

```
DELETE
     FROM Sales
     WHERE Invoiced AND Sale_date < {09/22/87};
```

NOTE

You can use a dBASE memory variable in a DELETE command wherever you can use a constant. In the previous WHERE clause, for example, you could delete each row containing a value that matched the value of a memory variable.

Using a Cursor to Delete

Use the WHERE CURRENT OF form of the DELETE command to delete the table row that corresponds to a result row pointed to by a cursor. For example:

```
DECLARE Inv CURSOR FOR
   SELECT Order_no, Sale_date, Staff_no, Cust_no, Invoiced
   FROM Sales
   WHERE NOT Invoiced;
      .
      .
      .
   OPEN Inv;
      .
   DO WHILE .T.
      FETCH Inv INTO morder_no, msale_date, mstaff_no, mcust_no, minvoiced;
      IF SQLCODE = 0
         * Print an invoice for order number in the current row
         * If successful, delete invoiced from Sales
         *
         DO Invoices WITH morder_no, msale_date, mstaff_no, mcust_no, minvoiced
         *
         IF minvoiced .AND. Sale_date < {09/22/87}
            DELETE FROM Sales
            WHERE CURRENT OF Inv;
         ENDIF
      ELSE
        EXIT
      ENDIF
   ENDDO
   CLOSE Inv;
```

After the Invoice program has invoiced the order for the result row pointed to by the cursor, the corresponding row in Sales is deleted if its order was entered before September 22, 1987.

NOTE

Rows deleted using a cursor are not removed from the table until the cursor is closed. If SET DELETED is OFF, subsequent SQL query results include deleted rows, prefixed by an asterisk. If SET DELETED is ON, deleted rows are not displayed in results, and do not affect SQL operations.

Embedding INSERT Commands

Use the INSERT command in a SQL program to add rows to a SQL table. The following program segment employs a dBASE data entry form for inserting new rows in the Inventry table:

```
DO WHILE .T.
      *
      @ 3,20 SAY "Add items to Inventry table"
      @ 5,8 SAY "Enter 5-digit part number" GET mpart_no
      @ 6,8 SAY "Enter a part description" GET mdescript
      @ 7,8 SAY "Enter the quantity on hand" GET mon_hand
      @ 8,8 SAY "Enter the stock location" GET mlocation
      @ 9,8 SAY "Enter the part's unit cost" GET munitcost
      @ 10,8 SAY "Enter 0 in the part number field to exit"
      *
      READ
      *
      IF mpart_no = 0
        EXIT
      ENDIF
      SELECT COUNT(*)
      INTO l_match
      FROM Inventry
      WHERE Location = mlocation AND Part_no = mpart_no;
      *
      *  Add new row if location is new for existing part
      *
      IF l_match = 0
         INSERT INTO Inventry
         (Part_no, Descript, On_hand, Location, Unitcost)
         VALUES (mpart_no, mdescript, mon_hand, mlocation, munitcost);
         *
         * Part number and location are not unique
      ENDIF
ENDDO
```

An operator types information for items that are received into inventory at each stocking location. A new row is added if the part number or location entered is new. If not, the program loops back to display the entry form without entering a row.

Multi-User and Transaction Programming

When you share dBASE IV with other users on a network, you may sometimes have to compete for use of tables. For this reason, a table is automatically locked when any user is inserting, updating, or deleting data.

If you encounter a lock while modifying data interactively, you can wait until the lock is released, or use the SET REPROCESS command to have dBASE IV retry your command. However, a lock encountered by an embedded SQL program that modifies data could result in incomplete program execution and inconsistent data.

To prevent this, use the dBASE IV BEGIN TRANSACTION and END TRANSACTION commands to define each data modification operation in a program as a transaction. You can then specify a procedure to handle an interruption in transaction processing caused by a lock or other problem.

If a transaction includes a number of interdependent data modification operations, include the ROLLBACK command to undo any changes made to the database if the transaction cannot be completed successfully.

The following example shows how to set up a transaction:

```
ON ERROR DO Recover
   SET REPROCESS TO 15
   BEGIN TRANSACTION
       UPDATE Staff
       SET Commission = Commission + mcomm
       WHERE State = "NY";
   END TRANSACTION
   ON ERROR
   IF COMPLETED()
      @ 21,15 SAY "Transaction successfully completed"
   ENDIF
```

The UPDATE command is defined as a transaction. If a problem occurs during update processing, the ON ERROR command executes the following recovery procedure:

```
PROCEDURE Recover
      @ 21,15 SAY "Your transaction has encountered an error condition"
      @ 22,15 SAY "Do you want to RETRY? (Y/N)" GET choice PICTURE "!"
      READ
      @ 21,15 TO 22,65 CLEAR
      IF choice = "Y"
        RETRY
      ELSE
        @ 21,15 SAY "Rolling back your transaction. Please wait."
        ROLLBACK;
      ENDIF
RETURN
```

The Recover procedure lets the user retry the update or discontinue it, rolling back any update made to the table before the error occurred.

NOTE

- *The following SQL commands are not allowed in transactions: data definition commands (see Table 31-4); startup commands (START DATABASE and STOP DATABASE); utility commands (DBCHECK, DBDEFINE, LOAD DATA, UNLOAD DATA, RUNSTATS); security commands (GRANT and REVOKE).*
- *Rows deleted by a DELETE command defined as a transaction are not removed from the table until END TRANSACTION is executed. Rows deleted by a DELETE...WHERE CURRENT OF transaction are not removed until the end of the transaction in which the associated cursor is closed. If the cursor is closed after the transaction is ended, rows are not removed until the cursor is closed.*
- *Rows deleted during a transaction are not visible to users, as if SET DELETED ON were in effect.*

Developing SQL Applications

For the sake of performance and data security, observe the following rules when creating a SQL application:

- Create database objects before referencing them.
- Avoid ambiguity in specifying the current database.
- Recompile an application periodically to reflect changes in the database.

Create Objects Before Referencing Them

Unlike dBASE programs, an embedded SQL program requires either

- That all database objects referenced by the program exist when the program is compiled; or,
- That objects be defined before they are referenced.

Therefore, a CREATE, DBDEFINE, or SELECT...SAVE TO TEMP command for an object must precede any command that addresses the object, for example, DELETE, DECLARE CURSOR, GRANT, INSERT, REVOKE, SELECT, or UPDATE. (Refer to the Embedding Data Definition Commands section earlier in this chapter.)

To ensure that data definition precedes reference, do one of the following:

- Create a separate initialization program file that is executed before the application program file.
- Place data definition commands at the beginning of a program file.

Avoid Ambiguity in Specifying the Current Database

In interactive or embedded SQL, you use the START DATABASE command to activate a database containing the tables and views that will be referenced by subsequent commands. This database remains the current database until you close it using STOP DATABASE, start another database using START DATABASE, or create a new database using CREATE DATABASE.

Consider the following SQL commands:

```
IF <condition>
        START DATABASE Mktg;
ELSE
        START DATABASE Sales;
ENDIF
UPDATE Staff SET Salary = 8000;
```

Because the dBASE IV compiler cannot evaluate the IF condition, the UPDATE command always addresses the Staff table in the Sales database because Sales is the current database when the compiler encounters the UPDATE command.

You can restructure this segment so that the Staff table chosen by the IF condition is updated:

```
IF <condition>
     START DATABASE Mktg;
     *
     UPDATE Staff SET Salary = 8000;
     *
ELSE
     *
     START DATABASE Sales;
     *
     UPDATE Staff SET Salary = 8000;
     *
ENDIF
```

Recompile an Application Periodically

During compile, dBASE IV determines the best methods of performing queries using information in the SQL catalog tables. These methods of query *optimization* may no longer be valid if you have changed the database since your application was last compiled.

Recompile an application when:

- The data that the application uses has changed significantly.
- You have dropped indexes that were used for optimization.
- You have created new indexes that weren't previously available for optimization.

Summary

This chapter provided guidelines for creating embedded SQL applications, programs written in dBASE IV that use SQL commands for data access. This section summarizes these guidelines.

To create SQL program modules:

Use the .prs extension to name application modules that use only SQL commands to access data defined as SQL tables. Use the .prg extension to name files containing pure dBASE code.

In a .prg file, use the semicolon (;) at the end of a line to continue coding on the next line; in a .prs file, use the semicolon both as a continuation character for dBASE commands and as a terminator for SQL commands.

To use memory variables in SQL programs:

Use dBASE memory variables in .prs files to specify WHERE search expressions, to transfer data to table columns using INSERT, to receive column values conveyed by FETCH, to receive row values conveyed using SELECT INTO, and to pass data between .prs and .prg files.

To use UDFs in SQL program files:

In user-defined functions (UDFs), use only SQL commands and dBASE commands and functions allowed in SQL mode. Do not call a UDF from within a SQL command. A UDF in a .prs file can call a UDF in a .prg file, and vice versa.

To monitor SQL status:

Use the Sqlcode memory variable with an ON ERROR routine to monitor whether an operation succeeded, failed, or returned no data. For example, you can use Sqlcode to determine whether a FETCH command returned any rows that need to be processed.

Use the Sqlcnt memory variable to determine the number of rows affected by an operation. For example, following an UPDATE, use Sqlcnt to decide whether to refresh catalog statistics for a table using the RUNSTATS command.

To conserve work areas:

Close unnecessary open files before switching to SQL mode.

To create SQL program files:

Use the dBASE internal editor to create ASCII text files for SQL programs. Use the same programming techniques and dBASE facilities that you use for creating pure dBASE program files. Check Appendix H of *Language Reference* to determine which dBASE commands and functions you can use in your SQL program.

Embedding data definition commands:

It is preferable not to embed data definition commands in a program, but to use these commands to define objects before program execution. Enter the commands interactively or in an initialization program that is run before program execution. If you must define objects in your program, define objects before referencing them. Never define an object within an IF...ELSE condition.

Embedding SELECT commands:

Use an embedded SELECT command to display data from a table or view; to transfer values from a single row to dBASE memory variables, or from selected rows to dBASE memory variables; and to update or delete rows.

Use the SQL cursor commands to transfer values from a number of rows, one row at a time, to memory variables: DECLARE CURSOR, to define the SELECT command that returns the result rows to which the cursor will point; OPEN, to execute the SELECT command and position the cursor before the first result row; FETCH, to advance the cursor to each result row, in turn, and transfer row values into corresponding program memory variables; CLOSE, to close the cursor.

Embedding UPDATE commands:

To change selected row values in a table or view, use the interactive form of UPDATE; or use the WHERE CURRENT OF form of the command to update the table row that corresponds to the result row pointed to by a cursor. For example: UPDATE Sales SET Invoiced = .T. WHERE CURRENT OF Inv;.

Embedding DELETE commands:

To delete selected rows of a table or view, use the interactive form of DELETE; or use the WHERE CURRENT OF form of the command to delete the table row that corresponds to the result row pointed to by a cursor. For example: DELETE FROM Sales WHERE CURRENT OF Inv;.

Embedding INSERT commands:

Use the interactive form of the INSERT command to add rows to a table or view.

To control multi-user access to data:

On a network, SQL tables are automatically locked while users are inserting, updating, or deleting data. A SQL program that modifies data can cause inconsistencies in a database if it cannot complete successfully because of a lock or some other problem. Therefore, use the dBASE BEGIN TRANSACTION and END TRANSACTION commands to define each data modification operation as a transaction. Use the ROLLBACK command in a recovery procedure to undo any change made during an incomplete transaction.

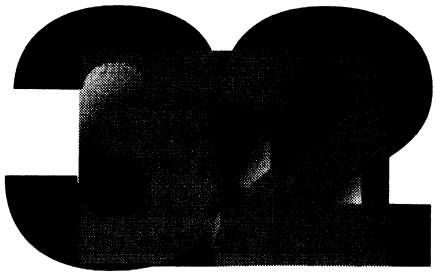
Rules for developing SQL applications:

Create database objects before referencing them. Do not define any object conditionally. Avoid ambiguity when specifying the current database. Recompile an application periodically to reflect changes in the database.

Optimizing Performance

Optimizing Your System
Optimizing dBASE IV
Optimizing Your dBASE Applications

Optimizing Your System



This chapter discusses several ways you can optimize your system to improve dBASE's performance. You can apply the methods explained here to any system configuration.

Of course, you can also improve your system through hardware enhancements. Adding more memory and a larger-capacity and faster hard drive, for example, will enhance dBASE's performance.

What This Chapter Covers

Topics in this chapter tell how you can achieve optimal dBASE performance by:

- Reorganizing files and directories
- Increasing disk space
- Tuning your disk cache utility

Reorganizing Files and Directories

One of the easiest ways to improve dBASE's performance is to keep the number of files in your working directories small. As the number of files in a directory increases, dBASE appears to be slower. The problem, however, lies in the relatively slow speed of DOS directory searches.

You can reduce the number of files in your directories by:

- Using the DBLINK utility to combine dBASE IV compiled objects into a single file. For more information about DBLINK, read Chapter 15, "Compiling, Debugging, and Linking."
- Distributing your application files among several directories.

Placement of Temporary Files

dBASE creates its temporary files in the current working directory unless you designate an alternate location. These temporary files are automatically created, for example, when you create or modify the structure of files in dBASE, when you perform sorting or indexing operations, or when you access the Control Center.

You can reduce the number of files in your working directory and improve dBASE's performance by directing all dBASE temporary files to a dedicated directory. If you have sufficient memory above 1MB, direct all dBASE temporary files to a RAM disk. To do so, use the DBTMP DOS environmental variable. The following shows how you use SET DBTMP in your Autoexec.bat:

```
SET DBTMP=<drive>:\<directory>
```

Using a RAM Disk

If you are deciding whether to use a RAM disk, assess the tradeoff of dedicating valuable extended memory to the RAM disk versus allocating that memory to the dBASE buffer manager. The RAM disk is defined in the DOS boot configuration, so any extended memory you allocate to it is permanently allocated.

Increasing Disk Space

In any processing environment, disk input/output performance degrades significantly when a disk drive is nearly full. To increase disk space, you can:

- Remove temporary files with the .$DB extension. Do not delete these files while dBASE is running as dBASE might be using them. If you used the DBTMP environmental variable, these temporary files are in one directory.
- Remove any unneeded backup files including files with the extensions .BAK, .DBK, .MBK, and .TBK. These are backup files; you might want to save them on floppy disks.
- Reduce wasted disk space by running a disk optimization utility periodically to defragment your hard disk.

NOTE
To run dBASE, you must always have at least 1MB of available disk space. If your system runs out of physical or logical (RAM disk) disk space, dBASE stops all processing, displays a disk full error message, and asks you to delete files or cancel the current operation.

Tuning Your Disk Cache Utility

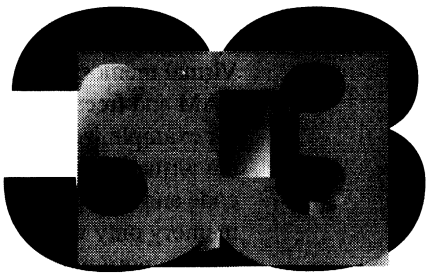
dBASE IV version 2.0 has an improved buffer manager that incorporates index and data-caching algorithms designed for database management. dBASE's internal cache usually performs better than an external cache, which is unaware of dBASE's higher-level file structures. dBASE uses extended memory (XMS) for both program and data storage.

When you're deciding whether to use an external cache (such as HyperDisk, SMARTDRV, PC Quick, Norton Cache, etc.), consider these points: the more extended memory you allocate to an external cache, the less there is for dBASE's buffer caching. Also, an external cache uses CPU cycles for every disk I/O operation, and causes CPU mode switches from protected mode to real mode and vice versa.

The advantage of an external cache is that it speeds up DOS directory accesses. This advantage might offset the performance penalties arising from the cache's own memory and CPU usage.

If you decide to use an external cache, consider allocating to it a small amount of memory, such as 128K. This approach leaves the maximum amount of memory available for dBASE's internal buffer manager.

Optimizing dBASE IV



This chapter discusses how you can optimize dBASE IV for better performance.

What This Chapter Covers

This chapter covers the following topics:

- Managing dBASE memory
- Allocating memory for caching index blocks
- Setting block sizes for memo field files and index files
- Optimizing dBASE's environmental settings

Managing dBASE Memory

When you start dBASE IV, it reserves memory for its system files, and allocates remaining memory to data buffers, index caching, and your dBASE applications.

dBASE's memory management system is fully dynamic: it evaluates your system's memory configuration and automatically chooses the settings that normally give the best performance. You will find that dBASE's default settings work well. To give you the flexibility to further tune the dBASE environment, dBASE IV also provides advanced users and developers with several new configuration settings.

NOTE
How you allocate memory to the dBASE subsystems affects the performance of dBASE and your applications. Before you change any of the default settings, you should thoroughly understand dBASE's memory management systems and their configuration parameters.

The following sections describe how dBASE's Virtual Memory Manager (VMM) and data buffering system manage memory allocation, and how you can change the default settings.

Virtual Memory Management

Virtual memory is a memory management technique that uses disk space to augment RAM and frees your program from your computer's physical memory limitations. If, for example, your computer has only 1MB of RAM, but your program requires 4MB, the virtual memory manager creates a swap file on your hard disk to store software code and data segments. The code and data segments are then loaded or swapped into memory only when they are needed. To release memory as the program runs, the virtual memory manager swaps segments out to the swap file.

dBASE's Virtual Memory Manager (VMM)

dBASE IV version 2.0 uses a Virtual Memory Manager (VMM), which lets dBASE access 5MB and more of extended or virtual memory. VMM runs automatically when you start dBASE IV and evaluates your system's configuration. It then uses internal configuration values to determine how to manage your system's extended memory and disk space for the best performance. VMM dynamically chooses the size of virtual memory, the size of the swap file, and the maximum amount of extended memory dBASE uses.

dBASE IV version 2.0 requires a minimum of 5MB of virtual memory. If your system has less than 5MB of extended memory, VMM creates a swap file on your hard disk to store software code and data segments. With VMM, you can run dBASE IV on systems with as little as 1MB of extended memory, provided you have enough available hard disk space for VMM to create the swap file.

NOTE
When running dBASE IV version 2.0 under OS/2 or Windows, VMM is disabled because OS/2 and Windows use their own virtual memory management systems. For more information about running dBASE IV under OS/2 or Windows, read the "Installing dBASE IV" chapter in Getting Started.

Configuring the Virtual Memory Manager

If you prefer, you can change VMM's default settings to create a more custom-fit environment for your application development.

Using the Dbase.vmc Configuration File

To change VMM's parameter settings, use a text editor to modify Dbase.vmc, the sample configuration file included with dBASE IV. dBASE's installation utility automatically copied this file to your dBASE home directory during installation.

When you start dBASE, VMM looks for Dbase.vmc in the following locations:

- the current directory
- the dBASE home directory
- all directories specified in your DOS path

If VMM finds Dbase.vmc, it uses the defined settings. Otherwise, VMM uses its internal default settings.

The following table describes the parameters you can set in Dbase.vmc:

Parameter	Description	Default
MAXMEM	The maximum amount of extended memory in kilobytes to be used when running dBASE IV	All available extended memory
MINMEM	The minimum amount of extended memory in kilobytes to allocate to dBASE IV.	1024
MAXSEL	The number of the highest selector VMM will manage.	All selectors
VIRTUALSIZE	The maximum size in kilobytes of VMM-managed space. dBASE requires a minimum of 5 MB; if less than 5MB of extended memory is available, VMM creates a swap file on the hard disk.	MAXMEM or 5120, whichever is greater.
SWAPNAME	The name and path of the swap file, such as, C:\DBASE\SWAP.SWP. If you're running dBASE from a server, create the swap file on your local computer to improve per–formance.	The directory that Dbase.exe was loaded from. By default, VMM assigns a randomly-generated name to the swap file.
DELETESWAP/ NODELETESWAP	Tells VMM to delete/save the swap file when your program exits.	DELETESWAP
NOSWAPFILE	Tells VMM to not create a swap file.	VMM creates a swap file as needed.
BLOCKSIZE	The size of each block, in kilobytes, in extended memory and in the swap file.	1024

(continued)

CUSHION	The amount of space in kilobytes to reserve in the swap file as scratch memory.	128
EMS	The maximum amount of EMS memory, in kilobytes, VMM will use for swapping before swapping to a disk file. This parameter is relevant only if you use an expanded memory manager.	0
TRYRO/ NOTRYRO	Specifies if data is marked as read-only or read/write when it is swapped in. Use TRYRO to mark a data segment as read-only when it is swapped in. VMM makes the data segment read/write when you write to it. If you haven't written to it when VMM needs the memory, VMM doesn't write the segment back out to disk, thus improving performance.	TRYRO
FAVORRO/ NOFAVORRO	Specifies if read-only data segments should be swapped out to disk first. Use FAVORRO to swap out read-only segments before swapping out read-write segments. Use NOFAVORRO to indicate no preference.	FAVORRO
RETRYIO	The number of times to retry an operation when an I/O failure occurs.	4
ENVLOW	Specifies that the environment segment of a protected-mode spawned program be allocated from DOS memory.	Disabled

NOTE
All of the parameter values in the sample DBASE.VMC file are commented, so VMM uses its internal default settings. To assign a different parameter value, remove the exclamation mark (!) to uncomment the parameter and enter the new value.

VMM's defaults provide the optimal configuration for most dBASE users. However, you might want to override the defaults when

- *Your program allocates large arrays (totaling several megabytes) and dBASE runs out of memory. In this case, increase the value of VIRTUALSIZE.*
- *You want to run other programs that use extended memory. To do so, set MAXMEM to a value lower than the amount of available extended memory.*

dBASE Data Buffer Management

dBASE IV version 2.0 provides a new data-buffering management system that optimizes file I/O and eliminates the need for external disk-caching software. It dynamically allocates memory to dBASE's data buffers.

To customize memory allocation to this buffer, use the DOS environmental variable, DBASEIV_BUFF. You set this environmental variable at the DOS prompt or in your AUTOEXEC.BAT file. The syntax for DBASEIV_BUFF is:

```
SET DBASEIV_BUFF=buffsize,buffmin,buffmax,initial
```

All parameter values are numeric; dBASE ignores any non-numeric values. Values are separated by commas, and you can omit trailing commas.

If you omit a parameter value, the dBASE default is used. For example, the following resets the buffmax size to 2048K, and default values are used for all the other parameters:

```
SET dBASEIV_BUFF=,,2048
```

The following table describes the parameters in DBASEIV_BUFF:

Parameter	Description	Default
buffsize	Size of each data buffer in kilobytes.Valid values are 2, 4, 8, 16, or 32.	8 or 16 (kilobytes), depending upon the amount of available memory.
buffmin	Minimum amount of memory in kilobytes allocated to data buffers.	128K or 1024K, dependingon how much extended memory is available, and whether dBASE is using 8K or 16K buffers If the amount of extended memory available is greater than 5MB, the default minimum number of buffers is 64. However, when you're running dBASE with 1MB to 5MB of extended memory, this value is set to a minimum of 128K or 16 8K buffers.

(continued)

buffmax	Maximum amount of memory in kilobytes allocated to data buffers.	All of available memory. Depending on the amount of extended memory available and the size of the buffers, this value varies between the buffmin value (e.g., 128K when 1MB of extended memory is available) up to about 14436K on a 16MB system.
initial	Number of buffers to allocate when dBASE starts up.	The number of buffers that will fit in buffmin, plus enough buffers to use three quarters of the difference between buffmin and buffmax. Changing this value increases dBASE's startup time because more buffers are allocated initially rather than on a dynamic as-needed basis.

The following example illustrates how you set DBASEIV_BUFF:

```
SET DBASEIV_BUFF= 8,128,2048
```

This sets the buffer size to 8K, guarantees a minimum allocation of 128K, and a maximum allocation of 2048K (2MB), and leaves the initial buffer allocation value at its default.

NOTE
If you enter a value that dBASE determines is unacceptable, dBASE automatically resets the value. For example, if your system has 1MB of extended memory and you set buffmin to 2048K, dBASE resets the value to its default of 128K.

Buffer Space Considerations

Generally, the more memory you allocate to dBASE's data buffers, the faster your application runs. For example, setting the buffer size to a value larger than 8 speeds up operations that include sequential file access, such as scanning a database file without an active index or copying a subset of one file to another. These operations can be up to five percent faster if you set a buffer size value greater than 8K and a buffmax of 4096K (4MB) when you use larger database files. However, performance reaches a plateau with a total buffer space (buffmax) of 4MB.

In some cases, you can improve performance by allocating less guaranteed minimum memory (buffmin) to the data buffer because more application memory is available. For example, operations such as building index tags benefit from this increased application memory. When dBASE creates tags, it doesn't have to create temporary sort/merge files if all the key values fit into memory.

Getting Information about Memory Allocation and Availability

If you plan to customize memory allocation, you can use dBASE's enhanced MEMORY() function to get information about the total amount of available memory, available low DOS memory, the amount of extended and virtual memory that VMM is managing, the amount of memory the data buffer manager is using, and the size of the swap file.

For more information about MEMORY(), refer to the Language Reference.

Allocating Memory For Caching Index Blocks

When you configure your Config.db, you can use INDEXBYTES to specify the maximum number of 1024-byte blocks that dBASE can use to cache index blocks into memory. The valid range of values is 2 to 2048, or 2K up to 2MB. The default setting is 256, or 256K. A higher INDEXBYTES setting increases indexing speed, but causes dBASE to use more memory for index caching that could be used for other operations instead.

When you close a file or active index tags, dBASE releases the index buffer memory for use in other processing. If your application searches indexes in the same file often, keep the data file open as long as possible to prevent unnecessary reads from disk to memory.

Setting Block Sizes for Memo Field Files

dBASE lets you set different block sizes for memo field files (.DBT). You can change the size of these files to control dBASE's access speed. You use the SET MBLOCK command to specify the size of the block dBASE allocates to new.DBT files. The syntax for SET MBLOCK is

```
SET MBLOCK TO <expN>
```

You can use the following ranges for SET MBLOCK:

	MBLOCK Unit	MBLOCK Size (bytes)
Minimum	1	64
Maximum	511	32704
Default	8	512

dBASE allocates, reads, and writes memo fields in whole blocks, regardless of the actual size of the memo. If your memos are typically much smaller than 512 bytes, you can save large amounts of disk space by using a small value for MBLOCK.

When you specify a size for memo field blocks, use a value that corresponds to the size you expect your memo fields to be, plus a reasonable allowance for increases as a result of subsequent editing.

For example, if you expect your memo field to be 130 bytes long (two lines of 65 bytes), use SET MBLOCK TO 3 (a block size of 192 bytes). This reserves about 62 bytes for later expansion within the current block. Compared to using 512 byte blocks (the default), this approach saves 320 bytes per memo.

When you set a block size, it applies to all memos in the .DBT, so consider the size of all possible memos when you select the optimum MBLOCK value. Using a relatively small block size is probably most efficient since memos larger than one block are always written into contiguous physical blocks in the .DBT file.

Setting Block Sizes for Index Files

dBASE lets you set different block sizes for index files (.MDX) with the SET IBLOCK command. The IBLOCK setting determines the size of indexing blocks. The syntax for SET IBLOCK is

```
SET IBLOCK TO <expN>
```

You can use the following ranges for SET IBLOCK:

	IBLOCK Unit	IBLOCK Size (bytes)
Minimum	1	512*
Maximum	63	32256
Default	2	1024

* Each unit equals 512 bytes. However, the minimum size for an .MDX node is 1024 bytes. When IBLOCK is set to 1, dBASE assigns 1024 bytes instead of 512.

When a single block can't contain all of the key values in an index, dBASE creates one or more subblocks as needed. In this hierarchical arrangement—indexes of indexes—known as the B-tree (balanced tree), subblocks can point up to the root block or to subblocks at lower levels.

dBASE's key-search speed depends on the structure (hierarchy) of the index, which in turn is determined by the size you allocate to each block.

Optimum Index Block Sizes

Determining the optimum index block size depends on several factors, such as the amount of available memory, the distribution of data, whether database files are linked, the length of key values, the number of unique keys, the value of INDEXBYTES (the amount of memory allocated to caching index blocks), the size of your DOS buffer, and the type of operation requested.

You can use the following principles to determine the optimum IBLOCK size for your application:

- dBASE reads and writes whole index blocks only.
- Only one IBLOCK setting can apply to any particular .MDX, regardless of how many tags that .MDX contains. If a specific index has properties that require an IBLOCK setting that is very different from those of the other indexes, consider using a separate .MDX for that tag.
- Using larger index blocks generally improves performance because it increases the amount of data transferred with each physical disk access.
- Larger index blocks produce a "flatter" B-tree, which generally reduces index searching times.
- If your application has a high number of unique key values, a smaller IBLOCK setting could result in a faster search speed because the keys are indexed in logical subblocks.
- The IBLOCK size shouldn't exceed the buffer size you set through the DBASEIV_BUFF environmental variable. If your IBLOCK size is larger than the buffer size, dBASE has to perform multiple DOS disk reads for each index block.

Internal performance testing has shown that IBLOCK settings of 4 or 8 provide good performance in a wide array of applications.

The following formulas show the components that relate the number of keys per node and the value for IBLOCK:

```
IBLOCK = (KeysPerBlock * (KeyLength + KeyPtrSize) + BlockHeader) / 512
KeysPerBlock = (IBLOCK * 512 - BlockHeader) / (KeyLength + KeyPtrSize)
  where,
BlockHeader (overhead) = 12 bytes
KeyPtrSize (overhead) = 4 bytes
KeyLength = Length of Key in bytes
KevsPerBlock = Number of keys per block
512 = size of one IBLOCK unit in bytes
```

Optimizing dBASE's Environmental Settings

You can designate a number of dBASE environmental settings in the Config.db file to improve dBASE's performance. They are described in the following sections.

SET DEVELOPMENT OFF

Use SET DEVELOPMENT OFF if you want to prevent dBASE from comparing the date and time of the source program to the date and time of the compiled object file.

SET SCOREBOARD OFF and SET STATUS OFF

Use SET SCOREBOARD OFF and SET STATUS OFF to prevent dBASE from updating the state of the NumLock, CapsLock, and Ins keys, or the status of a file every time the record pointer moves.

SET AUTOSAVE OFF

Use SET AUTOSAVE OFF to prevent dBASE from writing to disk every time a record is edited or added. When AUTOSAVE is OFF, dBASE writes data to disk only when

- You close the data file
- You change AUTOSAVE to ON
- The dBASE buffer system needs to use the buffer for another operation

SET CLOCK OFF

Use SET CLOCK OFF to prevent dBASE from displaying the clock.

SET ODOMETER TO 200

SET ODOMETER to 200, the maximum value, if you want to limit the number of times dBASE displays information about the progress of an operation.

SET REFRESH TO <high value> or 0

You can either SET REFRESH to a high number, or turn it off by setting REFRESH to 0. This is especially useful with the BROWSE or EDIT command in a multi-user application to reduce the number of screen updates. If you set REFRESH to 0, dBASE still shows changed values when a record is edited.

SET CONSOLE OFF

Use SET CONSOLE OFF while printing to a file or printer if you don't want the data to be written to the screen.

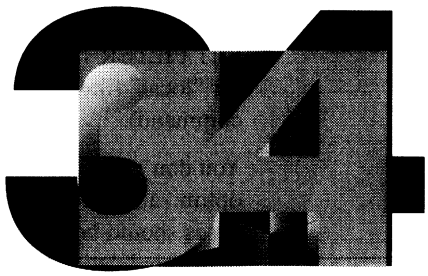
SET TALK OFF

Use SET TALK OFF to prevent dBASE from displaying information as each record or command is processed.

SET PRECISION TO 10

Unless you need a high degree of mathematical precision, use SET PRECISION TO 10 (the minimum) to improve math expression processing speed.

Optimizing Your dBASE Applications



This chapter provides programming tips for optimizing your dBASE applications. It also discusses dBASE's new filter optimization technique, which improves data access significantly.

What This Chapter Covers

Topics in this chapter explain how you can improve your dBASE application's performance by

- Taking advantage of dBASE high-performance filter optimizations
- Keeping frequently used files open
- Moving static expressions out of loops
- Using procedures instead of user-defined functions
- Keeping file and record locks to a minimum
- Avoiding UDFs in index expressions and SET FIELDS TO lists
- Declaring memory variables public
- Using SET RELATION instead of JOIN
- Using SEEK() instead of FIND
- Using SET KEY instead of SET FILTER
- Using IIF instead of IF...ENDIF
- Using SCAN...ENDSCAN Instead of DO WHILE...ENDDO
- Indexing instead of sorting files

dBASE's New High-Performance Filter Optimization

This version of dBASE includes new automatic filter optimization techniques that let dBASE access sets of records very efficiently. You can query a very large database for a small set of records and expect dBASE to return the results hundreds of times faster than before.

Data access speed is enhanced both for "global" filter conditions (when you use the SET FILTER TO command to specify the criteria for the records you want to find) and for "local" filter conditions (when you use a FOR clause to qualify the scope of a command).

You don't need to do any special programming to take advantage of the new filter optimization techniques: the benefits are automatic. For best performance, the databases should be indexed using expressions that correspond to the filter conditions used in the applications. However, whether or not your data is indexed, dBASE's search speed increases the more you work with your database because it retains the results of your previous operations.

Your programs can combine a global filter condition (SET FILTER TO), a key range expression (SET KEY TO), and a local filter expression (FOR clause) to maximize performance when processing a specific set of records.

Optimizable Commands

dBASE's filter optimization techniques work for any command that contains an expression with a FOR clause and a SCOPE clause of ALL (the default) or NEXT. When you use those clauses, the following commands execute more quickly:

AVERAGE	COUNT	LIST	SCAN
CALCULATE	DELETE	LOCATE	SORT TO
CHANGE	EDIT	RECALL	SUM
COPY TO	EXPORT TO	REPLACE	TOTAL ON
COPY TO ARRAY	LABEL FORM	REPORT	

For example, suppose a customer master database is indexed on CUSTOMERID, FIRSTNAME, LASTNAME, and CARD. The following commands benefit greatly from dBASE's filter optimization techniques:

```
COUNT FOR FIRSTNAME = "John"
LIST FOR CARD = "VISA" .AND. BALANCE >= 10000.00
LOCATE FOR CUSTOMERID >= 1000 .AND. CARD = "VISA" .OR. CARD = "MASTERCARD"
```

In the second example, only part of the filter expression corresponds to an index (BALANCE is not indexed). This query is partially optimized the first time. However, because dBASE retains the results of this search operation, future queries with BALANCE in the filter expression are fully optimized.

Handling Multiple File Queries

dBASE's filter optimization technique works with operations in related databases as well as single databases.

Multi-user Considerations

dBASE's filter optimization techniques are also effective in a multi-user environment. However, if another user modifies a shared file while your application has an active filter against that same file, dBASE reprocesses the filter condition before proceeding. This can cause significant delays in a busy multi-user environment.

Getting temporary exclusive access to the file can be a way to prevent such delays. To get exclusive access to a file, you can use SET EXCLUSIVE ON or FLOCK() to lock the file.

General Programming Hints

The following sections describe some programming practices you can use to optimize your dBASE application.

Keep Frequently Used Files Open

In dBASE applications, opening and closing files frequently with the USE command can slow execution. dBASE releases its buffered data from memory each time you close a database file. When you need information from that file again, dBASE has to retrieve the file and any specified indexes from disk. If your dBASE application uses particular databases often, keep them open.

Move Static Expressions Out of Loops

You can eliminate unnecessary processing time by removing from loops any command that can be executed outside the loop. For example, the code in Example 1 is more efficient than the code in Example 2 because dBASE executes SET TALK OFF only once.

Example 1

```
SET TALK OFF
Store 1 to x
DO WHILE x < 100
    STORE x +1 TO x
ENDDO
```

Example 2

```
Store 1 to x
DO WHILE x < 100
    SET TALK OFF
    STORE x +1 TO x
ENDDO
```

Use Procedures Instead of User-defined Functions

Whenever possible, use procedures instead of user-defined functions in your application (especially in loops) because running a procedure is faster than running a user-defined function. Of the following examples, Example 1 executes faster than Example 2:

Example 1

```
lc_Result = ""                    && Predefine the return value
 DO MyProc WITH lc_Result         && MyProc changes the value of lc_Result
```

Example 2

```
lc_Result = MyFunc()
```

Keep File and Record Locks to a Minimum

When you program for a multi-user environment, always release file and record locks as soon as possible to avoid long delays when multiple users try to access the same files.

For example, if you have an order-entry system that updates a parts file when you enter an order, instead of placing a record lock on the parts record when the order for that part is being entered in the order form, wait until the entire order has been entered, then lock and update the parts record.

Avoid UDFs in Index Expressions

Whenever possible, use dBASE's built-in functions instead of user-defined functions to build expressions. dBASE evaluates user-defined functions for each record it processes, so the fewer user-defined functions in your application, the faster dBASE's processing speed.

Declare Memory Variables Public

Declaring all global memory variables PUBLIC at the top of the main program improves overall performance because the values remain in memory. If the memory variables are not public, more processing is required since dBASE releases the variables from memory (creating spaces in the memory variable vectors), compresses and recalculates the vectors.

Use the M-> Prefix on Memory Variables

Precede all memory variable references with the m-> prefix. When a field and a memory variable share the same name, the field name has precedence. If a program uses an unqualified name, dBASE first must search the list of current field names, then search the list of current variable names to find the value of the variable. The m-> tells dBASE that the name is a memory variable reference, speeding up the search.

Use SET RELATION Instead of JOIN

The SET RELATION command links files much faster than the JOIN command. In the following code examples, Example 1 is more efficient than Example 2:

Example 1

```
USE Client
USE Transact IN 2 ORDER Client_id
SET RELATION TO A->Client_id INTO Transact
SET FILTER TO FOUND ("Transact")
SET SKIP TO Transact
GO TOP
COPY TO Newfile FIELDS Client_id, Client, B->Date_trans, B->Total_bill;
B->Order_id
```

Example 2

```
USE Client
USE Transact IN 2
JOIN WITH Transact TO Newfile FOR Client_id=B->Client_id FIELDS;
Client_id, Client, B->Date_trans, B->Total_bill, B->Order_id
```

Use SEEK() Instead of FIND

SEEK() locates data more efficiently than FIND. The FIND command requires a macro expansion of the key expression, whereas SEEK() uses an expression as an argument. In addition, the SEEK() function is faster than the SEEK command because SEEK() combines the two commands, SEEK and IF FOUND().

In the following code examples, Example 1 runs more efficiently than Example 2 or Example 3:

Example 1

```
USE Transact ORDER Order_id
ACCEPT "Enter an order number: " TO mOrder
IF SEEK(mOrder)
   ? "Order found"
ELSE
   ? "Order not found"
ENDIF
```

Example 2

```
USE Transact ORDER Order_id
ACCEPT "Enter an order number: " TO mOrder
FIND &mOrder
IF FOUND()
   ? "Order found"
ELSE
   ? "Order not found"
ENDIF
```

Example 3

```
USE Transact ORDER Order_id
ACCEPT "Enter an order number: " TO mOrder
SEEK mOrder
IF FOUND()
   ? "Order found"
ELSE
   ? "Order not found"
ENDIF
```

Use SET KEY Instead of SET FILTER

If you are searching for a specific set of records in an index, using SET KEY is more efficient than using SET FILTER because SET KEY limits the search to a specific block of records in an index instead of the entire database file. The difference in speed is especially significant when you search a large database. In the following examples, dBASE's search is faster in Example 1.

Example 1

```
USE codes ORDER city
SET KEY TO "Los Angeles"
BROWSE
```

Example 2

```
USE codes
SET FILTER TO city = "Los Angeles"
BROWSE
```

Use IIF() Instead of IF...ENDIF

Use IIF() instead of IF...ENDIF when you set a single variable to one of two values. The IIF() function evaluates faster than IF...ENDIF because it is only one line of code.

The following examples illustrate how differently the IIF() function and the IF...ENDIF construct perform the same task:

Example 1

```
Title = IIF( Sex = "F", "Ms.", "Mr.")
```

Example 2

```
IF Sex = "F"
   Title = "Ms."
ELSE
   Title = "Mr."
ENDIF
```

Use SCAN...ENDSCAN Instead of DO WHILE...ENDDO

When you apply incremental processing commands to the records of a database file, use SCAN...ENDSCAN instead of DO WHILE...ENDDO. SCAN...ENDSCAN has an implicit SKIP or CONTINUE and requires fewer lines of code.

The following examples compare how you can use SCAN...ENDSCAN and DOWHILE...ENDDO to execute a backorder procedure for each uninvoiced record in the Transact data file:

Example 1

```
USE Transact
SCAN FOR .NOT. Invoiced
   DO Backordr WITH Order_id, Client_id, Date_trans
ENDSCAN
```

Example 2

```
USE Transact
LOCATE FOR .NOT. Invoiced
DO WHILE .NOT. EOF()
   DO Backordr WITH Order_id, Client_id, Date_trans
   CONTINUE
ENDDO
```

Index Instead of Sorting Files

Indexing data in your database is more efficient than repeated sorting. Sorting is slow because every record in the database file must be moved. Sorting also requires more disk space.

However, the occasional sorting of a database to correspond to the order of its most important index key can greatly speed up processing when that index is active.

Use Simple Assignment Statements Instead of STORE

Simple assignment statements execute faster than STORE statements. In the following examples, dBASE executes Example 1 faster:

Example 1

```
Number = 10
```

Example 2

```
STORE 10 to Number
```

Appendix

SQL Samples Database

SQL Samples Database

This appendix describes the SQL tables used in the examples in this manual.

When you install dBASE IV and copy the sample files, a SQL database is created. The database is created with a default name of Samples in the directory named \DBASE\SAMPLES. Refer to Chapter 1 of *Getting Started with dBASE IV.*

Customer Table

Column Name	Data Type	Width	Decimal
CUST_NO	Character	6	
COMPANY	Character	25	
LASTNAME	Character	15	
FIRSTNAME	Character	10	
ADDRESS	Character	20	
CITY	Character	15	
STATE	Character	2	
ZIP	Character	5	

Record#	CUST_NO	COMPANY	LASTNAME	FIRSTNAME	ADDRESS	CITY	STATE	ZIP
1	000001	Leonard Design Services	Leonard	Rick	1550 Keystone St.	Oceanside	CA	92054
2	000003	Ace Furniture	Martin	Lisa	1960 Lindley Ave.	Wausau	WI	54401
3	000009	Custom Furniture	Pollock	Daniel	5934 Pine Needles	Yonkers	NY	10709
4	000011	The Office	LeClerc	Dominique	101 Pierce St.	New York	NY	10013
5	000016	American Business Supply	Daniels	George	5601 Grand Ave.	Los Angeles	CA	90233
6	000017	Black's Furniture Store	Jackson	Dennis	7010 Balcom Ave.	San Francisco	CA	94119
7	000018	Interior Systems	Goetz	John	899 Kenwood St.	Milwaukee	WI	53201
8	000019	The Designer	Hobbs	Luke	6043 Whiteside Blvd	New York	NY	10713
9	000022	Las Vegas Furniture	Hart	Paul	8301 Sale St.	Las Vegas	NV	89106
10	000024	Baker Furniture	Campbell	Linda	6700 Tyler St.	Phoenix	AZ	85012
11	000025	Modern Furniture Store	Hamilton	Robert	366 Shirley Ave.	Phoenix	AZ	85004
12	000027	A1 Office Supply Store	McVeigh	John	1240 Embarcadero	San Francisco	CA	94102
13	000028	Accent Furniture Designs	Squire	Ann	20984 Horizon Hills	Las Vegas	NV	89108
14	000031	Al's Furniture & Supplies	Thompson	Kathy	40555 Brentwood	St. Louis	MO	63121
15	000032	Contemporary Designs	Trujillo	Michelle	5670 Colorado Blvd	Milwaukee	WI	53220
16	000033	Interior Designs	Long	Chuck	40677 Misty Isle Dr	White Plains	NY	10605
17	000034	La Cienega Furniture	Keegan	Marilyn	6045 Vineland Blvd	Los Angeles	CA	90815
18	000035	Valley Furniture	Yanish	Diane	10111 Ventura Blvd	Encino	CA	91316
19	000036	New Horizons	Brendon	Kelly	12508 Robin Hood Ln	Chicago	IL	60619
20	000040	Design Center Interiors	Gilbert	Chuck	7619 Kraft Dr.	Las Vegas	NV	89106
21	000042	Cohen's Furniture	Cohen	Larry	908 Glen Oaks Ave.	San Francisco	CA	94119
22	000043	To Design Furniture	Tsuma	Tamio	4564 Benedict Canyon	Rochester	NY	14625
23	000045	Classic Interiors	Lawson	Eric	2015 Edmonton	St. Louis	MO	63106
24	000046	Commercial Interiors LTD	Young	Sandy	14097 Gilmore	Ventura	CA	93003

Staff Table

Column Name	Data Type	Width	Decimal
STAFF_NO	Character	6	
LASTNAME	Character	15	
FIRSTNAME	Character	10	
HIREDATE	Date	8	
LOCATION	Character	15	
SUPERVISOR	Character	6	
SALARY	Numeric	6	
COMMISSION	Numeric	4	1

Record#	STAFF_NO	LASTNAME	FIRSTNAME	HIREDATE	LOCATION	SUPERVISOR	SALARY	COMMISSION
1	000001	Zambini	Rick	02/15/80	LOS ANGELES	000000	6000	5.0
2	000003	Vidoni	Cheryl	03/06/80	NEW YORK	000000	5780	5.0
3	000004	Coudray	Sandy	06/06/80	LOS ANGELES	000001	6237	5.0
4	000006	Thomas	Pat	01/08/81	NEW YORK	000003	5875	5.0
5	000008	McLester	Debbie	04/12/81	LOS ANGELES	000001	4792	5.0
6	000011	Michaels	Delores	05/05/82	CHICAGO	000012	4927	7.0
7	000012	Charles	Ted	02/02/83	CHICAGO	000000	5945	5.0
8	000013	Marin	Mark	06/05/83	LOS ANGELES	000001	4802	11.0
9	000015	Roddick	Mary	02/13/84	NEW YORK	000003	5493	8.0
10	000016	Long	Nicole	08/18/84	NEW YORK	000003	5190	7.0
11	000019	Rolfes	Chuck	09/09/84	LOS ANGELES	000001	4586	6.0
12	000020	Sanders	Kathy	03/23/85	CHICAGO	000012	3783	5.0

Inventry Table

Column Name	Data Type	Width	Decimal
PART_NO	Character	6	
DESCRIPT	Character	30	
ON_HAND	Numeric	4	
LOCATION	Character	15	
UNITCOST	Numeric	8	2
DISCNTINUE	Logical	1	

Record#	PART_NO	DESCRIPT	ON_HAND	LOCATION	UNITCOST	DISCNTINUE
1	001001	WORKSTATION-ELECTRONIC OFFICE	2	CHICAGO	1296.29	.F.
2	001002	HOME OFFICE SUITE	2	LOS ANGELES	1395.49	.F.
3	001005	EXECUTIVE SUITE ENSEMBLE	1	CHICAGO	2125.79	.F.
4	001007	WOOD DESK-SINGLE PEDESTAL	29	NEW YORK	736.21	.F.
5	001008	WORKSTATION-STAND	22	LOS ANGELES	275.66	.F.
6	001009	CHAIR-ADJUSTABLE SWIVEL	124	CHICAGO	245.38	.T.
7	001015	CREDENZA-OAK SLIDING DOOR	15	NEW YORK	745.00	.T.
8	001019	TABLE-BOARD ROOM	12	NEW YORK	4250.00	.F.
9	001021	MANAGERS OFFICE ENSEMBLE	3	NEW YORK	2380.79	.F.
10	001022	TABLE-WALNUT OCCASIONAL	5	NEW YORK	414.95	.F.
11	001024	LAMP-BRASS TABLE	140	CHICAGO	230.79	.F.
12	001025	DESK-EXECUTIVE-5 FOOT	63	LOS ANGELES	985.00	.F.
13	001029	FILE CABINET-2 DRAWER	200	NEW YORK	89.95	.T.
14	001031	CHAIR-EXECUTIVE SWIVEL/TILT	79	LOS ANGELES	420.00	.F.
15	001032	FILE CABINET-4 DRAWER	15	CHICAGO	134.69	.F.
16	001033	CHAIR-TRADITIONAL ARM	20	CHICAGO	125.00	.T.
17	001038	LAMP-DRAFTING SWING ARM	169	LOS ANGELES	149.59	.F.
18	001038	LAMP-DRAFTING SWING ARM	47	NEW YORK	149.59	.F.
19	001007	WOOD DESK-SINGLE PEDESTAL	35	CHICAGO	736.21	.F.
20	001007	WOOD DESK-SINGLE PEDESTAL	62	LOS ANGELES	736.21	.F.
21	001013	CHAIR-MODERN PNEUMATIC	115	CHICAGO	275.80	.F.
22	001013	CHAIR-MODERN PNEUMATIC	35	LOS ANGELES	275.80	.F.
23	001032	FILE CABINET-4 DRAWER	71	LOS ANGELES	134.69	.F.
24	001038	LAMP-DRAFTING SWING ARM	89	CHICAGO	149.59	.F.
25	001027	DESK-EXECUTIVE-6 FOOT	20	CHICAGO	1475.00	.F.
26	001027	DESK-EXECUTIVE-6 FOOT	56	NEW YORK	1475.00	.F.
27	001031	CHAIR-EXECUTIVE SWIVEL/TILT	44	CHICAGO	420.00	.F.
28	001031	CHAIR-EXECUTIVE SWIVEL/TILT	76	NEW YORK	420.00	.F.
29	001024	LAMP-BRASS TABLE	56	NEW YORK	230.79	.F.
30	001025	DESK-EXECUTIVE-5 FOOT	47	NEW YORK	985.00	.F.
31	001001	WORKSTATION-ELECTRONIC OFFICE	3	LOS ANGELES	1296.29	.F.
32	001005	EXECUTIVE SUITE ENSEMBLE	0	NEW YORK	2125.79	.F.
33	001029	FILE CABINET-2 DRAWER	145	LOS ANGELES	89.95	.T.

Assembly Table

Column Name	Data Type	Width	Decimal
ASSEMBLY	Character	6	
SUBASSY	Character	6	
QTY	Numeric	3	

Record#	ASSEMBLY	SUBASSY	QTY
1	001001	001007	1
2	001001	001013	1
3	001001	001032	1
4	001001	001038	1
5	001005	001027	1
6	001005	001031	1
7	001005	001024	1
8	001021	001025	1
9	001021	001031	1
10	001021	001024	1
11	001021	001015	1
12	001002	001025	1
13	001002	001032	1
14	001002	001013	1

Sales Table

Column Name	Data Type	Width	Decimal
ORDER_NO	Character	6	
SALE_DATE	Date	8	
STAFF_NO	Character	6	
CUST_NO	Character	6	
INVOICED	Logical	1	

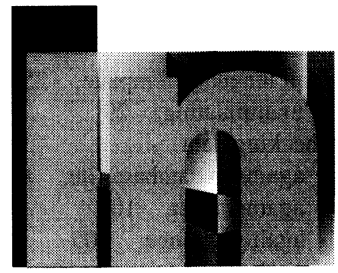
Record#	ORDER_NO	SALE_DATE	STAFF_NO	CUST_NO	INVOICED
1	020002	09/21/87	000008	000025	.F.
2	020003	09/21/87	000006	000043	.F.
3	020004	09/21/87	000019	000034	.F.
4	020005	09/21/87	000001	000016	.F.
5	020006	09/22/87	000012	000036	.F.
6	020007	09/22/87	000015	000019	.F.
7	020008	09/22/87	000003	000011	.F.
8	020009	09/22/87	000012	000018	.F.
9	020010	09/22/87	000011	000031	.F.
10	020011	09/22/87	000015	000040	.F.
11	020012	09/22/87	000008	000027	.F.
12	020013	09/23/87	000012	000036	.F.
13	020014	09/23/87	000001	000001	.F.
14	020015	09/23/87	000015	000019	.F.
15	020016	09/23/87	000015	000011	.F.
16	020017	09/24/87	000006	000032	.F.
17	020018	09/24/87	000011	000038	.F.
18	020019	09/24/87	000013	000016	.F.
19	020020	09/24/87	000013	000031	.F.
20	020021	09/25/87	000008	000046	.F.
21	020022	09/25/87	000004	000027	.F.
22	020023	09/25/87	000003	000040	.F.
23	020024	09/25/87	000012	000045	.F.
24	020025	09/25/87	000003	000019	.F.
25	020026	09/25/87	000004	000017	.F.

Items Table

Column Name	Data Type	Width	Decimal
ORDER_NO	Character	6	
PART_NO	Character	6	
QTY	Numeric	3	
SHIPPED	Logical	1	

Record#	ORDER_NO	PART_NO	QTY	SHIPPED
1	020002	001032	2	.F.
2	020002	001025	3	.F.
3	020002	001013	3	.F.
4	020003	001021	4	.F.
5	020003	001005	2	.F.
6	020004	001027	5	.F.
7	020004	001038	5	.F.
8	020004	001013	5	.F.
9	020005	001019	2	.F.
10	020006	001007	25	.F.
11	020006	001031	25	.F.
12	020007	001022	3	.F.
13	020007	001033	3	.F.
14	020008	001007	3	.F.
15	020009	001029	31	.F.
16	020010	001005	5	.F.
17	020010	001021	2	.F.
18	020011	001029	7	.F.
19	020011	001025	4	.F.
20	020011	001031	7	.F.
21	020012	001015	5	.F.
22	020013	001022	2	.F.
23	020013	001019	1	.F.
24	020014	001021	2	.F.
25	020015	001025	15	.F.
26	020016	001031	4	.F.
27	020016	001025	2	.F.
28	020017	001029	6	.F.
29	020018	001038	4	.F.
30	020019	001027	3	.F.
31	020020	001024	7	.F.
32	020020	001032	4	.F.
33	020021	001013	8	.F.
34	020021	001025	8	.F.
35	020021	001024	6	.F.
36	020022	001015	1	.F.
37	020023	001024	12	.F.
38	020024	001009	3	.F.
39	020024	001027	3	.F.
40	020025	001019	1	.F.
41	020026	001007	9	.F.
42	020026	001013	9	.F.
43	020026	001024	5	.F.

Index



Symbols

A

B

C

D

E

F

G

H

I

J

Q

R

S

T

U

V

W

Y