Contents

PROGRAMMING WITH dBASE III PLUS

Introduction

FROM USER TO PROGRAMMER

The on-disk tutorial, *Learning dBASE III PLUS*, and *Using dBASE III PLUS* have taught you how to use dBASE III PLUS for all your database management needs. This book shows you how powerful dBASE III PLUS really is.

dBASE III PLUS is not only a complete database management system, but it also includes a programming language called dBASE. With this programming language, you can create customized applications for your specific needs, allowing you control of all your database management tasks. You can even sell your applications to others.

If the word *programming* causes you chills of fear and apprehension, relax! Because dBASE III PLUS has so many built-in features, programming in dBASE is easier to learn than many other programming languages. Programming can even be a great deal of fun.

About This Section

This section gets you started with programming in dBASE. It provides the essentials that you'll need to know to write your own programs. When you have finished this section, you'll be familiar with the full power of dBASE III PLUS.

If you already know how to program in earlier versions of dBASE, use this book as a reference guide for the many new features of dBASE III PLUS.

You Know More Than You Realize

You may not realize it, but you've been programming in dBASE right from the beginning. When you issue commands from the dot prompt or in The Assistant (for example, when you select or type LIST to show the contents of a database file), you are telling dBASE III PLUS to perform certain operations. Generally, you issue a series of related commands in a certain sequence. Programming in its simplest form means collecting this series of commands together into a program file, saving it to the disk, and then performing the commands by running the program file.

Take a look at the following series of commands. Suppose you have a database file, called Money.dbf, which contains information about people who owe you money. Naturally, you would want to check this information on a regular basis to see who still owes you money. The Money.dbf file is INDEXed on the last name to the index file Last.ndx. Amount_due is a numeric field; Owing and Pastdue are logical fields. Gimme and Scrooge are REPORT FORMs that report on which people have to pay up. What do these command lines do?

```
USE Money INDEX Last
LIST FOR Owing
CLEAR
LIST FOR Owing .AND. Pastdue
CLEAR
SUM Amount_due FOR Owing
SUM Amount_due FOR Owing .AND. Pastdue
REPORT FORM Gimme FOR Owing PLAIN TO PRINT
REPORT FORM Scrooge FOR Owing .AND. Pastdue PLAIN TO PRINT
USE
```

That's a lot of typing to do on a regular basis! However, you can set up these same commands in a program file called Owing.prg, and all you would have to do is type Do Owing from the dot prompt to get the very same results.

This is a simple example, but it illustrates how much time you can save by learning a little dBASE programming.

Total Control

Where dBASE programming really shines is in its ability to allow you to control how your users work with dBASE III PLUS. This way, you can ensure that users don't have direct access to important database information. Your program acts as an intermediary between users and the database, protecting your valuable information from possible catastrophes and making dBASE III PLUS easier to use, especially for inexperienced users. You can even have dBASE III PLUS start your program automatically each day. This is known as a turnkey system.

If many people in your office share in daily tasks, for instance, entering new records or updating database information, you can set up programs to guide them through these tasks. Such programs require less learning time for new users, and they protect your database file. You can create menu-driven programs so that all users need do is follow the prompts and press a few keys for standard dBASE III PLUS operations, such as displaying information, updating database files, and printing reports.

How to Use This Section

You should have a general understanding of dBASE III PLUS by reading *Learning dBASE III PLUS* and *Using dBASE III PLUS* before you begin using this section. Because it is only meant to get you started with programming in dBASE, this section doesn't offer an exhaustive discussion of every dBASE programming feature. Work with this section and *Using dBASE III PLUS*, which contains an extensive reference section for all of the dBASE III PLUS commands and functions.

Rather than typing in commands from the dot prompt, you will primarily learn dBASE programming by studying a complete dBASE program — a checkbook management system that keeps track of a checking account. You can use it to balance your own checkbook, if you wish.

The checkbook management system is composed of a series of files on the Sample Programs and Utilities disk. Here is a list of these files:

Add.prg	Curson.bin
Bank.dbf	Deposits.dbf
Cancl.prg	Editvoid.prg
Cash.prg	Help.prg
Cbmenu.prg	Maint.prg
Cbmenu2.prg	Menumask.prg
Check.prg	Numwords.prg
Checks.dbf	Recon.dbf
Chkbook.mem	Reconcil.prg
Chkmask.prg	Reinit.prg
Chkno.ndx	Reports.prg
Cleanup.prg	Rprtpro.prg
Clrcash.prg	Tax.dbf
Clrdep.prg	Taxcodes.prg
Cursoff.bin	Yearend.prg

You'll learn more about the checkbook management system in Chapter 1. Because the following files provide most of the programming examples in this book, you should print out these files, following the instructions below, and refer to the printouts as you study this book. Here they are:

Add.prg
Cancl.prg
Cbmenu.prg
Check.prg
Clrcash.prg
Clrdep.prg
Reconcil.prg

You can use a word processing program or dBASE III PLUS to print the program files on continuous-form paper by following these steps:

1. Make a working copy of the Sample Programs and Utilities disk. Put the original in a safe place.
2. Start dBASE III PLUS and SET DEFAULT TO the drive containing the working Sample Programs and Utilities disk. On dual-floppy systems, the command is: SET DEFAULT TO B ←┘.
3. Turn on your printer and adjust the paper so that the top-of-form setting is correct. The top-of-form is where the paper perforation is located.
4. Type at the dot prompt: TYPE Add.prg TO PRINT ←┘.
5. When you have printed Add.prg, repeat the previous step, substituting the correct filename, for each file in the list.

What Each Chapter Covers

The first three chapters of this section deal with essential programming concepts and dBASE programming techniques. Read these chapters in order. The rest of the section provides a breakdown of dBASE programming from a task-oriented approach. You can skip around as you wish and study the topics you need in any order. However, if you're new to programming, it's recommended that you read the chapters in sequence.

Chapter 1 discusses basic programming concepts and how to write programs.

Chapter 2 investigates the structure and makeup of dBASE programs.

Chapter 3 shows you how to use memory variables in programs.

Chapter 4 steps you through the Cbmenu.prg module of the checkbook management system as an example of what a main program does.

Chapter 5 introduces you to converting fields and memory variables from one type to another, such as from string to numeric. This information is important for designing screen forms.

Chapter 6 shows you how to set up screen forms in your programs and how to get user input.

Chapter 7 continues the discussion of setting up screen forms by presenting templates and ranges for filtering input. The end of the chapter contains points to consider when designing screen forms.

Chapter 8 looks more closely at screen forms by providing many tips for enhancing the look of your screens. It also explains format files and how to work with memo fields.

Chapter 9 discusses the important notion of error trapping and how to check user input for correctness.

Chapter 10 discusses the setting up of database and index files, using several work areas at once, and retrieving information.

Chapter 11 investigates ways to use and change data in the database file from your programs. It also talks about setting up relationships between work areas.

Chapter 12 takes a look at how to control printing in your programs and special printing effects.

Chapter 13 discusses several aspects of housekeeping: how to complete your programs, how to close database files and restore the working environment, and how to handle file maintenance in your programs.

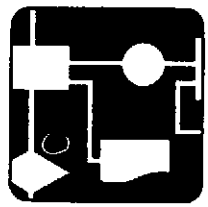
Chapter 14 steps you through one module program, Cancl.prg, from the checkbook management system as an example of how programming tasks work together in a real-life situation.

Chapter 15 shows you ways to test and debug your programs.

Chapter 16 investigates more advanced features, such as the use of PROCEDURE files, parameter passing, hidden variables, and assembly language routines. You'll also see how dBASE III PLUS interacts with the operating system and other applications. The end of the chapter illustrates a turnkey system.

You're now ready to begin learning how to program in dBASE.

Chapter 1
PROGRAMMING IN A NUTSHELL



If you're new to programming, you'll find it worth your while to learn some general programming principles and good programming techniques before actually writing dBASE programs. You can then build your future programming efforts on this foundation. Please read Chapters 1 through 3 in sequence. Each chapter provides important information and builds upon concepts in previous chapters. If you're familiar with other programming languages, you might wish to review these first three chapters.

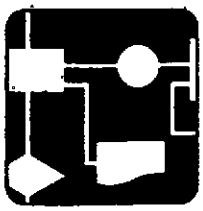
What This Chapter Covers

This chapter discusses the following:

- What programming is
- The difference between programs and applications
- Things you should know, such as setting the working environment, input and output, error trapping, menus, and program flow
- The basic program flow control structures: branching, looping, and dealing with conditions
- Some popular techniques for developing programs, including using the top-down approach, structured programming, and the importance of documentation
- The steps to designing programs

Preparing for This Chapter

You should have a working knowledge of dBASE III PLUS. Make sure that you have your printout of the example program files before using this book. See the Introduction for information about printing program files. You'll also run the checkbook management program, so have your Sample Programs and Utilities disk handy.



What Is Programming?

A program is the set of instructions and commands that tells your computer to do specific tasks. Programming is not something done only with computers. In a sense, any series of instructions in a specific order is a program. For example, a recipe is really a program. When you use a recipe, you follow a series of step-by-step instructions to cook something. Each instruction is a command — for instance, add ½ teaspoon of salt — just as each dBASE instruction is a command.

The difference between instructing a person to cook and programming a computer to compute is that the person can choose to alter the recipe slightly and thus bend the rules a bit. A computer never bends its rules. It follows instructions to the letter — whether they're correct or not.

A program must contain every instruction that you want the computer to perform in a precise order. To organize and write these instructions correctly and efficiently is what programming is all about. Once you've written a program in dBASE, you can save it to the disk in a program file and run the program whenever you wish.

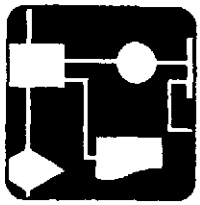
Programs and Applications

A program can be a series of commands that do one basic task, or an entire programming project that performs all your tasks. The latter is often called an application program. You can break down the entire programming project into modules, each module performing one specific task. One main program usually controls all these modules.

So, a program can refer to the entire project or merely one modular part of it. The checkbook management system, for example, is a series of modular programs working together.

Languages and Interpreters

Because it contains English words, such as IF and DO, the dBASE programming language is easy to use. However, computers can't read English words, so the dBASE III PLUS interpreter reads each command in order and translates it into a language that the computer understands. Whether you include the commands in a program or issue them at the dot prompt, you still rely on dBASE III PLUS to interpret each command for the computer.



When you tell dBASE III PLUS to CREATE a new database, the dBASE III PLUS interpreter first compares the command's individual letters, C-R-E-A-T-E, to a table of commands. This is known as parsing the command. If the interpreter finds an exact match between what you typed and the commands in the table, it executes the instructions that go with that command. If it doesn't find an exact match, it gives you an error message. The interpreter can't understand an incorrect command, so you must enter commands precisely.

You can save yourself many headaches by avoiding the four most common programming mistakes:

- Misspelling commands
- Forgetting to separate commands from their expressions with at least one space
- Syntax errors, that is, using a command incorrectly
- Issuing an incomplete command

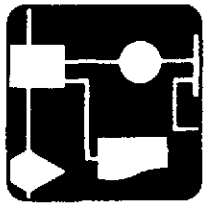
Programming Concepts

This section discusses some general concepts that apply to dBASE programming. These concepts are, on the whole, applicable to other programming languages as well, although the specific commands differ.

Writing and Editing dBASE Programs

You write a program and save it in a program file with the extension .prg. dBASE III PLUS has a built-in word processor designed for creating and editing dBASE program files. It allows you to stay in dBASE III PLUS, edit, and then run your programs.

To get into the dBASE III PLUS word processor, type MODIFY COMMAND, followed by the filename. dBASE III PLUS automatically supplies the .prg file extension for you. For example, if you were beginning a new program file called Test.prg, the command from the dot prompt would be MODIFY COMMAND Test.



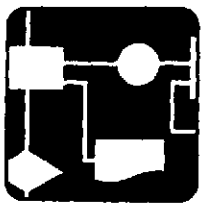
When you wish to edit an existing program file, you also use MODIFY COMMAND with the name of the file. The editing commands of dBASE III PLUS's word processor are very similar to those of WordStar™. When you use MODIFY COMMAND, you'll see a menu of the most important edit commands at the top of the screen. Press F1 to toggle the menu on or off. A complete list of the edit commands is in the Quick Reference Guide and in *Using dBASE III PLUS*.

When you've finished editing a program and wish to save it to disk, press Ctrl-End. If you don't want to save your changes, press the Esc key and respond with Y to the prompt: Abort editing ? (Y/N). dBASE III PLUS reinstates the file to its original form and returns you to the dot prompt.

You can use any word processor or text editor to create and edit dBASE programs, provided that it can create an ASCII text file. (The American Standards Committee for Information Interchange, ASCII, standardized the internal codes used by your computer for the upper case and lower case letters of the alphabet, numbers, punctuation marks, and control characters. These standards are known as the ASCII code.)

Program files can contain only text. Each command is on a separate line that ends with a ↵. If you use a word processor other than dBASE III PLUS's to write your programs, make sure that your program text files do not contain any formatting codes, for example, special page formats, hyphenation, and boldface print.

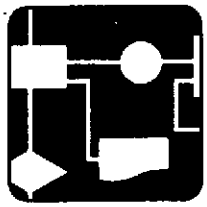
Most popular word processors let you create text files without formatting codes. Framework II™ permits DOS text files, while WordStar uses the non-document mode. When using a word processor to write dBASE programs, remember to include the .prg extension in the filename. With Framework II, always save the file as a DOS text file from the Disk Menu, not as a regular Framework II file.



TIP
If your computer has enough memory, you can set up the Config.db file so that your word processor of choice is the default word processor. Then, when you use MODIFY COMMAND, dBASE III PLUS automatically loads your word processor. See *Using dBASE III PLUS* for information about the Config.db file.

Although a program file can be as long as you wish, the dBASE III PLUS word processor can handle files of up to 5,000 characters, approximately 100-200 lines. Some of the files in the checkbook management system are large. When you use MODIFY COMMAND to open a large program file, you may get the message **File too large, some data may be lost.** If this happens, use another word processor to view these files.

TIP
If you merely wish to view the contents of a program file, use either MODIFY COMMAND or the TYPE command. When you TYPE a file, it scrolls past you on the screen. Press **Ctrl-S** to stop the scrolling, then press any key to continue.



Running the Program

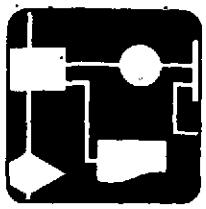
When you wish to run the program, issue the DO command, followed by the name of the program file. dBASE III PLUS opens the program file and reads each line of the file, starting at the top of the file, performing each command in sequence. When it has reached the last command in the file, or if the user decides to cancel the program prematurely, dBASE III PLUS closes the program file and returns to the dot prompt. If the last command in a program file is QUIT, control returns to the operating system.

Setting Up the Program Environment

When you work with dBASE III PLUS from the dot prompt, that is the *environment* in which you are working. You have a large number of ways to change this environment for your particular needs. For example, if all your database and index files are on the B drive and dBASE III PLUS is on the A drive, you normally use SET DEFAULT TO B at the beginning of your session with dBASE III PLUS. You would include at this point in the program other environmental conditions, such as screen color. You'll learn how to set the program environment in Chapter 4.

Input and Output

Every program accepts input and produces output. These are terms for the source and destination of information, respectively. For example, input could come from fields in a database file in USE, or the user could enter data from the keyboard during the run of the program. Output could be information that updates previous data in a database file, a listing of fields on the screen, or a printed report.



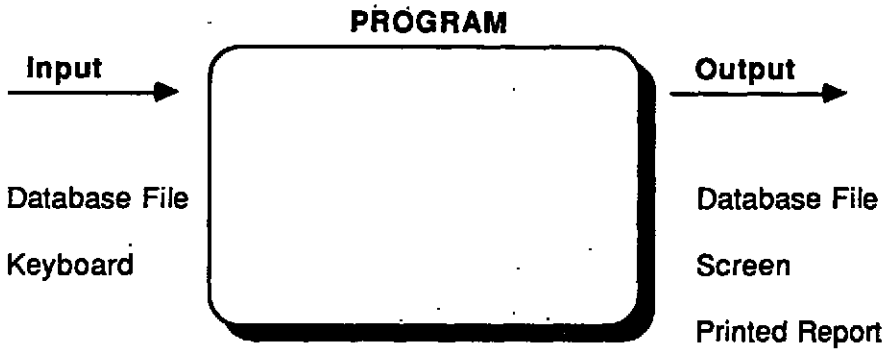


Figure 1-1 Types of input and output

Your customized program will handle input and output differently than dBASE III PLUS does from the dot prompt. The program has to be able to deal with input and output without any intervention on your part. You, the programmer, may not be around when someone else is running your program.

Screen forms are a great way to handle input and output. For input, screen forms act as fill-in-the-blank instructions, with prompts and messages to guide the user. You can design screen forms which mimic your own forms. This makes your program more accessible to inexperienced users, because they will recognize forms that they already know.

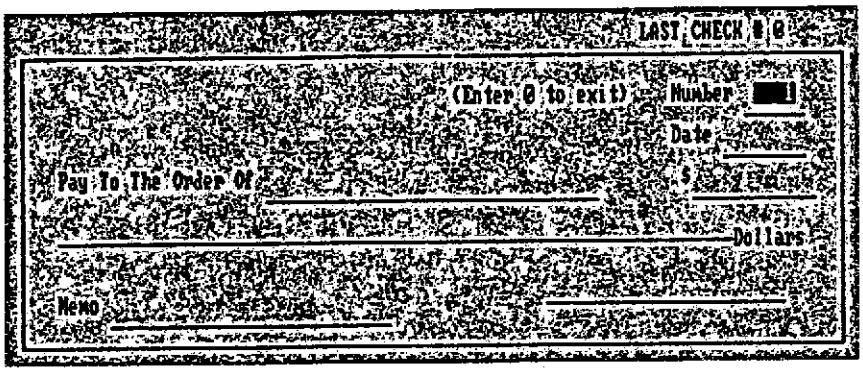


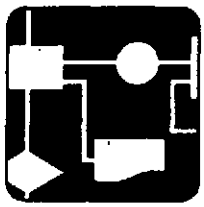
Figure 1-2 A typical screen form

You can also use screen forms to display output, show what the user has just typed, ask for verification from the user that the input is correct, or provide information that the user requests. Designing screen forms is an important part of programming. You will learn more about them in Chapters 6, 7, and 8.

You can store information, such as data that the user has keyed into an on-screen form, in temporary storage locations in the computer's memory. In dBASE, these temporary storage locations are called memory variables. Memory variables are like little cubbyholes where you can store information and call up this information when needed with the ?, DISPLAY MEMORY, or LIST MEMORY commands. You'll learn more about them in Chapter 3.

Garbage In/Garbage Out

Computer programmers often refer to a concept known simply as GIGO, which stands for garbage in/garbage out. This means that the quality of the output is directly related to the quality of the input. There are many ways to eliminate the GIGO problem. They are collectively known as error-trapping routines. All well-designed programs have provisions for error trapping.



Remember that a computer has to have instructions for every possible situation or condition, so be sure to consider all potential errors that the user might make when you set up error trapping. For example, if you want the user to type Y or N in response to a yes/no question, make sure that the program knows what to do if the user types another letter by mistake. You will learn more about error trapping in Chapter 9.

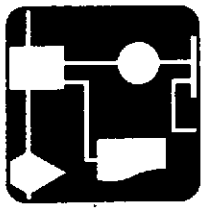
Database Integrity

Keeping your database information correct is called maintaining the integrity of the database file. The program should maintain and safeguard the unity and accuracy of the database files. To achieve this, you can use memory variables to store new information for validation before updating the database file. Also, your program should make frequent backups of database files. This is the topic of Chapter 13.

The Flow of a Program

Design your program so that the commands work in a logical, progressive order. This is known as program flow. However, if you want the program to repeat the same command a number of times, or to perform a command only if a certain condition exists, the step-by-step approach is impractical. You wouldn't want to retype the same commands over and over, for instance, to display a menu on the screen every time the menu were to appear. Similarly, you wouldn't want the program to evaluate a situation that may not occur.

You can use control mechanisms that instruct the program to branch to subprograms that handle repetitive tasks, repeat a series of commands while a condition exists, or deal with a possible condition. These three types of control mechanisms are known as branching, looping, and conditions, respectively. Branching, looping, and conditions are all temporary changes to the basic program flow. They let you create programs with flexibility and power.



Some kind of true or false condition governs all three basic programming control structures. That is why the logical operators .NOT., .AND., and .OR. are important for computers. They let you determine actions according to whether a condition is met, true = .T., or not met, false = .F. When programming, you will use logical operators often. Below is a brief discussion of the basic programming control structures. You will investigate them in more detail in Chapter 2.

Making a Detour (Branching)

To use the recipe analogy from the beginning of the chapter again, suppose you're making stuffed peppers. At a certain point, the recipe tells you to fill each pepper with the stuffing, which is in another recipe. You would turn your attention away from the main recipe for a while to make the stuffing recipe. After you'd made the stuffing, you would return to where you left off in the main recipe and continue. This procedure, called branching, is also an important concept in computer programming.

You will use the branching technique a great deal to direct the flow of the program according to what the user wants to do. A simple example of branching is a menu, which lists the choices available in the program. After the user has chosen a task, the program branches to the instructions for that task, which are in a subprogram file. When the task is completed, program flow returns to the program containing the main menu for another choice from the user.

In dBASE, you issue the DO command, followed by the name of the subprogram, for branching. The program module which contains the DO command is the calling program. When it comes to a DO instruction, the computer performs the instructions at the branch location and, when finished, returns to the calling program to execute the next command in sequence. Here is a graphic illustration of branching.

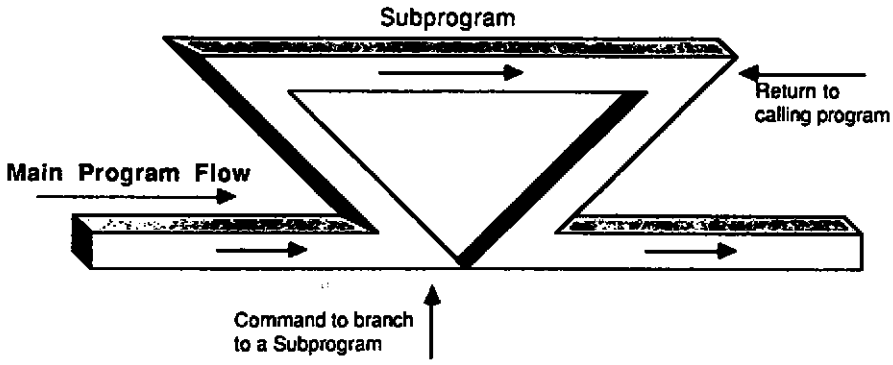


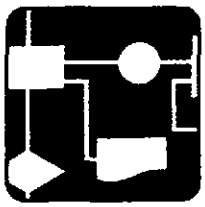
Figure 1-3 Branching

Normal program flow is interrupted by an unconditional branch to a subprogram. When the subprogram is finished, program flow returns to the line immediately following the branch command in the calling program.

Repeating a Procedure (Looping)

You may often want your program to perform the same steps repeatedly while a given condition is true. In computer programming, this is known as looping. The program repeats the same commands in a loop until it reaches the point where the condition is no longer true. It then leaves the loop and continues with the next instruction in the program. In dBASE, you enclose the commands to be performed in a loop within a structure that begins with a DO WHILE command and ends with an ENDDO command. These two commands constitute the boundaries of the loop.

As an example, suppose you want to replace each unit price field in a database file with an inflated price. With the looping technique, you instruct the program to do the loop while the record pointer has not reached the end of the file. dBASE III PLUS goes to the first record in the file, performs the desired operation, skips to the next record, and loops back to repeat the same operation.



Eventually, dBASE III PLUS will reach the end of the database file. This is the condition which it has been told to look for, so it ends the loop. The program continues with the next instruction following the loop. Figure 1-4 shows you how a loop works.

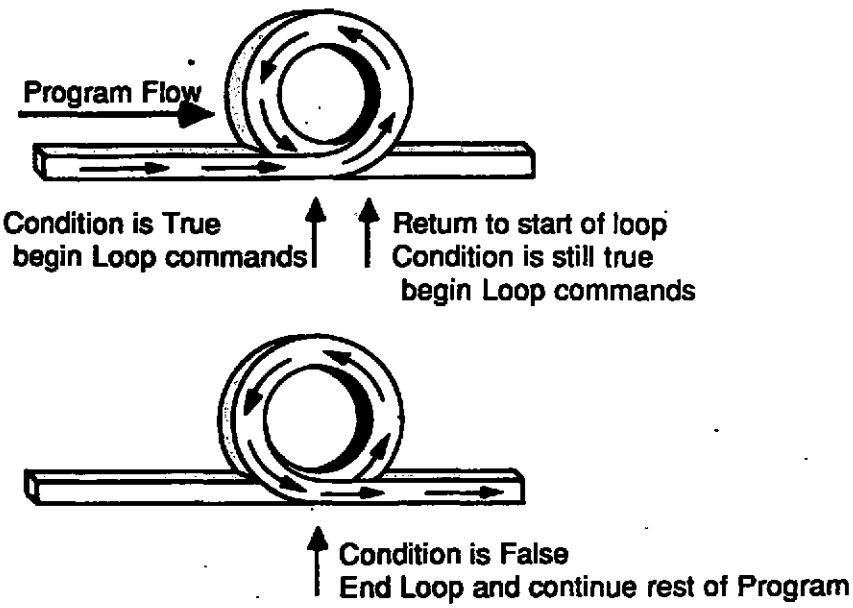
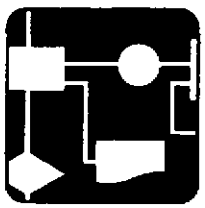


Figure 1-4 How a loop works

Normal program flow arrives at a condition to be tested. If the condition tests as true, the program performs the commands in the loop and continues to repeat the loop as long as the condition at the beginning of the loop is true. When the condition is no longer true, the program leaves the loop and continues with the command following the loop.



Dealing with Possibilities (Conditions)

Looping is a good technique when the condition is always predictable, for example, there will always be an end-of-file somewhere. However, you may wish to link certain instructions to conditions that may or may not happen. Suppose you're replacing all price information with an inflated price, but only for items that haven't sold in the last six months. There might be no records that fit this condition, or there might be many.

You can set up this type of control in two ways, depending on the context. If there are only two possibilities, you normally use a construction that begins with an IF statement and ends with an ENDIF. The program checks each field in the file, making replacements only in those fields that conform to the condition.

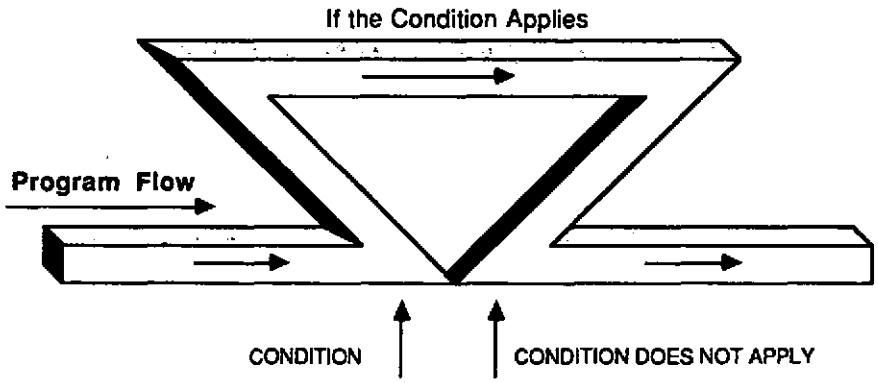


Figure 1-5 Conditions

When it reaches the condition, the program evaluates the condition. If the condition evaluates to true, the program executes the commands; if not, it skips the section and goes on.

If there are several possibilities, that is, a multiple-choice situation, you use a construction that begins with DO CASE and ends with an ENDCASE statement. In either situation, after the program has performed the commands within the IF...ENDIF or DO CASE...ENDCASE construction, it continues on the line following ENDIF or ENDCASE.

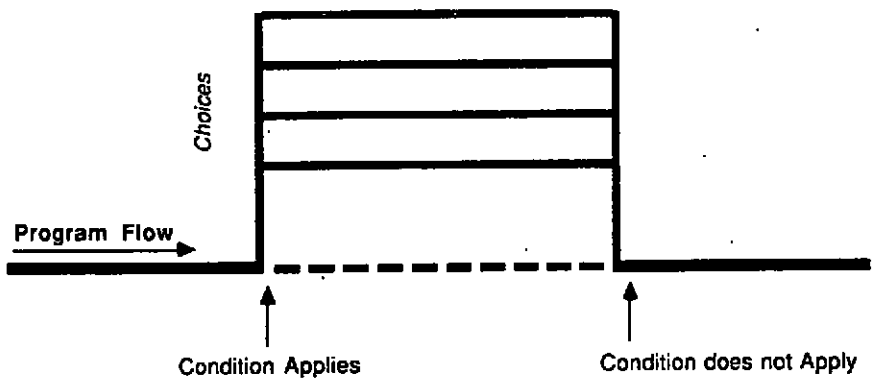


Figure 1-6 Multiple-choice conditions

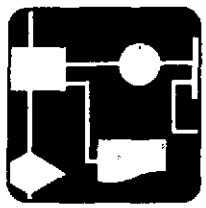
Here's the difference between a loop and dealing with conditions. In a DO WHILE loop, the program continues to test the condition at the end of each loop to see if it should perform another loop. The program can't leave the loop until the condition is no longer true. However, when it encounters an IF...ENDIF or DO CASE...ENDCASE construction, the program evaluates the condition only once.

Menus

Most dBASE programs start with a main menu, which gives the user a list of program choices. To see this in action, run the checkbook management program now. Start dBASE III PLUS and SET the DEFAULT drive. Make sure that the Sample Programs and Utilities disk containing the program is in the default drive. Then type:

```
. DO Cbmenu ↵
```

DO is the dBASE command that runs a program, and Cbmenu is the name of the main program file in the checkbook management system. You don't have to type the .prg extension with the DO command.



NOTE
This book uses upper case and lower case conventions to distinguish dBASE commands, in upper case, from filenames, in mixed case. However, you can type commands in lower case to save yourself unnecessary use of the Shift key. Although dBASE III PLUS requires only the first four letters of a command, your programs are much easier to read when you use the complete command.

After the screen clears, the first thing you see is the main menu, shown below:

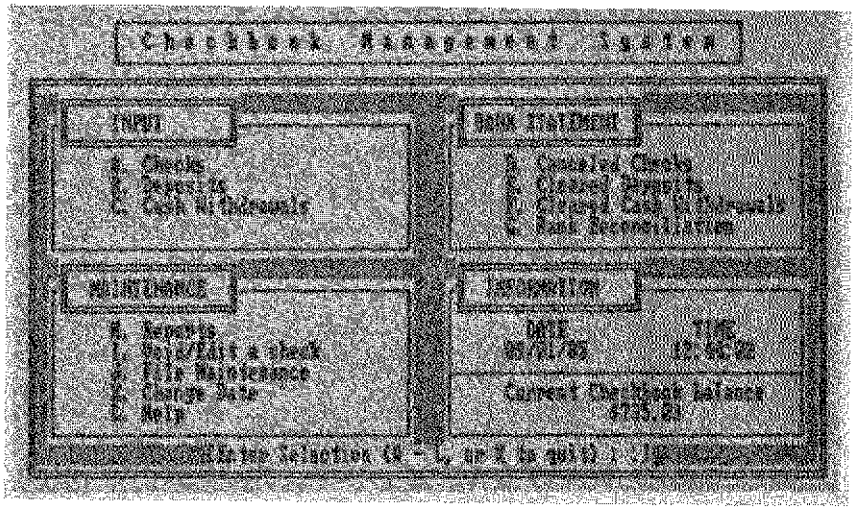
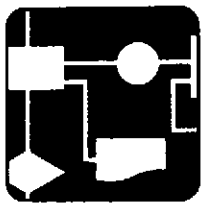


Figure 1-7 The checkbook management system main menu



Notice that the program gives the choices for balancing a checkbook. It also provides an out (X to quit) so that the user can leave the program easily. Always include a way to exit your program and return to the dot prompt.

When you choose an item from the main menu, the program branches to a subprogram that performs the task. The subprogram may contain even more choices in another menu, called a submenu. Type H now to look at the Reports submenu:

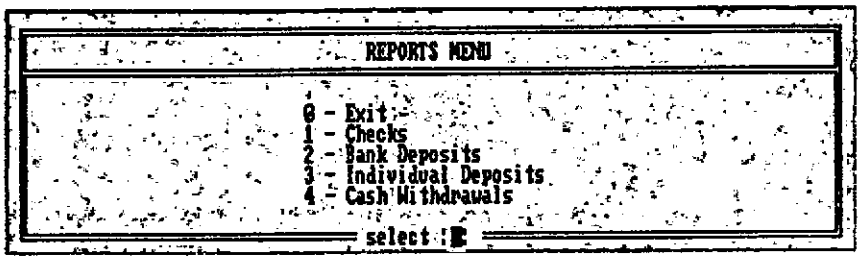


Figure 1-8 The Reports submenu of the checkbook management system

When you are finished, press 0 to return to the main menu.

One Entry, One Exit

The use of a main menu and submenus illustrates an important dBASE programming concept: one entry, one exit. That is, there is only one way to get to a submodule through the main menu, and there is only one exit returning to the main menu. In general, the main, or calling, program contains the main menu and governs all the other subprograms.

Take a look at the diagram of the checkbook management system, which is on the next page. There are subprograms that branch from the Cbmenu.prg file. Below these are other subprograms, such as Chkmask.prg, which branch from some of the subprograms above them. All subprograms return to their calling program, and the only way that the user can exit correctly from the entire program is through the main menu. This is tight control of program flow.

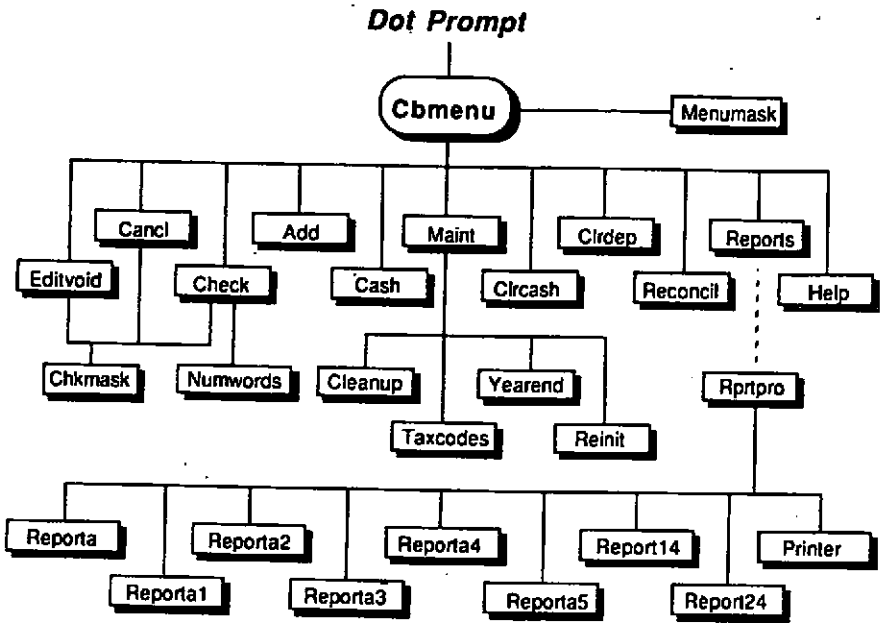


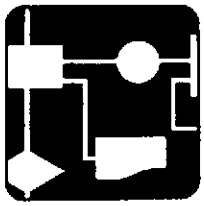
Figure 1-9 Diagram of the checkbook management system

Each box represents a program module in a separate program file. Although not shown here, each file ends with the extension .prg. Notice how the subprograms branch from the main program, Cbmenu.

Designing and Writing the Program

"Is that all there is to programming?" you may ask. Essentially, yes; the basic concepts are simple. However, just as you can't expect to create a culinary masterpiece merely by writing a recipe with the steps in the correct order, you can't write a program using only branching, looping, and conditions. You will find that the fine points of programming will take you longer to master.

Writing a program with the commands that the computer will understand is also known as coding the program. In Chapters 2 through 14, you'll learn how to code a dBASE program. This section provides some basic steps to follow.



Involve the User in the Design Phase

Never underestimate the importance of involving users in the design phase. When you've finished the program, you may have other projects, but the users continue to work with your program. If they have participated in the design phase, there is a better chance that the program will meet their needs.

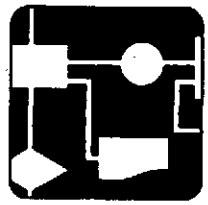
Some questions to ask potential users are:

- What do you want the program to do?
- Can the program meet your requirements?
- How do you want the program to work?
- Should the program contain on-line help screens?

Some questions you should resolve for yourself before starting to write the program are:

- Can you expand or enhance the program easily?
- How should you set up the database file or files?
- Where will the program get its input, from the keyboard or from database files?
- Will the program have to filter or change the input and output?
- What kind of screen forms or printed reports are needed?

You must decide exactly what your program is to do before you begin. Start with the kind of output you need, then decide how you want the program to produce this output. For example, if your program is to generate daily, monthly, and yearly reports, decide what information is to be in these reports and how this information will appear in print.



Top-Down Program Design

An important programming concept to remember is the top-down approach. You start with the general and gradually work down to the specifics. After you've identified exactly what the program does, write out the program design in English to clarify its overall structure. Don't worry about the actual code yet. It's more important to determine the correct steps in their proper order. For example, if you want to write a program that displays a record from a database file, you might write an outline like this one:

1. Set up the working environment.
2. Tell the user what the program does.
3. Get the name of the database and any related files, such as index files.
4. Open the database and related files.
5. Determine what record to display.
6. Determine what fields to display.
7. Display the desired fields in the record.
8. Ask if the user wants to display another record.
9. If so, repeat steps 5 through 7 above.
10. If not, close the database files and return to the dot prompt.

After deciding the overall program flow, identify the basic activities that the program performs. Then, gradually break each general activity into smaller and smaller units, or modules, like the subsections of a general outline. Each module should ultimately perform one specific task in the program. Think in terms of the smallest possible unit.

Some programmers use flow charts to organize a programming project visually. A flow chart is a diagram of how the various modules in a program interrelate. It's similar to a family tree. The flow chart doesn't have to be too elaborate; even a simple diagram will help clarify the program structure. Figure 1-7, the diagram of the checkbook management system, is a case in point.

Once you have divided a program's individual tasks into distinct units, you can write the program code for each unit separately. Because you are only writing one basic task at a time, it's much easier to write a program in this fashion. Breaking an entire programming project into logical units is called structured programming, or the modular approach. You outline, write, and test each modular unit individually. Once you have finished all the units, you can fit them together within the overall program design.

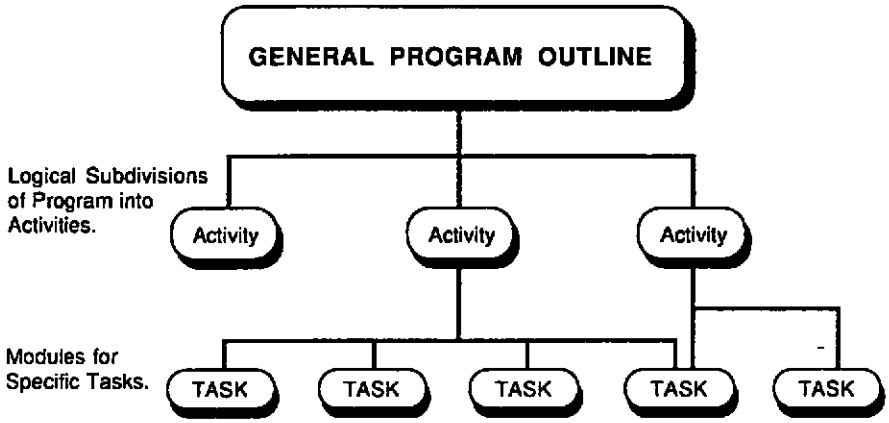
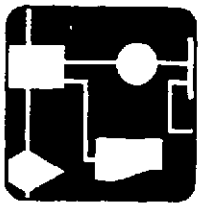


Figure 1-10 The top-down approach to program design

Begin with a general project outline, then gradually break it down into activities that logically go together. Finally, divide these into smaller and smaller tasks, eventually ending with individual modules that handle specific tasks. As illustrated, some specific task modules may be used in several different activities.

Meet the Programming Language Halfway

Once you've completed the breakdown of each module, write out what the module does in a combination of English and program code. This technique is known as pseudocoding. It allows you to clarify the necessary operations and the coding for each unit by bringing in some of the actual program commands and syntax. Because dBASE commands are English verbs, you actually are doing part of the real coding at the pseudocode stage.



Here's an example of a dBASE program module written in pseudocode. The dBASE commands are in upper case. Don't worry if you haven't learned some of these commands yet; just read them as English words. What does this module do?

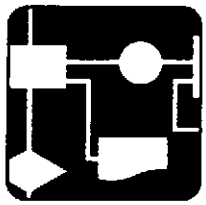
```
CLEAR the screen
USE the database file Names.dbf, which is INDEXed ON the
   Last_name field TO the index file Last.ndx
Show a screen form to ask for information
DO a loop WHILE the user wants to find a name
   GET the desired name from the user
   STORE the name in a temporary place (memory variable)
   FIND the name in the database file
   IF the name is found
      DISPLAY the record
   ELSE
      Give the user a message that the name was not found
   END of the IF construction
   ? Ask the user if another name is desired
   IF the user says yes
      LOOP back to the start of the DO clause and repeat it
   ELSE
      EXIT the loop and continue with the rest of the program
   END of the IF construction
END of the DO WHILE loop
```

Notice that indentations in the example clarify which lines go together. After you've written the program in pseudocode, you've written part of the real code already. When you have checked the pseudocode and are ready to write the program, it won't take long to convert the lines into correct dBASE instructions.

Don't Reinvent the Wheel

Besides simplifying a complex project, top-down and structured programming techniques allow you to reuse modules whenever you need them. You can even keep an entire library of modular routines at your service. Programmers often refer to this as a tool kit.

For example, if you write a module to display certain field names, you may find that you can use this module in many different programs. Once the module is written and tested, you can use it again and again.



Document Your Efforts

Documentation of program code and of entire programs is an extremely important part of program development. It's beneficial to supply adequate comments to yourself in the program code that describe what each step does. In dBASE, a comment line begins with the word NOTE or an asterisk (*). The dBASE III PLUS interpreter disregards comment lines when you run the program. Here are some sample comment lines:

```
* This line tells you something
NOTE This line tells you something
```

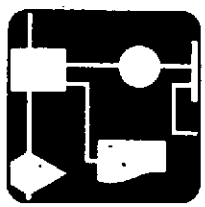
You cannot place a dBASE command after a comment on the same line, but you can supply a comment following a command by preceding the comment with &&. For example:

```
DO Accounts     && Run the accounting program module
```

NOTE
The && must be separated by at least one space from the actual command section on the line.

The checkbook management programs contain extensive comments about what the program is doing at each step of the way. Take a look at your printout of any program file to see what kind of information these comments contain. In addition, for every example of programming code mentioned in this book, there is a note in the program file indicating the chapter that describes the example.

Comments help to make the logic of your program code easier to follow. Later, you or someone else may need to modify the program code, or check for problems or bugs, and your original comments will be invaluable.



You should also document the entire programming project as you go along. Tell the user what the program will do, how it will do it, and whether your work is on schedule. When the program is finished, you can edit the documentation for a user's manual or reference guide to the program.

Allow for Future Program Expansion

Designing your program in modules allows you to expand it more easily later. For example, you can increase the number of choices in the main menu without disrupting the rest of the program. Just rewrite the menu program module, adding the new subprograms. Allow for future growth of all major programming projects right from the start. If users like your program, chances are they'll want it to do more tasks later.

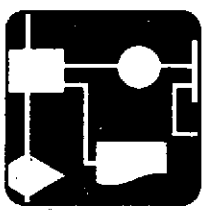
Test and Test Again

After you've written a module, test it thoroughly before you combine it with other modules into a complete program. This method isolates problems where they occur. When testing, try to crash the program — that is, do everything to make it fail. If you don't find any problems, your program may be fine. If you do find problems, fix them by debugging the program. Chapter 15 discusses how to test and debug your programs.

Even the most well-tested programs show problems, often months after the program is in use. When the entire program is complete, test it, and then test it again and again, until it seems to work correctly. Also, have others test it to see if they come up with problems that you've missed.

Put the Pieces Together

When all the individual modules are completed, reverse the top-down approach by gradually piecing together the modules within the overall program design. The end result will be your completed program.

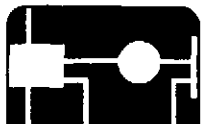


Summary: The Steps to Program Development

You now know some basic programming concepts and practices. You also know the steps to program development:

1. Determine what the program is to do.
2. Using the top-down approach, determine the general flow of the program. Divide the program into logical units, or modules, and continue breaking down the units until each module does only one specific task. Diagram your program design, if you wish.
3. Write each module separately; first in English, then in pseudocode, finally in dBASE code.
4. Document the project and your coding as you go along, not when you're finished.
5. Test each module independently of the others before assembling the final program.
6. Test and debug the final program thoroughly.

Chapter 2
dBASE PROGRAM STRUCTURE AND FLOW



In the previous chapter, you looked at program flow and the basic program control structures in brief. You can now investigate examples of dBASE program control in more depth. You should read this chapter in conjunction with the next, which continues the discussion of essential dBASE programming concepts.

What This Chapter Covers

This chapter discusses the following:

- The general makeup of dBASE programs, including the preamble, setup area, and body of the program, and how dBASE handles input and output
- How to use the four dBASE program flow constructions: DO, DO WHILE...ENDDO, IF...ENDIF, and DO CASE...ENDCASE
- How to nest constructions
- How to use the LOOP and EXIT commands

Preparing for This Chapter

You should have a working knowledge of dBASE III PLUS before beginning this chapter. If you are new to programming, first read Chapter 1.

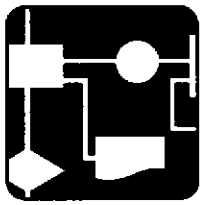
The Makeup of a dBASE Program

In general, a dBASE program has certain well-defined sections. Below is a brief overview of them.

The Preamble

The first section, the program header or preamble, contains information on the program's name, what it does, who wrote it, and an editing history. The editing history reveals when writing began and the last edit occurred.

Look at the example listing for any module of the checkbook management system to see the type of information included in the program header. Although there is no mandatory format for the program preamble, the format in the example program is the one that Ashton-Tate's own staff uses and recommends.



The Setup Area

A general setup area follows the preamble. This section determines the operating environment for the program. There are several important defaults for program files which the SET commands establish. A detailed discussion of these settings is in Chapter 4. The calling program generally sets the working environment, although occasionally a different setting is needed in a particular subprogram. Take a look at your printout of the Cbmenu.prg file. The setup area in this program starts with the command SET TALK OFF.

You might also need to specify the database relationships, that is, what database and index files the program uses, and initialize memory variables to store program input and output. Although you haven't had much experience with memory variables yet, in the next chapter you'll learn how important they are to effective dBASE programming.

The Body of the Program

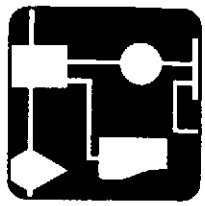
The body of the program contains commands that do the work of the program, such as getting input from the user, displaying information, changing database information, and producing output. Remember that programming gives you control mechanisms to interrupt program flow and call other program modules. You will investigate dBASE's program control commands in the following section.

The Closing Section

Every program requires some housekeeping. For instance, the program needs to make sure that all database files are properly closed to maintain database integrity. It must ensure that the standard defaults are reinstated before control returns at the end of the program to the dot prompt or operating system. Housekeeping is the subject of Chapter 13.

Leaving the Program

When you use RETURN in the main program module, it ends the program and returns you to the dot prompt, or the next highest level of program control. RETURN, however, does not close any database files USEd in the program. If you don't close the files USEd in your program, you risk corruption of your data. The only commands that close database files are CLOSE DATABASES, CLEAR ALL, USE, and QUIT. If you use QUIT, not only do you end the program and close database files in USE, but you also leave dBASE III PLUS and return to the operating system.



dBASE Program Flow

In the previous chapter, you saw that there were several ways to control program flow, and you learned the dBASE commands for branching, looping, and conditions. In this section, you will investigate these commands in more depth.

Do (Branching)

Besides using DO to run a program from the dot prompt, you also use the DO command from within a program to run a subprogram. The only way you can have a program branch to a subprogram is with DO, which represents an unconditional branch to the subprogram. When the subprogram is finished, the RETURN command, usually the last line in the subprogram, returns control to the calling program. The calling program then continues on the line following the DO command.

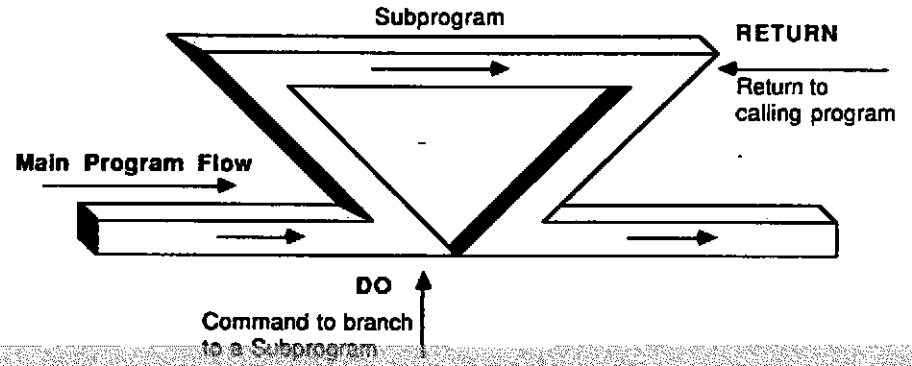
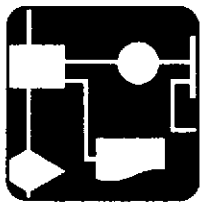


Figure 2-1 Using DO for branching

For example, a program may have this series of command lines:

```
CLEAR                  && Clear the screen
USE Names              && Open Names.dbf file
DO Listname            && Branch to Listname.prg
DO Newname             && Branch to Newname.prg
```



Listname is a separate program file, Listname.prg, which lists the names in the database file, Names.dbf. When this subprogram is finished, the RETURN command on the last line of Listname.prg sends program flow back to the next command in the calling program, which then branches again to another program called Newname.prg.

> **TIP**
> The RETURN TO MASTER command returns program control to the main, or top-level, program, no matter how many subprograms have been called. It's a good and quick way to bypass stepping back through several subprograms to get to the main program.

DO WHILE... ENDDO (Looping)

DO WHILE...ENDDO is the only dBASE construction for looping, repeating a series of actions while a condition is true. You can understand a DO WHILE...ENDDO loop as meaning DO such-and-such an action or another program WHILE such-and-such a condition is true. The DO WHILE command starts the loop, which continues until the condition is no longer true.

If you didn't tell the program where the loop ends, dBASE III PLUS would continue to repeat the commands following DO WHILE indefinitely until you turned off the machine. So, you must mark the end of every DO WHILE loop with an ENDDO command. When the condition in the DO WHILE command is no longer true, the program can leave the loop and continue with the next command following ENDDO.

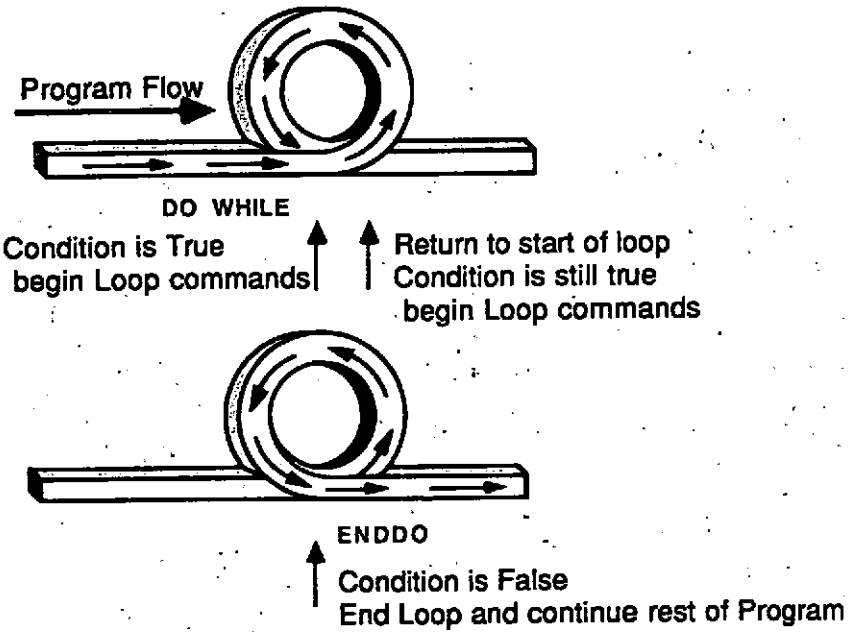
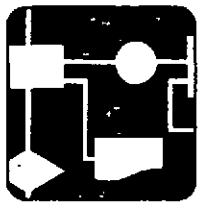


Figure 2-2 Using DO WHILE...ENDDO for looping

Here are two sample DO WHILE constructions:

```
* Do the loop as long as the last name field is not Smith
DO WHILE Lastname <> "Smith"
   * (commands)
SKIP
ENDDO

* Do the loop as long as the record pointer
* is not at the end-of-file
DO WHILE .NOT. EOF()
   * (commands)
SKIP
ENDDO
```



Note the use of the relational operator < >, not equal to, and the logical operator .NOT., as well as the end-of-file function, EOF().

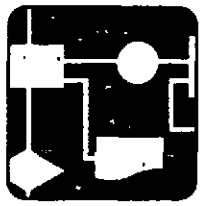
Don't confuse DO WHILE...ENDDO with DO; they are different commands. DO forces an unconditional branching to another program. DO WHILE...ENDDO makes the program loop as long as the condition stated in the DO WHILE command evaluates as true.

Here is a simple and complete DO WHILE...ENDDO loop. Note that the first command is outside the loop and not part of it:

```
* Position the pointer at the beginning of the file
GO TOP
* Do while the record pointer hasn't reached
* the end of the file
DO WHILE .NOT. EOF()
   * Show the first name, middle initial, & last name fields
   DISPLAY First, Middle, Last
   * Skip to the next record
   SKIP
ENDDO  && End of DO WHILE loop
```

This loop DISPLAYs the first name, middle initial, and last name field for each record in the database. It is very similar to issuing the command DISPLAY ALL First, Middle, Last from the dot prompt. When the record pointer reaches the end of the database file, the loop ends.

Indentations help you distinguish what sections go with which constructions. It is standard dBASE programming practice to use three spaces for each indentation. The indentations don't affect the running of the program.



IF...ENDIF (Conditions)

IF...ENDIF also relies on a logical condition, but the condition may or may not be present. The computer checks whether the condition is true only once at the beginning of the IF construction and performs the IF...ENDIF construction only once. You can think of an IF...ENDIF construction as IF such-and-such a condition is true, then perform such-and-such an action. Even though you don't type it in, the word *then* is assumed. Like DO WHILE...ENDDO, the IF construction must have a corresponding closing command, ENDIF.

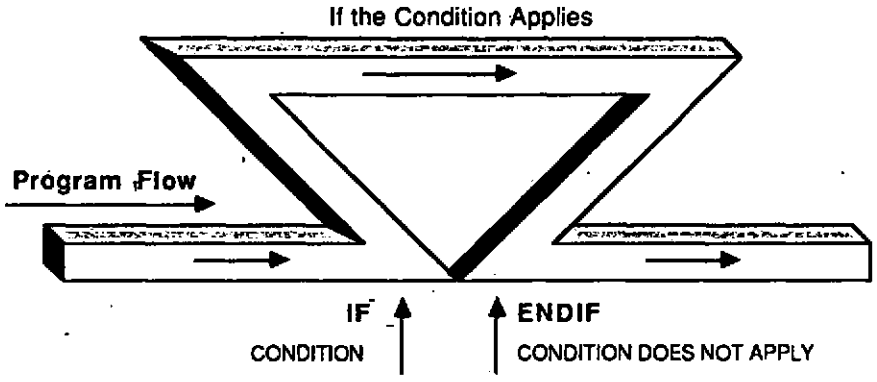
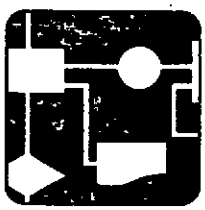


Figure 2-3 Using IF...ENDIF for conditions

Note the distinction between DO WHILE...ENDDO and IF...ENDIF. DO WHILE...ENDDO *forces* dBASE III PLUS to continue repeating the loop WHILE the condition is true, while IF...ENDIF *asks* dBASE III PLUS to decide if a specific condition is true only once and performs the commands in the IF...ENDIF construction only once.

```
IF Balance = 0.00          && If there's no money in your
   ?"You're broke"         && account, you get the message
ENDIF
```



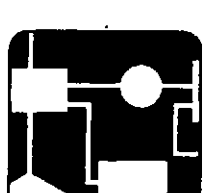
You can also expand an IF...ENDIF construction to make the program choose between two possibilities: if A is true, do B, else do C. So there can be an ELSE command, which governs the other possibility. You can have only one ELSE for every IF...ENDIF section. ELSE must be on a separate line. Here is another simple IF...ENDIF construction. What does it do?

```
IF Last # "Smith"
   DISPLAY First, Middle, Last
ELSE
   SKIP
ENDIF
```

Answer: If the field called Last does not equal the name Smith, the program DISPLAYs the First, Middle, and Last fields. If the Last field does contain Smith, the program SKIPs to the next record. Note the use of the relational operator #, which is equivalent to <>, that is, not equal to. If there is no ELSE statement, the program continues to the command after the ENDIF, when the IF condition is not true:

```
IF Amount > 100.00
   DO Inflation
ENDIF
```

The above IF...ENDIF construction makes the program branch to the Inflation.prg module only if the Amount field is greater than $100.00. Notice that you can include or nest a DO clause within another construction. You'll learn more about nesting in another section.



TIP
Be careful when you supply numeric ranges. In the above example, dBASE III PLUS skips over all Amount fields that are 100.00 or less. However, it doesn't skip over the amount 100.01. You could include 100.00 in the condition in either of two ways:

```
IF Amount > 99.99    && Amount is greater than 99.99
   * (commands)
ENDIF
```

or

```
IF Amount >= 100.00 && Amount is greater than or equal to 100.00
   * (commands)
ENDIF
```

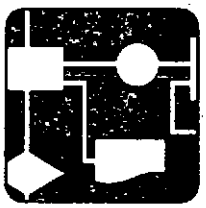
Not being specific about numeric ranges causes needless problems. Avoid this type of trap.

Expanding DO WHILE... ENDDO and IF...ENDIF

DO WHILE...ENDDO and IF...ELSE...ENDIF constructions can evaluate more than one condition. You could rewrite the IF...ENDIF example to evaluate the Amount field as being over 100.00 but under 1,000.00:

```
IF Amount > 100.00 .AND. Amount < 1000.00
   * (commands)
ENDIF
```

Although you don't have to repeat the IF, you must repeat the field name or memory variable in each part of a complex clause, even if the field or variable name is the same. You can construct more complicated DO WHILE constructions, too:




```
* Do the loop as long as the variable choice
* is less than 0 or greater than 9
DO WHILE choice < 0 .OR. select > 9
   * (commands)
ENDDO
```

Be careful that you use the correct logical operator. Remember that .AND. implies both one condition and the other, while .OR. signifies either the first or the second condition. Refer to *Using dBASE III PLUS* for a discussion of logical operators.

You can also mix two conditions. In this example, Past_due is a logical field, so the IF line means: if the amount is greater than 100.00 and Past_due is true:

```
IF Amount > 100.00 .AND. Past_due
   * (commands)
ENDIF
```

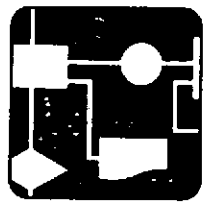
Similarly, if the amount is greater than 100.00, but Past_due is false:

```
IF Amount > 100.00 .AND. .NOT. Past_due
   * (commands)
ENDIF
```

Because Past_due is a logical field, you can't say Past_due = .T. or, alternatively, Past_due = .F. Note that when you use more than one logical operator, you must have a space between them so that dBASE can properly execute the command: .AND. .NOT.

DO CASE... ENDCASE for Multiple Choices

IF...ELSE...ENDIF is useful for having the program branch depending on one or two conditions: if A is true, do B, else do C. When the program provides a series of possible conditions, it's much simpler to use the DO CASE...ENDCASE construction. This is just a list of the possible choices and what to do for each — a multiple choice situation.

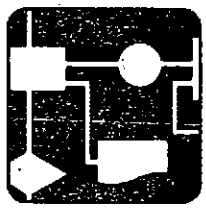


Because DO CASE alerts dBASE III PLUS that possibilities follow, the DO CASE line contains only the command. Each possibility appears on a separate line following the word CASE. Each different choice must follow the word CASE. An ENDCASE line completes the construction.

Figure 2-4 Using DO CASE...ENDCASE for multiple choices

The most frequent use of the DO CASE command is governing a menu where the user has one of several options. For example, in the checkbook management system main menu, the user has the choice of letters A through L, and X. The program efficiently handles these multiple choices with a DO CASE construction. A discussion of the main menu choices is covered in Chapter 4.

You can combine all other possible choices that don't need their own CASE command by using OTHERWISE in the DO CASE...ENDCASE clause. You can only have one OTHERWISE for each DO CASE. The following is a typical DO CASE construction. Each choice calls a different subprogram module:



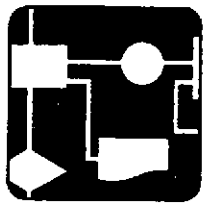

```
* Determine what action to take on a choice
DO CASE
   * The choice is A
   CASE choice ="A"
      DO Accounts          && Run accounting program
   * The choice is B etc.
   CASE choice ="B"
      DO Taxes             && Run taxes program
   CASE choice ="C"
      DO Reports           && Run reports program
   CASE choice ="D"
      DO Leave             && Run exit program
   * The choice is something other than A, B, C, or D
   OTHERWISE
      DO Warning           && Run error message program
ENDCASE
```

DO CASE is very similar to IF...ENDIF. Both accomplish the same thing, but DO CASE usually determines several possible conditions, while IF...ENDIF governs one or two possible conditions. Soon you'll see how elegant a DO CASE construction can be when compared to IF...ENDIF.

Nesting

Each of the four basic control structures can work within others. This is known as nesting. For example, here is a DO CASE...ENDCASE construction nested within an IF...ENDIF clause:

```
* If the condition is true
IF Last_name = "Smith"
   DO CASE
      CASE First_name = "Joe"
         * (commands)
      CASE First_name = "Mary"
         * (commands)
      OTHERWISE
         * (commands)
   ENDCASE
ENDIF
```



Take care that the ending command lines are in the correct order. Each nested construction is an inviolable whole. Think of them as a series of mixing bowls, one sitting inside another.

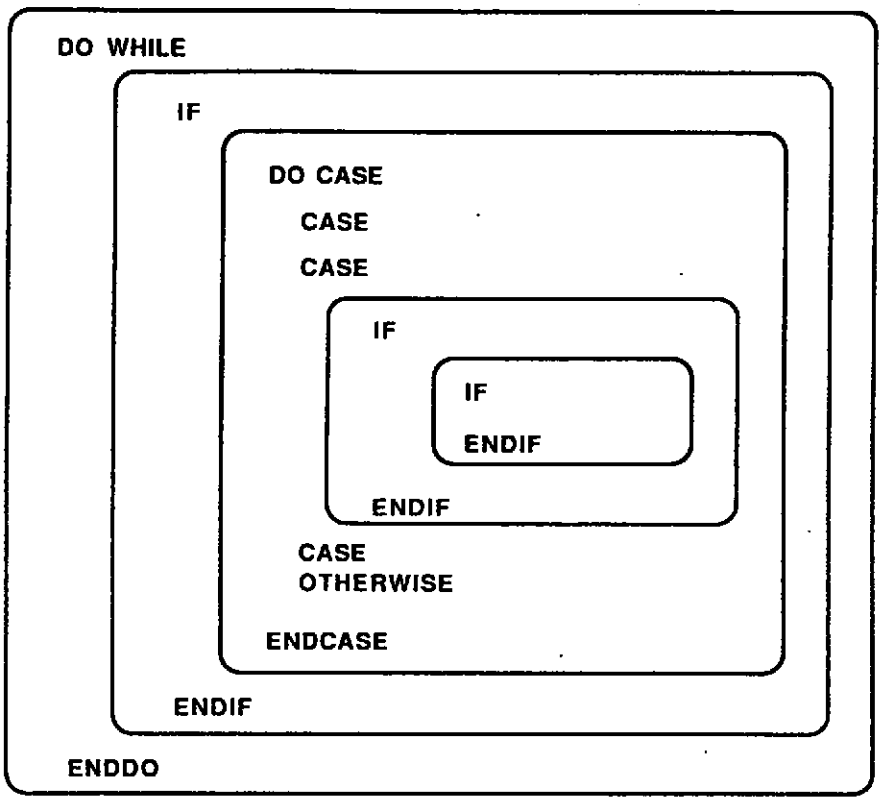
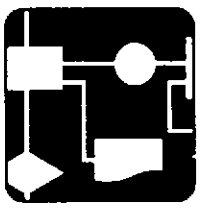


Figure 2-5 Nesting constructions



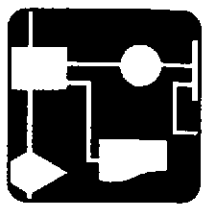
Here is an incorrect construction. Why?

```
DO CASE
   CASE selection = "1"
      DO <subprogram>
   CASE selection = "2"
      IF Balance = 0
        ? "You're broke"
      ELSE
        ? "You're still in the black"
        * (commands)
   OTHERWISE
      * (commands)
   ENDCASE
ENDIF
```

Because the IF is nested within the DO CASE, you must close the IF construction before closing the DO CASE construction. The correct form of this short module is:

```
DO CASE
   CASE selection = "1"
      DO <subprogram>
   CASE selection = "2"
      IF Balance = 0
        ? "You're broke"
      ELSE
        ? "You're still in the black"
        * (commands)
      ENDIF
   OTHERWISE
      * (commands)
ENDCASE
```

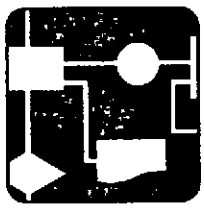
Using indentations is extremely helpful when you have nested clauses. Indenting makes the code more readable and lets you determine at a glance whether control structures are properly terminated.



TIP
Always check that each beginning DO WHILE, IF, and DO CASE construction has a corresponding ENDDO, ENDIF, or ENDCASE, respectively. Make sure that there are no superfluous ENDDO, ENDIF, or ENDCASE lines, and that ENDDO, ENDIF, and ENDCASE are spelled as one word.

Another way to avoid confusion when you have nested clauses is to repeat the condition on the end line. This allows you to check visually which DOs go with which ENDDOs. dBASE III PLUS disregards anything else past the end command. You could thus write the previous DO WHILE example like this:

```
* Position the pointer at the beginning of the file
GO TOP
* Do while the pointer hasn't reached the end of the file
DO WHILE .NOT. EOF()
   * List the first name, middle initial, & last name fields
   LIST First, Middle, Last
   * Skip to the next record
   SKIP
ENDDO && WHILE .NOT. EOF()
CLEAR
```

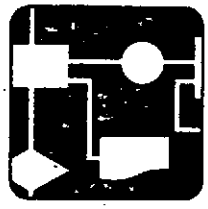


Another Way to Loop

Sometimes you may wish to leave an IF...ENDIF or DO CASE...ENDCASE construction nested within a DO WHILE...ENDDO loop and return to the beginning of the loop. Use the LOOP command, but be careful. LOOP immediately interrupts the flow of the program and takes the program back to the beginning of the most recent DO WHILE. Here's an example of LOOP:

```
* Loop as long as the record pointer is not at the
* end of the file
DO WHILE .NOT. EOF()
   * If the record is already marked for deletion
   IF DELETED()
      * Move to the next record and start again
      SKIP
      LOOP
   ELSE
      * (Many more commands)
   ENDIF && DELETED()
ENDDO && WHILE .NOT. EOF()
```

The test for deleted records is put at the beginning of this lengthy DO WHILE loop so that the program doesn't have to evaluate the rest of the commands in the loop. The LOOP command has no meaning unless it occurs in an IF...ELSE...ENDIF or DO CASE...ENDCASE construction, which is nested within a DO WHILE...ENDDO construction.



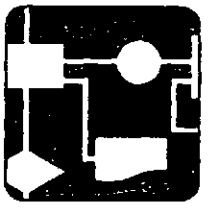
How to Get Out of a Loop

Occasionally, you will need to leave a DO WHILE...ENDDO loop earlier than the program expects, without performing the rest of the commands in the loop. You can do this with the EXIT command. Here is the previous nesting example rewritten with an EXIT command:

```
DO WHILE .T.
   DO CASE
      CASE selection = "1"
         DO <subprogram>
      CASE selection = "2"
         * If your balance is 0
         IF Balance = 0
            ? "You're broke"
         ELSE
            * Get out of the loop
            EXIT
         ENDIF
      OTHERWISE
         * (commands)
   ENDCASE
ENDDO && WHILE .T.
```

EXIT doesn't cancel the program. It merely causes the program to leave the current DO WHILE...ENDDO loop. The program then continues with the next command after the ENDDO line.

Chapter 3
USING MEMORY VARIABLES



dBASE programming frequently needs to hold information temporarily in memory to control a program. This information is stored in what dBASE calls memory variables. It is strongly suggested that you thoroughly understand how memory variables work in programs before you begin programming in dBASE.

What This Chapter Covers

This chapter discusses the following:

- How to set up memory variables in dBASE programs
- The different types of memory variables
- How to use memory variables in programs
- How to declare memory variables PUBLIC or PRIVATE
- How to use memory variables as program flow controls

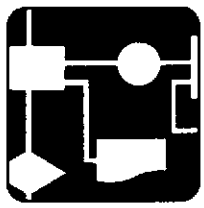
Preparing for This Chapter

You should have a basic understanding of general programming concepts and the dBASE control structures before reading this chapter.

Memory Variables Explained

A memory variable is a temporary storage location for information. You identify a memory variable by its name, which can be up to ten characters in length. A memory variable can contain numbers, character strings, a date, or even a logical expression, .T. or .F. When you wish to use the information in the memory variable, you refer to the memory variable by its name, not by its contents. The contents of a memory variable can change during the course of the program, but its name remains the same.

Each variable takes up a certain number of bytes in memory, depending on the type of variable. dBASE III PLUS assigns to character-string memory variables the length of the assigned string plus 2 bytes, to numeric and date variables a length of 9, and to logical variables a length of 2.



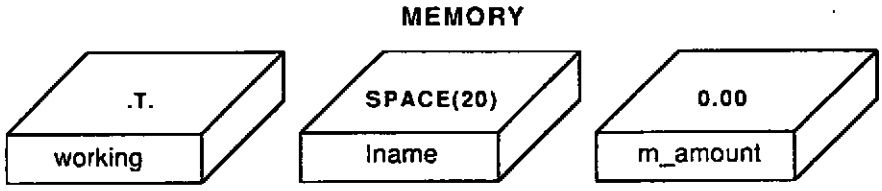


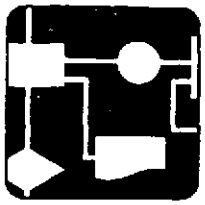
Figure 3-1 Memory variables

NOTE
There are no memo type memory variables.

Initializing Memory Variables

You must create, or initialize, a memory variable before you can use it. You do this by STOREing data in the variable. Whenever you initialize a memory variable, dBASE III PLUS automatically assigns the variable type according to the information you've STOREd in it. You initialize the memory variable's name and its contents at the same time.

There are two ways to initialize memory variables. One is using the STORE command: its name reminds you that you are temporarily storing information in a memory variable. The second method is typing the memory variable name first, followed by an equals sign and the information to be STOREd in it. Below are some examples.



Logical Type Variables

The following command creates a memory variable named *working* and puts the logical value true, .T., in it.

```
STORE .T. TO working
```

This next example is the equivalent of the previous example:

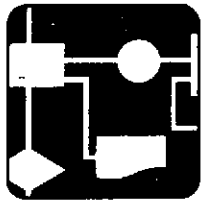
```
working = .T.
```

The information on the right side of the equals sign becomes the contents of the memory variable on the left side. If you are familiar with programming, you realize that this method is similar to the way other programming languages initialize variables.

> **TIP**
> The dBASE III PLUS commands and functions, such as CONTINUE or DELETED(), are reserved words. Because using reserved words for memory variable names could cause confusion, be sure to pick other names for variables.

To change the contents of a memory variable, reinitialize it. For example:

```
STORE .F. TO working && The variable working now contains .F.
```



Character Type Variables

To initialize a character type memory variable, you must enclose the character string in a delimiter, that is, single or double quotes, or square brackets:

```
STORE "Vincent" TO mfname
```

This creates a character type memory variable called *mfname* and STOREs the string **Vincent** in it. The length of this variable is 7. If the character string already contains a standard delimiter character, use a different set of delimiters. For example, if a single quote is in the string, enclose the entire string in double quotes or square brackets:

```
STORE "That's incorrect ... try again" TO mprompt1
```

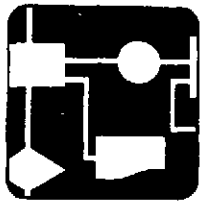
In your programs, you will often wish to initialize character memory variables that contain nothing but blanks. A quick way to do this is with the SPACE() function. For example, suppose you need a memory variable called *m_first* to temporarily hold the user's input for the actual First_name field in a database file. This field is 20 characters long. Initialize the *m_first* variable like this:

```
STORE SPACE(20) TO m_first
```

Date Type Variables

You can create a date type memory variable with the DATE() function, but it only STOREs today's date in the memory variable; assuming, of course, that you entered that date when you started your computer. The command would be:

```
STORE DATE() TO today
```



To STORE another date in a memory variable, you employ the character to date conversion function, CTOD(). You'll learn more about this function in Chapter 5. The following command initializes a date type variable called *birthday*:

```
birthday = CTOD("04/26/85")
```

To initialize a blank date to *birthday*, use:

```
birthday = CTOD("  /  /  ")
```

Numeric Type Variables

To initialize a numeric type memory variable, make sure that you include the correct number of decimal places, if any. Otherwise, dBASE III PLUS assumes that the memory variable only contains integers. Thus:

```
STORE 0 TO number
```

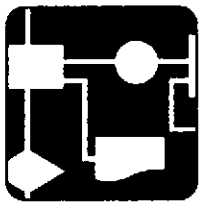
creates a new numeric memory variable called *number* and STOREs 0 to it. dBASE III PLUS only allows integers to be in this memory variable. However,

```
STORE 0.00 TO number
```

creates a numeric memory variable with two possible decimal places.

You can initialize several memory variables of the same type and length in one line, with each variable separated by a comma. This example is from the Add.prg module:

```
* initialize memory variables
STORE 0.00 TO subtotal,totals,cash
```



TIP
A standard convention in this book is for memory variables to be shown in lower case italics. This distinguishes them from database filenames and field names, which by convention are written with initial capitals.

How you name memory variables is up to you, but a good approach is to choose a name that describes what the memory variable does. It's possible to have a memory variable called *x*, but later, when you're reading your program, it may be difficult to remember what *x* represents. So use a descriptive name:

```
STORE 129.50 TO mcost
```

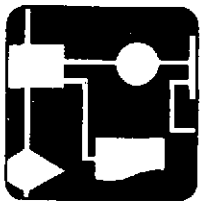
Then you can readily remember what this line means:

```
REPLACE Amount WITH mcost * 1.05
```

Many programmers use the letter *m* or the characters *m_* to begin each memory variable name when there may be confusion with an actual field name. For example, if there is a field called Last_name, a related memory variable could be called *m_last*. This is a good way to show the relationship between fields and memory variables. In Chapter 11, you'll see how dBASE III PLUS can distinguish between field names and variable names with the same name using the M-> feature.

Limitations

You can have a total of 6,000 bytes of information stored in up to 256 different memory variables. This is more than adequate for all but the largest of programs. If necessary, you can change the amount of memory for variables with the MVARSIZE parameter in the Config.db file. See *Using dBASE III PLUS* for more information.



How Memory Variables Work in Programs

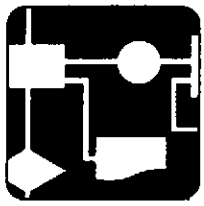
Because they allow the programmer to control input and output, memory variables are important for programming. The program initializes a memory variable to hold the user's input. When the user has supplied the information, the program STOREs this input into the variable and asks the user to validate the information.

This method ensures database integrity by not changing the database immediately. When the user confirms that the information is correct, the program REPLACEs the current field information with the contents of the memory variable. The memory variable can then be cleared and used again for the same task.

For example, if the task at hand is to update information in the Last_name field, here's how to set it up:

1. Initialize a character-type memory variable called, for example, *m_last*, to be the same length as the Last_name field. Generally, you initialize memory variables of the exact type and length as their related database fields.
2. Ask the user to supply the new information in an on-screen form.
3. Temporarily STORE the user's input TO *m_last*.
4. Display what the user has typed (that is, the contents of *m_last*) and ask the user to confirm that the information is correct.
5. If the information is incorrect, give the user a chance to correct it.
6. After the user confirms that the information is correct, use the REPLACE command to replace the current information in the Last_name field with the contents of the *m_last* memory variable. Repeat the above steps for the next name.
7. When the user indicates that there are no more names to be entered, close the database file.

The above steps are a general pattern to follow. You'll see these steps in action when you learn how to design screen forms in Chapters 6, 7, and 8, and how to evaluate and use input, Chapter 9.



PUBLIC and PRIVATE Variables

In dBASE III PLUS, there are two classifications of memory variables: public and private. You declare the status of a variable with the PUBLIC and PRIVATE commands. You can use a PUBLIC variable in all program modules, no matter where you declare the variable PUBLIC. A PRIVATE variable is in effect only in the current program or subprogram, and all programs that depend on it. The terms PUBLIC and PRIVATE variables are often called global and local variables, respectively, in other programming languages.

PRIVATE Variables

In dBASE programs, memory variables are PRIVATE unless you declare them PUBLIC. So, for a PRIVATE variable, you merely initialize the variable. For example, the following variable is initialized in a subprogram:

```
m_cost = 0.00
```

The *m_cost* variable is PRIVATE to that subprogram and all programs called by the subprogram. dBASE III PLUS releases the variable when program control RETURNs to the calling program from the subprogram in which the variable was initialized as PRIVATE.

The checkbook management program contains many PRIVATE variables in subprograms that work in those subprograms only. When a variable is PRIVATE to a subprogram, its contents won't cause confusion with the same PRIVATE variable in an unrelated subprogram. So, you can use standard names for variables that do the same thing in each subprogram.

PUBLIC Variables

Setting up a PUBLIC variable is a two-step process: (1) declaring the variable PUBLIC, and (2) initializing the variable. Here's an example:

```
PUBLIC balance
balance = 0.00
```

This declares the variable *balance* as a PUBLIC variable and initializes it as a numeric type with a contents of 0.00.
dBASE III PLUS never clears PUBLIC variables unless explicitly told to do so.

Here is an illustration of PUBLIC and PRIVATE variables:

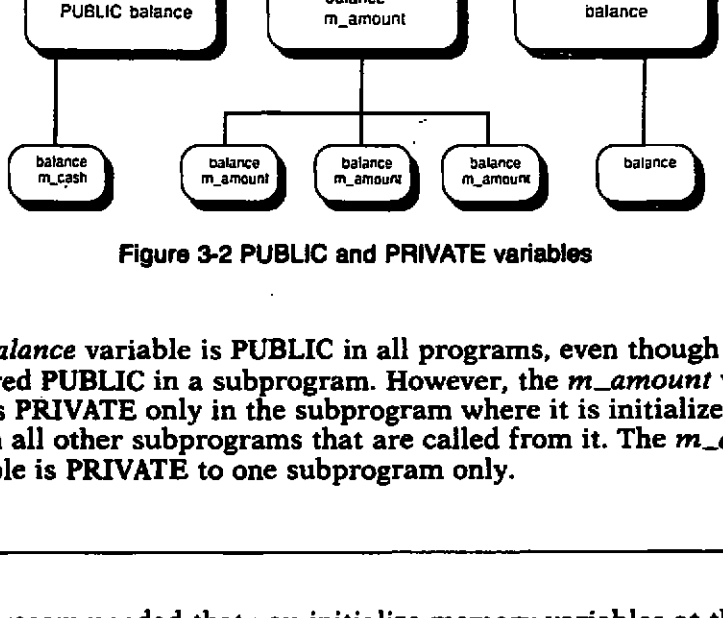
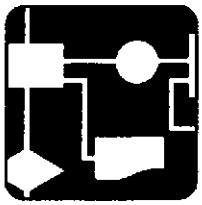


Figure 3-2 PUBLIC and PRIVATE variables

The *balance* variable is PUBLIC in all programs, even though it's declared PUBLIC in a subprogram. However, the *m_amount* variable is PRIVATE only in the subprogram where it is initialized and in all other subprograms that are called from it. The *m_cash* variable is PRIVATE to one subprogram only.

TIP
It's recommended that you initialize memory variables at the beginning of a program in the main program file if you use the variables throughout the entire program. You can then locate these variables quickly. Because the main program is the highest-level program, PRIVATE variables initialized in the main program remain in effect in all modules and thus act like PUBLIC variables.



Getting Rid of Memory Variables

All PRIVATE memory variables disappear when a program finishes or when the subprogram that initialized the variable RETURNs to the calling program. There is another way to remove a PRIVATE memory variable from memory before the program or subprogram ends. To do this, use the RELEASE command:

```
RELEASE mcost
```

You can also RELEASE certain PRIVATE memory variables and retain others by using a wildcard, if the variables have similar names:

```
RELEASE ALL LIKE m*
```

The * is a wildcard that tells dBASE III PLUS to RELEASE ALL memory variables that begin with *m* no matter what other letters are in their names. You can also RELEASE some memory variables with the exception of others of similar names. Thus,

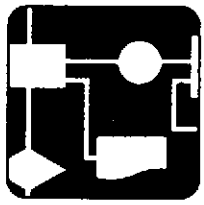
```
RELEASE ALL EXCEPT m*
```

would RELEASE the variables that don't begin with the letter *m*.

PUBLIC variables are never released automatically. The only way to remove PUBLIC variables from memory is with the one of these commands: CLEAR MEMORY, CLEAR ALL, or RELEASE <public variable list>. Because you might inadvertently clear variables that you want to retain, be careful when using CLEAR MEMORY or CLEAR ALL.

For example, to clear the two PUBLIC variables *mcost* and *mbalance* from memory without clearing the other variables, use this command:

```
RELEASE mcost, mbalance
```



A PUBLIC variable can be PUBLIC to the entire system as well as to the dot prompt and to programs that aren't even necessarily related. So, take care when working with several programs that you CLEAR MEMORY to remove unnecessary PUBLIC variables. In Chapter 16, you'll see how to temporarily hide a PUBLIC variable.

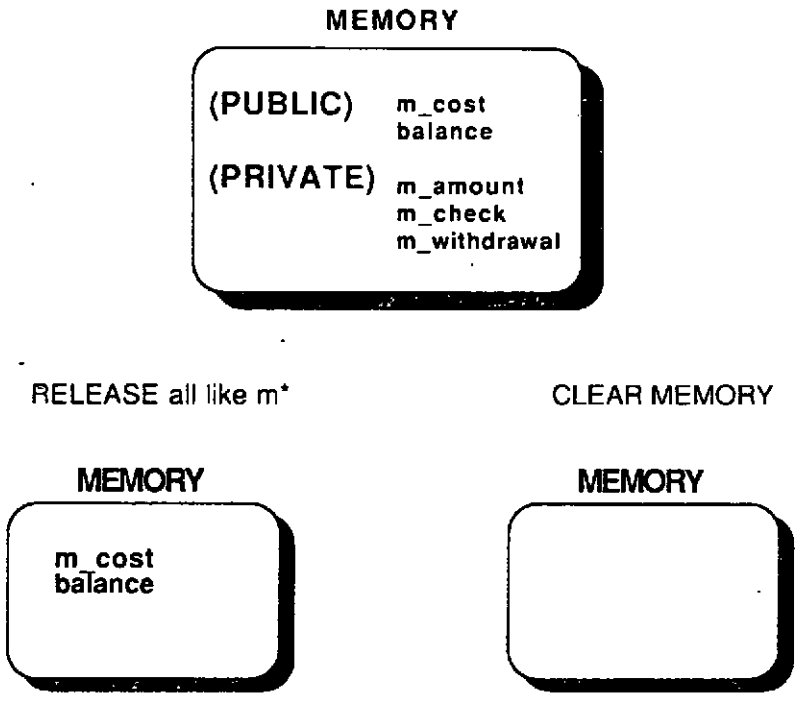
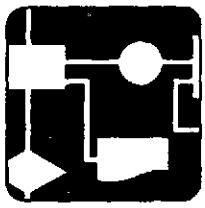


Figure 3-3 How to RELEASE memory variables

Memory Files

You can reuse variables by setting them up in memory files. The SAVE command SAVEs the contents of memory in a memory file. Memory files have the extension .mem. When you want to use these memory variables, issue the RESTORE command to bring the variables into active memory from the memory file.



Setting Up Memory Files

To set up a memory file, initialize at the dot prompt the memory variables that you want in the file. When you have them all in memory, issue the SAVE command with a memory filename. For example, you have several variables in memory and you want to SAVE them to a memory file called Setup. At the dot prompt, you type SAVE TO Setup ↵.

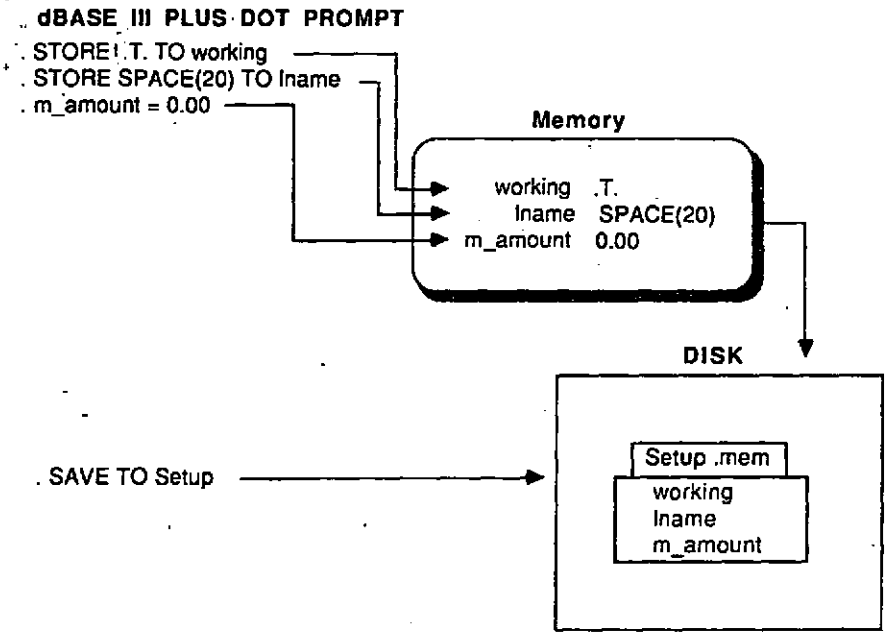
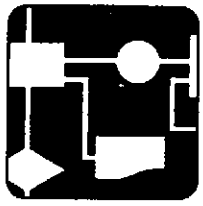


Figure 3-4 Setting up a memory file

Restoring Memory Files

To return the contents of this file to active memory, type RESTORE FROM Setup ↵. Both SAVE and RESTORE assume the file extension .mem. However, whenever you RESTORE memory variables from a file, you automatically RELEASE all the variables currently in memory unless you instruct dBASE III PLUS to retain them. This is done with the ADDITIVE expression.



As its name implies, ADDITIVE adds the contents of the RESTOREd file to the variables currently in memory. For example, if the program is to RESTORE the memory variables that are in a memory file called Chkbook.mem but not RELEASE the other variables currently in memory, the command would be:

```
RESTORE FROM Chkbook ADDITIVE
```

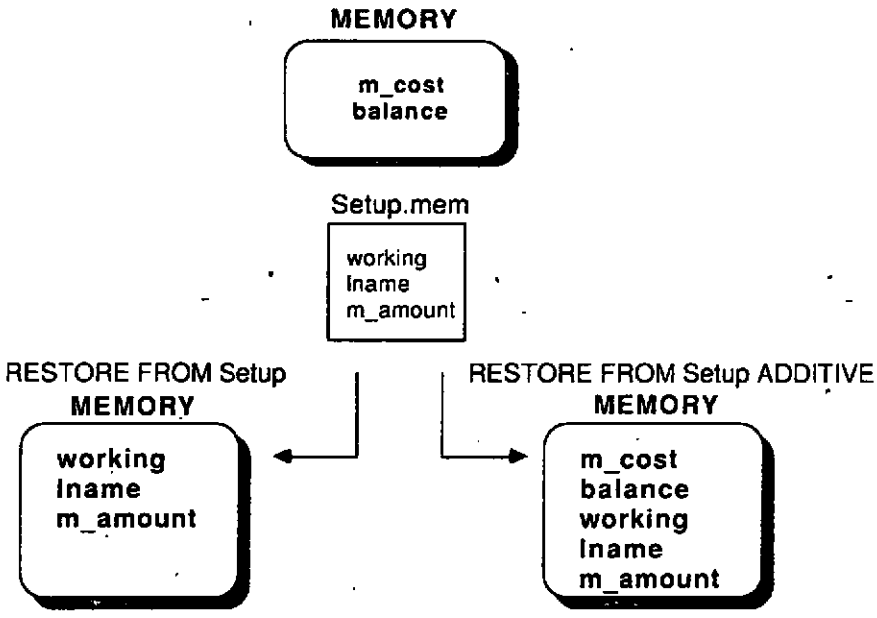
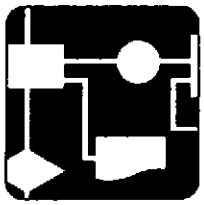


Figure 3-5 Using RESTORE and what happens to memory



NOTE
The PUBLIC/PRIVATE status of a variable is not saved to the .mem file. If you RESTORE a memory file at the dot prompt, the variables come back as PUBLIC. If you RESTORE a memory file within a program file, the variables come back as PRIVATE. Remember that any memory variables initialized at the beginning of the program in the main program file remain in effect in all modules and subprograms.

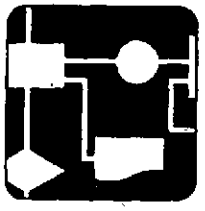
To get variables to RESTORE as PUBLIC in a subprogram file, you need to do a PUBLIC <memvar list> before doing the RESTORE FROM. You need to use the ADDITIVE option to retain variables already in memory.

For example, a memory file, Setup.mem, contains three variables: *mcost*, *mbalance*, and *mamount*. To RESTORE this file during a program run and make these three variables PUBLIC to the program, enter these commands in the program:

```
PUBLIC mcost, mbalance, mamount
RESTORE FROM Setup ADDITIVE
```

Using Memory Files

Programmers often use memory files both to initialize memory variables at the beginning of the program and to reinitialize them. This is another way to clear the contents of a memory variable. It is possible to have several memory files that the program can individually RESTORE whenever it needs them. The advantage to this approach is that you can RESTORE the original contents of certain memory variables without affecting others.



TIP
One benefit to using a .mem file is that you can add, change, or delete memory variables if necessary during the course of program development. For instance, if you forget to include a variable, you can add it later to the Setup.mem file. At the dot prompt, RESTORE the memory variable file. Then STORE the new variable into memory. Finally, SAVE the file back to disk. The SAVEd file will contain the previous memory variables and the new one, too.

Logical Memory Variables as Program Flow Controls

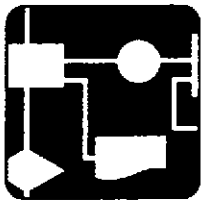
You can use logical memory variables as controls, which is similar to using logical fields to help you retrieve information. For example, a Christmas card database file may have a logical field called Last_year, which indicates which people sent you Christmas cards (that is, Last_year = .T.). When you prepare this year's mailing list, you could check which people might not get cards, because they didn't send you a card last year:

```
LIST FOR .NOT. Last_year
```

The logical field allows you to isolate records quickly on a given true/false, yes/no basis. Because program flow also relies a great deal on true/false conditions, you can use logical memory variables in a similar fashion. One advantage to using logical memory variables for program flow is that they can indicate by their name exactly what is going on.

For example, suppose your program is to add records to a database file. The program initializes a logical type memory variable called *adding*:

```
adding = .T.
```



You can then use this variable in a loop or condition:

```
DO WHILE adding
   * (commands)
ENDDO WHILE adding
```

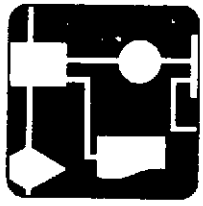
When you study this module later, you'll understand at a glance what the DO WHILE...ENDDO loop does.

NOTE
You can use the & function with a variable in the conditional part of a DO WHILE loop only if the value of the variable does not change during the run of the loop. dBASE III PLUS evaluates the DO WHILE condition once only, at the beginning of the loop. After the first time, it executes the loop from memory. Here is an example:

```
STORE "Lastname = 'Jones'" TO condition
DO WHILE &condition .AND. .NOT. EOF()
   * (commands)
ENDDO
```

The loop runs as expected, because the value of the variable *condition* doesn't change during the run of the loop. However, changing the value of *condition* inside the DO WHILE loop may result in an infinite loop. See *Using dBASE III PLUS* for more information on macro substitution.

Chapter 4
SETTING UP THE MAIN PROGRAM



You are now ready to take a look at the different activities or tasks that a dBASE program can do. Each of the following chapters covers related topics. If you are new to programming, study the chapters in order. If you are already an experienced programmer, you can skip to the topics that interest you.

What This Chapter Covers

Using the checkbook management system as an example of a dBASE main program module, you'll learn that the main program generally does the following:

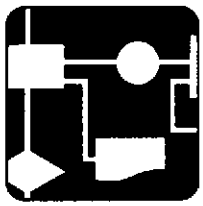
- Closes previously opened database files
- Establishes the program's working environment
- Initializes memory variables
- Contains a control structure that works for the entire program, usually a DO WHILE...ENDDO loop
- Presents the main menu to users
- Contains a control structure that handles the menu choices, usually a DO CASE...ENDCASE structure
- Closes database files and resets the working environment before the program RETURNs to the dot prompt

Getting Ready to Do This Chapter

Understand the basics of programming, how the dBASE program control structures work, and what memory variables are before reading this chapter. Start dBASE III PLUS and place the Sample Programs and Utilities disk in the default disk drive. Take a look at the diagram of the checkbook management system in Chapter 1, and have your printout of the Cbmenu.prg file handy. This is the main program module for the checkbook management system.

What the Main Program Does

The main module of any dBASE program controls the program as a whole. It calls and runs the subprograms and establishes the working environment for the entire program. It usually presents the user with the main menu and contains a controlling structure, such as a DO WHILE...ENDDO loop, which determines the entire program flow.



The main program may also do housekeeping, ensuring that all database files are closed properly at the beginning and at the end of the program, before the user returns to the dot prompt.

The Setup Area

You saw in Chapter 2 that all dBASE programs start with a preamble and a setup area. You have already seen an example of the program preamble. This section discusses at greater length what goes in the setup area.

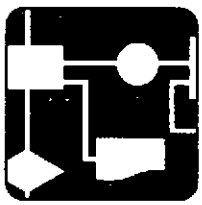
Closing Database Files in Use

Immediately before running your program, the user may have been working with dBASE III PLUS from the dot prompt, in The Assistant, or using another application written in dBASE. So, make sure that your program has a provision to close any database files previously in USE. The CLEAR ALL command in the Cbmenu.prg does this. Include this command at the beginning of your main program.

The Working Environment

When you use dBASE III PLUS from the dot prompt or from The Assistant, a default working environment is provided for you. For example, there is the menu bar at the top of the screen and the status line at the bottom, along with various messages and prompts.

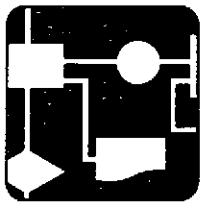
However, when you write a dBASE program, it's up to you to decide how the program is to look. You may wish to mimic the dBASE interactive environment, or design your own, as in the checkbook management system.



SET Command	Dot Prompt Environment	Program Environment
BELL	ON	OFF
COLOR	W/N, N/W	W/B, B/W [for white on blue]
DEFAULT	B [depends on computer]	C [for hard disk]
ESCAPE	ON	OFF
HEADING	ON	OFF
HELP	ON	OFF
MENU	ON	OFF
PATH	B:\ [depends on computer]	C:\DBASE\WORK [for a subdirectory]
SAFETY	ON	OFF
SCOREBOARD	ON	OFF
STATUS	ON [may be OFF]	OFF
TALK	ON	OFF

Table 4-1 Difference between the working environment at the dot prompt and in a program

Setting up the working environment is the next thing you do in the main program module. To change the working environment defaults, use SET commands. There are many SET commands that apply to dBASE programming. SET TALK and SET ESCAPE are the most important.



> **NOTE**
> This section assumes that you are working with the dBASE III PLUS default settings. Another way to establish defaults is to put them in the Config.db file. See *Using dBASE III PLUS* for more information.

SET TALK and SET ESCAPE

At the dot prompt or in The Assistant, dBASE III PLUS answers your commands with messages on the screen. For example, if you initialize a memory variable with the STORE command, dBASE III PLUS shows on the screen the contents of the new memory variable:

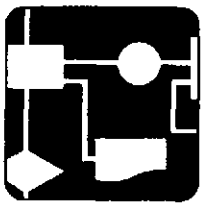
```
. STORE "Enter code number -->" TO prompt
Enter code number -->
```

Because it is responding to your command, dBASE III PLUS calls this TALK. The default is TALK ON. However, you probably don't want dBASE III PLUS's TALK to disrupt your screen appearance. So, issue the SET TALK OFF command. At the beginning of the main program module, most dBASE programs include the command SET TALK OFF.

Pressing **Esc** on the keyboard interrupts and cancels a dBASE III PLUS command. Because you want to control when the program stops, you may not want the user to have access to this key during the run of the program. Include the SET ESCAPE OFF command to stop access to the **Esc** key.

SET ESCAPE OFF gives you strict control over how the user leaves the program. For example, if the program is updating a database file and the user accidentally hits the **Esc** key with SET ESCAPE ON, the database information may be corrupted.

There are many other SET commands for governing the program's working environment. Below is a discussion of those often used in the setup area of the main program file.



Ringing the Bell

dBASE III PLUS rings a bell when input completely fills a field or when the user types an incorrect entry. However, it's a good idea to control the bell yourself. As you'll see in the next chapter, you can sound the computer's bell with a simple command whenever you want to alert users that they have made a mistake or call their attention to something, such as an important screen prompt. This can be controlled with SET BELL ON or OFF.

Color Monitors

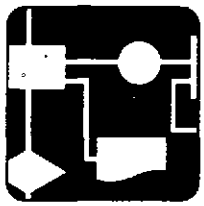
Use the SET COLOR TO command to determine the color attributes of the screen and set high intensity and blinking displays. However, don't go overboard on special effects. They are disrupting or irritating in a busy office. For example, the default color is white letters on a black background, but if you wanted inverse video, you would use the following command:

```
SET COLOR TO N/W
```

This means a black foreground on a white background. The Cbmenu program employs this technique when the user wishes to change the date, the **K** choice from the main menu. Refer to the reference section for SET COLOR in *Using dBASE III PLUS* for more details. The first code SETs the foreground and background colors of the standard display, the second code SETs the foreground and background colors of the enhanced display, and the third is for the border color.

SET COLOR ON/OFF switches between color and monochrome display modes. If the user has a color monitor, you must determine what mode the display is in. One method is to use the ISCOLOR() function to test if the user has a color card. This function returns a logical value, .T. or .F. For instance, the following module switches to monochrome mode if the display is in color mode:

```
IF ISCOLOR()
   SET COLOR OFF
ENDIF
```



The Default Disk Drive

The SET DEFAULT command tells the program where to find files. When you start dBASE III PLUS, it assumes that the logged drive is the default drive. A different default drive may contain database files, a data catalog, or even the other subprograms that run under the main program. To change the default drive, for example to the C drive, use the command:

```
SET DEFAULT TO C
```

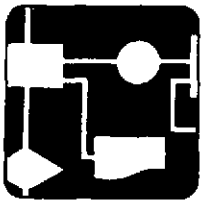
where C is the new default drive. (Also see the Directory Paths section below.)

The Function Keys

You can configure nine of the ten function keys on the keyboard to do whatever you want. (You can't configure the **F1** key, which is reserved by dBASE III PLUS for its help feature.) If you want the function key to enter a command, end with a semicolon to indicate the ↵ key. For example, if you want the user to be able to press the **F2** key to issue the command CLEAR, use the command:

```
SET FUNCTION 2 TO "CLEAR;"
```

Be sure to enclose the command that the function key is to perform in delimiters — either single or double quotation marks or square brackets. There is a limit of 30 characters, including semicolons, that you can program into a function key. If you have a computer with named, instead of numbered, function keys, refer to the discussion of the FKLABEL() and FKMAX() functions in the reference section of *Using dBASE III PLUS*.



TIP
It's good practice to reset the function keys to their default settings at the end of the program and before the program returns to the dot prompt. For example, this line resets the **F3** key to its default setting:

```
SET FUNCTION 3 TO "LIST;"
```

A list of these default settings is in the reference section under SET FUNCTION in *Using dBASE III PLUS*.

Database Field Headings

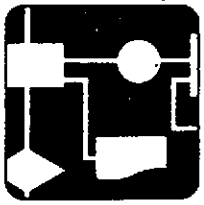
The HEADING is the line that contains the field names when you use certain commands, such as LIST or DISPLAY. Because they usually create their own headings for screen displays, most programmers SET HEADING OFF so that this line doesn't appear.

The Help Message

When the user types an incorrect command at the dot prompt or in ASSIST, dBASE III PLUS responds with the message:

Do you want some help (Y/N)?

If you don't want dBASE III PLUS to present this question, then SET HELP OFF. There are other ways to give the user help in your program. For instance, the checkbook management system has an on-line help file, Help.prg, that describes what each choice in the main menu does.



The Menu Bar and On-Screen Menus

In full-screen applications, dBASE III PLUS presents the user with a menu that shows what certain keys, such as the cursor keys, do. You can see this menu by pressing **F1**. In your program, however, you probably don't want the menu to be on the screen by default, so use SET MENU OFF to turn it off. If your program uses a full-screen command, such as BROWSE, the user can still toggle the menu on with **F1**.

> **NOTE**
> This command does not control the menus that you create. It only governs the dBASE III PLUS on-screen menus.

Directory Paths

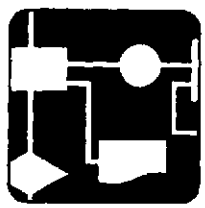
You can use SET PATH to tell dBASE III PLUS to look in other subdirectories for files. If you are unfamiliar with the terms subdirectory, root, and path, check your DOS manual. This command is important for hard disk systems. For example, the command line:

```
SET PATH TO C:\WORK
```

directs the program to look for files in the subdirectory WORK on the C drive, if the program can't find the files in the current directory. The command SET PATH TO, without a path name, releases the PATH.

The Safety Valve

When you attempt to overwrite a file that already exists, such as when you INDEX a database file to an existing index file or COPY a file to an existing file, dBASE III PLUS asks you to verify what you're doing. This is called SAFETY, and the default is ON. If you don't want users to see these messages, SET SAFETY OFF.



The Status Bar and Message Line

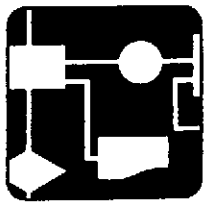
The status bar that appears at the bottom of the screen is a useful aid, but you may not want it on in your program. If so, make sure to include the SET STATUS OFF command in your program's setup area. If your intention is to mimic dBASE III PLUS's normal screen display, you can include the status bar in your programs. You can also use the SET MESSAGE command to put a message below the status line. It works only with the APPEND, CHANGE, EDIT, INSERT, and READ commands.

The Top Row of the Screen

When STATUS is OFF, the top row of the screen is the SCOREBOARD. dBASE III PLUS reserves this line for its own messages. For example, when you DELETE a record, dBASE III PLUS shows the message **Del** on the scoreboard. In your programs, you can use this line to display prompts, or headings, or to ensure that dBASE III PLUS's messages on this line don't appear unexpectedly. So, include the command SET SCOREBOARD OFF in your program. You'll learn more about the scoreboard when you study screen displays in Chapters 6, 7, and 8. The checkbook management program intends to include dBASE III PLUS's message, so SCOREBOARD is ON.

> **NOTE**
> If STATUS is ON, turning the SCOREBOARD OFF has no effect, and dBASE III PLUS messages appear on the status bar. If STATUS is OFF, the scoreboard is on line 0, and if SCOREBOARD is ON, the dBASE III PLUS messages appear on this line. If SCOREBOARD is OFF, the information is suppressed.

There are many more SET commands that govern specific aspects of the dBASE III PLUS working environment. You may have them in the setup area, too, or at the point in the program where they are necessary. Because they relate to topics in other chapters, you will investigate them later.



TIP
Write a standard boilerplate program preamble file that includes the CLEAR ALL command and the SET commands that you use regularly. Then you can copy it whenever you begin a new programming project.

Establishing Memory Variables

After deciding about the work environment, most programmers initialize the memory variables that they will need throughout the program. The checkbook management program RESTOREs variables from a memory file called Chkbook.mem and initializes one other variable:

```
RESTORE FROM Chkbook
today = DATE()
```

Recall that, although these variables are PRIVATE, because the main program initializes them, they work throughout the entire program.

Take a look at the Chkbook.mem file. At the dot prompt, type:

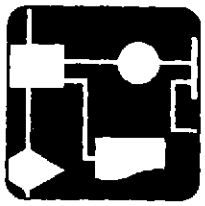
. RESTORE FROM Chkbook ↵

The contents of this file are now in memory. Type

. DISPLAY MEMORY ↵

This is in part what you'll see:

```
BALANCE     public    N     0.00  (          0.00000000)
LASTDEP     public    N     0     (          0.00000000)
LASTCHK     public    N     0     (          0.00000000)
LASTWTH     public    N     0     (          0.00000000)
```



These are numeric type memory variables, and the *balance* variable has two decimal places. In the next chapter, you'll see how to manipulate memory variables and field types for your own purposes, such as displaying them on the screen.

Why does the checkbook management program use a memory file? The variables in the Chkbook.mem file contain the last transactions from the previous run of the checkbook management system. When you first use this program these variables are all 0.00. However, when you start adding deposits, withdrawals, and checks, these variables retain the last transaction amounts.

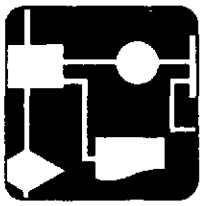
So, when you run the program again, the RESTORE FROM Chkbook command brings in the amounts. Because the program should check if any changes were made to the amounts, it initializes temporary variables that store the beginning amounts. These are the variables *mbalance*, *mlastchk*, *mlastdep*, and *mlastwth*. Later, when the user finishes the program, there are instructions for the program to check whether any new transactions occurred. It compares the contents of the temporary variables with the current contents of the variables RESTOREd from Chkbook.mem. These latter variables may contain new amounts during the run of the program.

The Continuous Loop

Next follows a standard convention that many dBASE programmers use to govern the flow of the entire program. Most dBASE programs have just one main menu, from which all other submenus branch — the one entry/one exit concept. Because the program continually returns to this main menu after each subprogram ends, until the user decides to end the program and return to dBASE III PLUS, the entire flow of the program is in just one DO WHILE...ENDDO loop:

```
DO WHILE .T.
```

The comment line above this command notes that this command forces dBASE III PLUS to do the loop forever, because true is always true. The main menu is in this DO WHILE...ENDDO construction. This is a continuous loop, and for good reason. The program loops continually back to the main menu until the user exits by choosing the **X** choice.



So, with one simple construction, you can ensure that the program shows the main menu whenever control returns to the main program. You don't have to resort to any complicated or tiresome retyping.

The next DO WHILE .T. loop is somewhat fancy. It shows the current time and counts off the seconds on the screen continually until the user selects a choice. When you learn about the INKEY() function in Chapter 8, this will make more sense to you.

The Rest of the Main Program Module

For every choice from **A** to **J**, and choice **L** for help, the main program will call a subprogram. The program handles this entire multiple choice situation, except for the **K** choice, with the DO CASE construction in the main program. The program picks a subprogram, depending on what choice the user types. The program uses the ASCII character function CHR() and the substring search operator $, which instructs the program to look for either an upper case or lower case letter choice. You will investigate these features more thoroughly in Chapters 5 and 8, respectively.

The **K** choice is an exception, so the program has the instructions to handle this choice first:

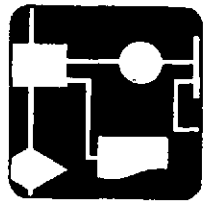
```
IF .NOT. CHR(i) $ "Kk"
   EXIT
ENDIF
```

These lines tell the program to EXIT the current DO WHILE loop if the choice is not **K** or **k**. The program goes right to the DO CASE construction immediately following the ENDDO line. If the choice is **K** or **k**, then the rest of the IF construction does a screen display for changing the date.

Cleaning Up

Your programs should CLEAR ALL database files before the user begins work, and they should do housekeeping when the user finishes, that is, when the user types the **X** choice. Take a look at the CASE construction for this choice:

```
CASE CHR(i) $ "Xx"
```



Look at the IF...ENDIF construction below this choice:

```
IF balance <> mbalance .OR. lastchk <> mlastchk .OR. ;
   lastwth <> mlastwth .OR. lastdep <> mlastdep
```

The IF line instructs the program to check whether the original amounts in the *balance*, *lastchk*, *lastwth*, and *lastdep* variables have been changed. That is, the IF line checks to see if one or more of them are not equal to the amounts in the temporary variables *mbalance*, *mlastchk*, etc. The program SAVEs the contents of the RESTOREd memory variables back to the Chkbook.mem file for use the next time. However, the program only SAVEs if the user has entered new amounts, such as more deposits, checks, or withdrawals.

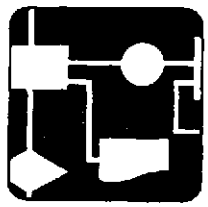
Note also that the program first RELEASEs all temporary variables that begin with the letter *m* with the line:

```
RELEASE ALL LIKE m*
```

It also RELEASEs the other, now unnecessary, variables used in the program. The program does this before using the SAVE command so that it doesn't SAVE the unnecessary variables in the Chkbook.mem file.

Finally, the program resets the working environment to the way it was before the program ran, CLEARs ALL database files, CLEARs the screen, and RETURNs to the dot prompt.

Chapter 5
FUNCTIONS FOR FIELDS AND MEMORY VARIABLES



You communicate with the user by means of screen forms, prompts, and messages. Before you can learn how to set these up, you have to know about the various ways to convert data to a form acceptable for screen displays. The data can be either field information or the contents of memory variables.

What This Chapter Covers

This chapter discusses the following:

- Why type conversions are necessary
- What concatenation means
- How to use functions with field and memory variable types
- How to convert numeric fields and variables to strings, strings to numbers, dates to strings, and strings to dates
- How to deal with the time in a dBASE program

Preparing for This Chapter

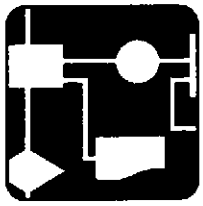
Know the basics of programming in dBASE, field and memory variable types, how to open and USE database files, the DISPLAY, LIST, ? commands, and how to FIND a record. Because it's most effective to see the examples in this section directly, start dBASE III PLUS.

Displaying Information

In dBASE programs, the DISPLAY or LIST commands work exactly as you expect. Remember that the DISPLAY ALL command pauses the scrolling of database information when the screen is full. For example, if the fields you wish to DISPLAY are First, Middle, and Last, you might use the command line:

```
* Show the three fields with the record number off
DISPLAY OFF ALL First, Middle, Last
```

However, this may disrupt any other screen messages or prompts that you've set up. Because it can interrupt the nice look of your screen displays, the LIST command is more dangerous from a programming standpoint. Remember that LIST without any parameters shows all the database file information on the screen, but does not pause the display unless you type **Ctrl-S**.



When you use your own customized screens, you will want to position the field information differently on the screen than the way dBASE III PLUS positions it. DISPLAY and LIST don't give you much flexibility. Neither does the query command, ?.

In the next chapter, you'll learn a better way to display information with the @...SAY command. @...SAY allows you to place information, such as field information, the contents of memory variables, or strings, anywhere on the screen.

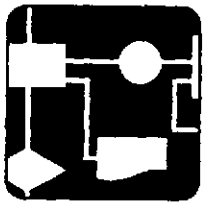
This, too, has its limitations. If you want to show numeric, date, and character data on the very same line of the screen, you can't mix and match different data types in the same @...SAY command, unless this data is in the form of character strings. So, you must convert non-character data to character strings.

Type Conversion Explained

A character type field or memory variable can contain any printable character from the keyboard, including numbers and punctuation marks. However, information requiring a calculated result, such as a dollar amount, works better in a numeric field. In addition, the dBASE III PLUS date fields provide ways to check for the day of the week or month.

Numeric, date, and logical fields contain printable characters, but, as far as dBASE III PLUS is concerned, a numeric, date, or logical field is not in character format. dBASE III PLUS won't allow you to mix numeric, date, and character fields on the same line of the screen, or join together a numeric field with a string field. You must first convert numeric and date information into a character string.

> **NOTE**
> You convert fields or memory variables to character strings merely for display purposes. You are not changing field types in the database file, nor are you modifying the structure of the database file.



The many dBASE III PLUS conversion functions, discussed below, help you do string conversions. Here are some general rules for using these functions:

- If you use the name of a field or variable in the function, dBASE III PLUS will automatically ascertain its type and length.
- When you want to use strings that aren't in fields or memory variables, enclose them in delimiters, usually single or double quotes.
- When converting a numeric field into a character string, be aware of the length and number of decimal places of numeric fields.
- Date fields are special fields with their own conversion functions.
- For logical fields, all you need to do is supply the character representatives of .T. or .F., such as **True** or **Yes** for .T.
- Memo fields are a special case.

Concatenation

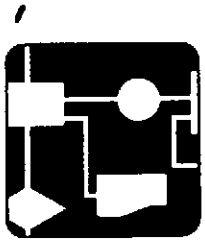
Although it's logically impossible to add or subtract one character string from another, you can assemble strings together to form a new string. This is known as concatenation, and dBASE III PLUS uses the plus sign, +, for it. When you concatenate strings, ensure that the correct spaces are between them.

For example, you have a character field called Part_no which contains a six-character part number, and the current record contains ABC123 in the field. The program line to concatenate the field information with the string **The part number is** would be:

```
? "The part number is" + Part_no
```

which would result in: **The part number isABC123**. The two strings run together, because you forgot the space between them. You could do this:

```
? "The part number is " + Part_no
```



Notice the extra space after **is**. You could also do it this way:

```
? "The part number is" + " " + Part_no
```

Notice the extra space in the quotation marks. There is another way to concatenate strings, which you'll learn about shortly.

Comparing Strings

Even though strings aren't numbers, you can still compare them. That's because the ASCII code values for each printable character are in a certain numerical order. Refer to the ASCII code table in Appendix E of *Using dBASE III PLUS* for the code values. For example, if you typed ? 'A' < 'a' you would get a **.T.** response, because the ASCII code for **A** (65) is less than the ASCII code for **a** (97).

Similarly, you can compare a character string with the line **? '950' > '750'**. Note that these are strings because they are delimited, even though they contain numbers. Because the ASCII code for **9** (57) is higher than the ASCII code for **7** (55), you get a **.T.** response.

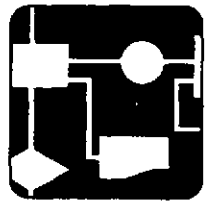
WARNING
Don't mix types in comparisons. For instance, you can't compare a string to a number. Make sure that you're comparing the same types. Otherwise, you'll get the error message **Data type mismatch**. Moreover, in string comparisons, dBASE III PLUS compares all characters until it runs out on the right-hand side of the relational operator, =. For instance, if it were doing this comparison:

```
? 'abcd' = 'abc'
```

it would give a **.T.** response. But this comparison:

```
? 'abc' = 'abcd'
```

gives a **.F.** response.



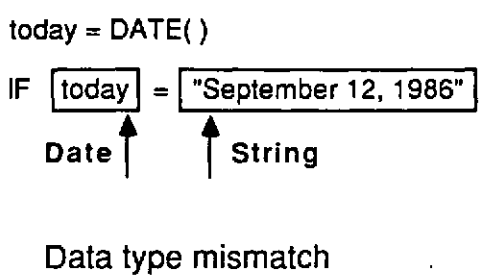


Figure 5-1 You can't compare fields or variables of different types

Numeric Functions

There are several functions that work with numeric data to produce a numeric value. They are relatively straightforward. You can find thorough explanations of these functions in *Using dBASE III PLUS*. Here they are:

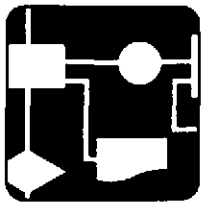
ABS() — absolute value
EXP() — exponential value
INT() — integer value
LOG() — natural logarithm
MAX() — maximum value
MIN() — minimum value
MOD() — modulus (remainder)
ROUND() — round a number
SQRT() — square root

These functions always return a numeric value. The INT() function is important to the discussion of conversions here. However, it will make more sense to you after you've first looked at the string functions.

String Functions

dBASE III PLUS has many functions that deal with character strings. You'll use the following string to illustrate these functions:

Your choice is incorrect — change it? (y/n)



So that you don't have to type in the same string continually, first initialize it as a memory variable called *string*:

```
STORE 'Your choice is incorrect -- change it? (y/n)' TO string
```

You can determine a great many conditions using string functions. One is to check whether or not the user has typed a correct response.

The Length of a String

Often you need to know how long a string is. For example, if the user has typed in a command, you can check its length to make sure that the input is correct. The function for this is LEN(). This function always returns a numeric value. The entire string, the character field name, or the memory variable name must be in parentheses. Here are some examples:

```
? LEN(string)
```

returns the length of the initialized memory variable *string*, which is 44. The delimiters are not included in the length.

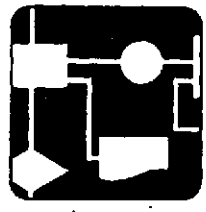
```
? LEN("What's my line?")
```

returns 15, which is the length of the string **What's my line?**. If there were a database file in USE containing a field named First, the command

```
? LEN(First)
```

would return the length of this field. You can set up a simple IF construction to test for string length:

```
* If the length of the input variable is not 5
IF LEN(input) <> 5
   DO Error                  && Branch to Error.prg
* The length is 5, so go on
ELSE
   * (commands)
ENDIF
```



Getting Part of a String

You can ask dBASE III PLUS to give you only a part of the string. This is known as a substring, and the function for it is SUBSTR(). You must tell dBASE III PLUS where to start in the string and how many characters to count starting from that position. The first number gives the starting position; the second number gives the length from the starting position. For example:

```
? SUBSTR(string,1,24)
```

tells dBASE III PLUS to display the memory variable *string* starting at position 1 and continuing for 24 characters. The result is: **Your choice is incorrect**.

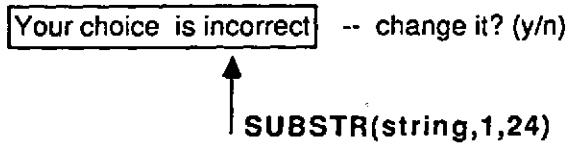


Figure 5-2 A substring is part of a string

If you provide a starting number but no ending number, then dBASE III PLUS assumes that you want the rest of the string, starting at that position:

```
? SUBSTR(string,29)
```

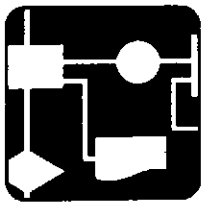
results in the answer: **change it? (y/n)**.

With the SUBSTR() function, you can reuse part of a string in another string, without having to retype the whole string. For example:

```
? "I don't like your answer, do you want to " + ;
SUBSTR(string,29)
```

gives the new string:

I don't like your answer, do you want to change it? (y/n).



Note that there is a space after the first **to**, so the two strings are correctly spaced.

Left and Right Sides of Strings

You can get certain characters in the string starting from the left or the right side with the LEFT() and RIGHT() functions, respectively. LEFT() is like the SUBSTR() function, but you don't have to supply a starting address, because it is automatically positioned at the left side of the string. Conversely, RIGHT() starts with the last position in the string and works backward (that is, right-to-left). To illustrate:

```
? LEFT(string,24)
```

would result in:

Your choice is incorrect

whereas

```
? RIGHT(string,16)
```

would result in:

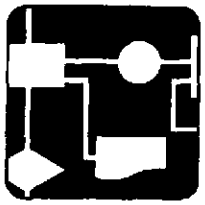
change it? (y/n)

Substring Position

The AT() function returns the starting position number of the substring in a string. For example,

```
? AT('change', string)
```

gives the number 29, the position where the string **change** starts. If the substring is not in the string, then AT() returns the value **0**. Note that the substring **change** is enclosed in delimiters.



Changing Between Upper Case and Lower Case

Sometimes you may want to convert a string to upper case or lower case. The UPPER() and LOWER() functions will convert the entire string:

```
? UPPER(string)
```

would result in

YOUR CHOICE IS INCORRECT — CHANGE IT? (Y/N)

whereas

```
? LOWER(string)
```

would result in

your choice is incorrect — change it? (y/n)

How would you convert only part of a string? Use the UPPER() or LOWER() function with the SUBSTR() function. For example:

```
? 'T' + LOWER(SUBSTR('THIS IS NOT COMPLETELY IN LOWER CASE',2))
```

gives the answer

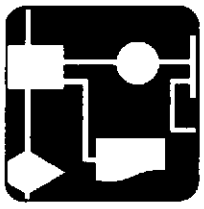
This is not completely in lower case

because the SUBSTR() function starts the string at position two and includes the rest of the string.

The Add.prg module uses the UPPER() function to test for a user response:

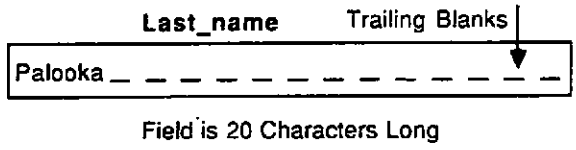
```
IF UPPER(answer)="Y"
   EXIT
ENDIF
```

You can use the string functions in many different combinations, but be careful to include the correct number of parentheses and to use the concatenation operator, +, between strings.



Trimming an Entry

It is important to avoid trailing blanks. When you enter information in a character field and leave a few empty spaces in the field, dBASE III PLUS fills them up with trailing blanks. If you don't want them to appear when you display the field, use the TRIM() function to take them out.




```
.? Trim (Last_name)
```

Palooka

Figure 5-3 Trailing blanks

For example, the First_name field in a database file is 15 characters long, and you want to display the field information for the current record together with the Last_name field. The two fields contain the entries Joe and Palooka, respectively. The line

```
? First_name + Last_name
```

would result in

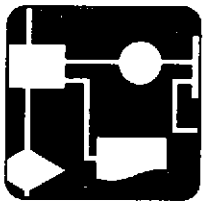
Joe Palooka

The entry

```
? TRIM(First_name) + Last_name
```

would result in

JoePalooka



That's not what you want either. The solution is:

```
? TRIM(First_name) + ' ' + Last_name
```

OR

```
? TRIM(First_name), Last_name
```

Either way gives you:

Joe Palooka

If First_name contained Stephanie and Last_name contained Stevenson, the answer to the same query would be:

Stephanie Stevenson

TRIM() trims only the trailing blanks. There are two other trimming functions: RTRIM() and LTRIM(). RTRIM() is exactly the same as TRIM(). It trims trailing blanks. LTRIM() trims the leading blanks at the beginning of a string. This is very useful for catching user errors. If the user inadvertently types a space to begin a character entry, you can use LTRIM() in your program to correct the entry.

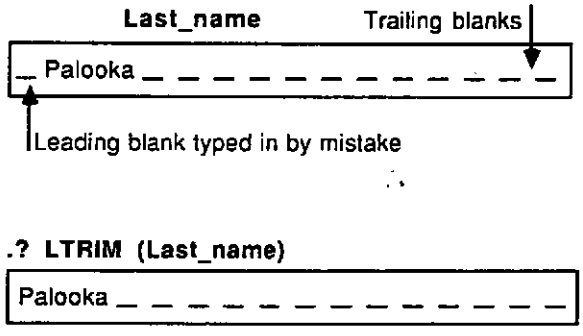
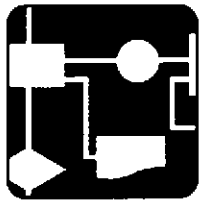


Figure 5-4 Leading blanks



TIP
Using the TRIM(), LTRIM(), and RTRIM() functions will help you avoid user errors and increase data precision. When you want the program to STORE the contents of a character field in a memory variable, use LTRIM() with TRIM() to avoid any unnecessary leading or trailing blanks in the field. Similarly, if the user is to type in a name, first make sure that you account for leading and trailing blanks in the name before doing anything else with it.

For example, you initialize a memory variable called *mlast* to hold the user's input of a last name:

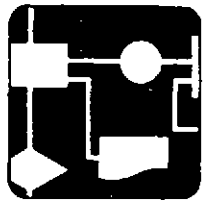
```
STORE SPACE(20) TO mlast
```

The last name that the user types probably will contain under 20 characters, but the user might type a space before the name accidentally. To ensure that dBASE III PLUS only deals with the real last name, and not the leading or trailing blanks, have the program do this:

```
STORE LTRIM(TRIM(mlast)) TO mlast
```

Getting rid of trailing blanks is especially important when you need an exact match between the contents of a memory variable and field information.

Another common use of the trimming function is to find the actual length of a field's contents. For example, if a database file contains a field called Name, LEN(Name) always returns the field length rather than the actual length of Name. LEN(TRIM(Name)) solves this problem.



Another Way to Concatenate

You can also concatenate strings with the – operator, which does two operations at once. It joins two strings, just like the + operator, but it also moves the trailing blanks from the first string to the end of the resulting string. Thus:

```
? First_name - Last_name
```

results in:

StephanieStevenson

This concatenation operator is most useful when you want to join two character fields, such as a department number field and an employee number field to result in a new, composite number. Here, you're creating a new employee number by concatenating the character field Dep_no, with a length of four, with the character field Emp_no, which also has a length of four:

```
STORE Dep_no - Emp_no TO newnum
```

If the Dep_no field contains "A " with three trailing blanks, and the Emp_no field contains 123X, the *newnum* variable contains the string "A123X ". Note that the three trailing blanks have been moved to the end of the new string.

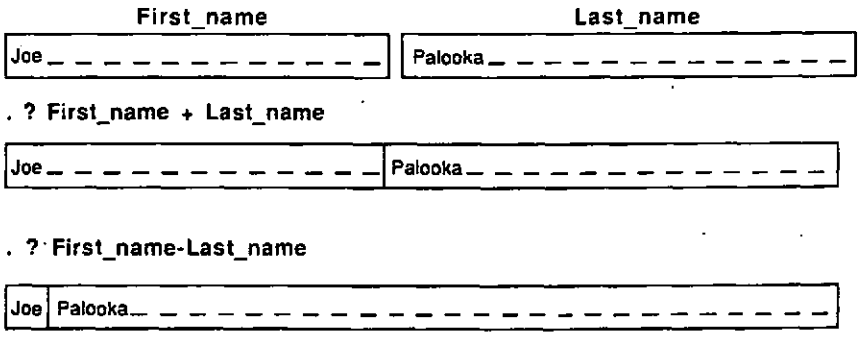
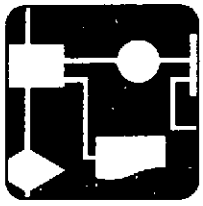


Figure 5-5 Two types of concatenation



Strings as ASCII Characters

Computers convert all characters to special numeric codes. Every printable, keyboard, and screen character has a unique ASCII code. These characters and their ASCII codes are in Appendix A of *Using dBASE III PLUS*. You can display a character with the CHR() function if you know its ASCII code number.

Frequently, you will use this function to create special screen displays and to ring the bell, which also has an ASCII code number. For example,

```
? CHR(201)
```

is the same code which produces the upper left corner of the box in the checkbook management system main menu.

There's also the ASC() function, which returns the ASCII code for a character. Thus:

```
? ASC('a')
```

gives the integer 97, the ASCII code for **a**. You can use this function to increment letters. For example,

```
? CHR(ASC('a') + 1)
```

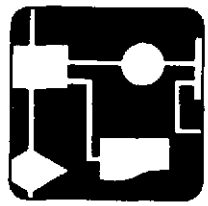
gives the answer **b**.

Controlling the Bell

ASCII code 7 is what makes the bell sound:

```
? CHR(7)
```

Use the bell sparingly for situations where you want to grab the user's attention. Remember that dBASE III PLUS also rings the bell when the user has typed in the contents of a field completely, unless you SET BELL OFF. CHR(7) will ring the bell regardless of whether BELL is ON or OFF.



NOTE
There are a few ASCII codes in the range 1-31 that have two meanings, depending on their context. When you learn about the @...SAY command in the next chapter, you'll see that the ? command and the @...SAY command can produce different characters for the same ASCII code number in the range 1-31.

Numeric to Character Conversion

There are several functions that deal specifically with converting numeric data into character strings, a common conversion in programming.

The STR() Function

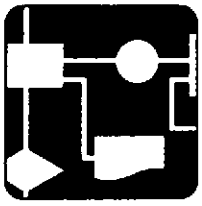
STR() is the dBASE III PLUS function that returns the string equivalent of a number. For instance,

```
? STR(3000)
```

returns the character string " 3000" but without the quotation marks. Notice the blanks at the beginning of the string. The STR() function normally returns a string that is ten characters long. dBASE III PLUS pads the number with blanks. However, it is advisable to give the STR() function two arguments to determine the length of the new string and the number of decimal places. For example,

```
? STR(3000,4)
```

gives the string answer **3000**. The 4 refers to the four total display places of the string. You eliminate any leading blanks by determining the total number of places yourself.



In the Reconcil.prg module of the checkbook management system, the program uses the STR() function together with another string in an @...SAY line. This line displays a memory variable called *diff* converted to a string on row 19, column 23:

```
@ 19,23 SAY "The difference is $"+STR(diff,10,2)
```

Notice that the dollar sign is not part of the STR() function, but by concatenating the two strings an amount is formed on the screen. For example:

The difference is $ 23.05

To display a period at the end of this line, concatenate it to the end:

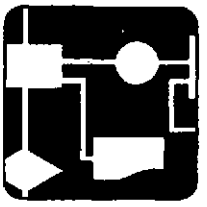
```
@ 19,23 SAY "The difference is $"+STR(diff,10,2)+"."
```

Converting Numbers with Decimal Places

The string in the above example has two decimal places. Using an optional third argument in the STR() function signifies the number of decimal places in the new string. However, the total number of places, that is the second argument, must include the number of decimal places plus one space for the decimal point:

```
? STR(35.50,5,2)
```

gives you the string **35.50**. The second argument, 5, refers to the total number of places, including one place for the decimal point. The third argument, 2, signifies the total number of decimal places. In addition, if the number is negative, the total number must include an extra place for the minus sign.



If you give dBASE III PLUS arguments that don't make sense, for instance if you don't include the decimal places and decimal point in the total number of places of the string, dBASE III PLUS will give you an error message. For example:

```
STORE 1 TO num
? STR(num,1,2)
```

results in the error message:

*****Execution error on STR() : Out of range**

because you can't have two decimal places but only a total of one place for the string. Similarly, if you try this:

```
STORE .50 TO num
? STR(num,2,2)
```

you get the same error message, because you need at least four total places for the display: two for the decimal places, one for the decimal point, and one for a leading 0. In the above example, dBASE does not return .50; it must return 0.50. Be careful when converting numbers with decimal points that you include the total number of places in the new string.

Similarly, if you define a STR() function with fewer places than the number, you get an overflow message, shown as a string of asterisks:

```
num = 55555
? STR(num,4)
****
```

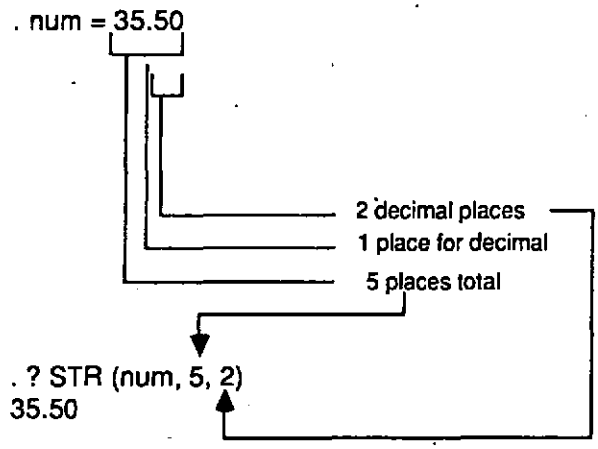
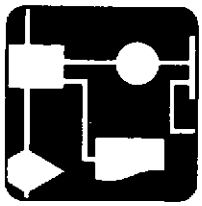


Figure 5-6 Take care with decimal places

TIP
You can control the minimum number of decimal places that certain numeric functions show with the SET DECIMALS command. This command works only with the EXP(), LOG(), SQRT(), and VAL() functions, and with division. The command SET DECIMALS TO instructs dBASE III PLUS to show the answers to these operations with two decimal places. However, if an answer returns a number with four decimal places, use the SET FIXED ON command, which establishes a fixed number of decimal places for display output.



If a number has no decimal places, convert it to a string to display it with decimal places:

```
string = 35
? STR(string,5,2)
```

gives the string **35.00**.

Strings to Numbers

You can convert a string to a number with the VAL() function:

```
. STORE "599.85" TO string
? VAL(string)
```

returns the number **599.85**. To convert the string to an integer, you use the VAL() function with the INT() function:

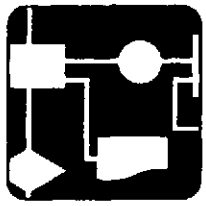
```
STORE "599.85" TO string
STORE INT(VAL(string)) TO newstring
? newstring
```

returns the integer number **599**.

NOTE
Be careful when using the VAL() function with strings that don't contain numbers. dBASE III PLUS returns the value of 0 in this case. For example,

```
? VAL("Hello there!")
```

results in 0.



Date Arithmetic

Although it displays dates in a format that you can readily understand, dBASE III PLUS sees date fields and date type memory variables as special numbers. dBASE III PLUS has special functions that manipulate date information. The date functions allow you to work with dates in general without knowing what the actual date is. For example, the DATE() function returns today's date, which you entered when you started DOS. It's in the form MM/DD/YY, although you can change that format.

You can add a number to a date to get a new date. For example, if you are writing an accounting program that relies on dates, you can have a command like this:

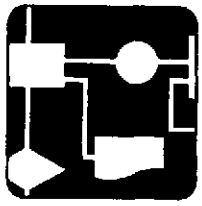
```
STORE DATE() + 30 TO overdue
```

The program adds thirty days to the current date to get a date variable called *overdue* which contains the new date. Then use this memory variable to test for a condition:

```
* If the account is overdue today
IF overdue = DATE()
   Do Dunltr          && Send out a dun letter
ELSE
   * Or if the account is past due
   IF overdue > DATE()
      DO Nasty        && Send out a stronger letter
   ENDIF
ENDIF
```

Similarly, you can also subtract a number from a date to get a new date, or you can subtract two dates to get the numeric result of the number of days between the dates:

```
STORE DATE() TO today
STORE today - 1 TO yesterday
STORE yesterday - 1 TO daybefore
STORE today - daybefore TO diff
```



You can get the numeric equivalents to the days of the week or month, the months of the year, or the year itself with other date functions. For example, if the current DATE() is September 12, 1986, note what you get for each function below:

```
? DOW(DATE())
```

6, for the sixth day of the week

```
? DAY(DATE())
```

12, for the twelfth day of the month

```
? MONTH(DATE())
```

9, for the ninth month of the year

```
? YEAR(DATE())
```

1986, the year

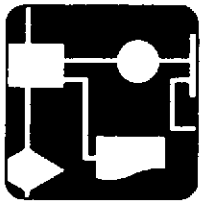
dBASE III PLUS considers Sunday as the first day of the week. You can get the strings or substrings of these numbers if you wish:

```
? SUBSTR(STR(YEAR(DATE()),4),3,2)
```

gives the string **86**, for the example used, September 12, 1986. You can also STORE the value of any of these functions in a memory variable for later use in your programs. If you do this, remember to declare the variable PUBLIC first.

Date Formats

The standard format for the date, MM/DD/YY, is the default American format. There are others you can use; for example, the British format is DD/MM/YY, and the German format is DD.MM.YY. You use the SET DATE command to change the format of date fields or variables. So, SET DATE BRITISH changes the format to DD/MM/YY. See *Using dBASE III PLUS* under SET DATE for the other date formats.



There is also a SET CENTURY command, which switches the display for the year between two digits and four. SET CENTURY is normally OFF, so use SET CENTURY ON to display a date in the form MM/DD/YYYY, for the American format. Thus September 12, 1986 would appear as 09/12/1986.

> **NOTE**
> Use SET CENTURY ON if you want to input a century other than 1900. Even with SET CENTURY ON, a date will still take up only eight characters in a file structure and nine in a memory variable.

Date-to-Character Conversions

When you wish to use dates with character strings, you first have to convert the dates to strings. You do this with the date-to- character function, DTOC(). For example:

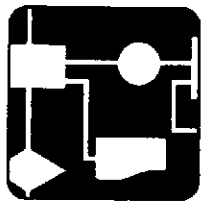
```
? DTOC(DATE())
```

gives a string of the current date. If the date were September 12, 1986, with SET CENTURY OFF, then the answer to the above inquiry is the character string **09/12/86**. If you have SET the date to the BRITISH format, the answer to the same query is **12/09/86**. With SET CENTURY ON and SET DATE BRITISH, the result is **12/09/1986.**

You can, of course, use any of the other string functions once you've converted a date to a string. For example:

```
SET DATE BRITISH
? SUBSTR(DTOC(DATE()),1,2)
```

gives the string **12**.



WARNING
You can't perform arithmetic operations, such as adding numbers to dates, on strings which you create from dates. First, you have to convert the strings back to dates. You'll see how in a moment.

There are other useful date-to-character functions that reduce needless programming efforts on your part. Note the string results for the following functions when DATE() is 09/12/86:

```
? CDOW(DATE())
```

Friday

```
? CMONTH(DATE())
```

September

You can provide your users with string equivalents to a date to make your printed reports look a lot better. For example:

```
? CDOW(DATE()) +", " + CMONTH(DATE()) +" " + ;
LTRIM(STR(DAY(DATE()),2)) +", " + STR(YEAR(DATE()),4)
```

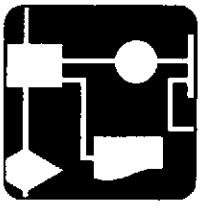
gives the string:

Friday, September 12, 1986

Once you've set up this command line, it works with the current DATE(), no matter what it is.

Character-to-Date Conversions

You can convert a string in the correct date format to a date with the CTOD(), character-to-date, function. The correct format is MM/DD/YY. For example, assuming that the date format is AMERICAN, and SET CENTURY is OFF:




```
STORE '09/12/86' TO string
STORE CTOD(string) TO newday
```

results in the date variable **09/12/86**. If you use a string directly in a CTOD() function, you must enclose the string in delimiters:

```
? CTOD('09/12/86')
```

09/12/86

If you attempt to convert a string to a date, but the string is not in the correct date format, dBASE gives you a blank date:

```
? CTOD('September 12th, 1986')
  / /
```

However, if you convert a string that is in the correct date format, but the corresponding date is incorrect, dBASE III PLUS corrects the date. For example:

```
? CTOD('02/29/85')
```

gives the answer **03/01/85**, because dBASE III PLUS knows that February in 1985 only had 28 days. So it gives the next available real date.

Using Dates in Comparisons

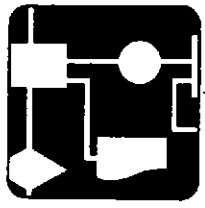
Because chronological comparisons are different than string comparisons, a comparison of two strings could give you an incorrect date. You must convert characters to dates to get correct results. For example,

```
? '01/01/86' > '12/31/85'
```

gives the answer **.F.**, although January 1, 1986 is the day after December 31, 1985. The correct syntax is:

```
? CTOD('01/01/86') > CTOD('12/31/85')
```

This yields a **.T.** response.



So, it's best to convert a string to a date before doing chronological comparisons. For example, if you want to DISPLAY all database records with a date field, Chg_date, containing the date 09/12/86, issue the command:

```
DISPLAY OFF ALL FOR Chg_date = CTOD('09/12/86')
```

You'll use this technique frequently in reports, for instance, to validate that what the user types is an actual date.

How to Initialize a Date Variable

The only way to initialize a date variable is to use an existing date field or a function that returns a date, that is, either DATE() or CTOD(). To initialize a date variable, *today*, with the current date, you would enter:

```
today = DATE()
```

To initialize a date other than the current date in a memory variable, use the CTOD() function:

```
lastweek = CTOD('08/07/85')
```

You can also initialize a blank date variable:

```
newdate = CTOD('  /  /  ')
```

Using Time

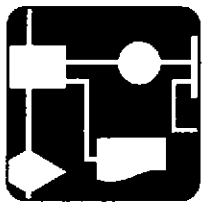
dBASE III PLUS's TIME() function returns the current time in the form hours/minutes/seconds, provided that you entered the time at the DOS prompt when you turned on your computer.

```
? TIME()
```

gives the string answer **14:30:43** if it's 2:30 p.m. and 43 seconds.

TIME() always returns a string. To save the current time in a database field, define the field as character type with a length of 8, and then REPLACE its contents with a string made from the current time. For instance, if it's now 2:30 p.m. and the field name is Now:

```
REPLACE Now WITH TIME()
```



The field Now contains **14:30:43**. The seconds, of course, will be different. You could then split Now into substrings if you wanted to include only part of the time:

```
REPLACE Now WITH SUBSTR(Now,1,5)
```

puts the string **14:30** into Now.

Memo Fields

You can't use the string functions on memo field information, because memo fields aren't strings. A memo field is like a signpost, pointing to a separate file which contains the free-form contents for the memo in each record.

The only way you can change a memo field from within a program is to set up a special format file. You'll see how to edit or add to memo fields in Chapter 8. There is no such thing as a memo-type memory variable.

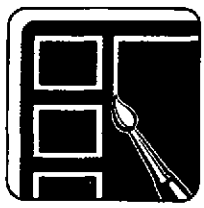
Logical Fields

Use logical-type data to determine conditions in DO WHILE, DO CASE, and IF constructions. However, if you want to display the contents of a logical field or variable on the screen, you can use a string equivalent. This isolates the user from the dBASE III PLUS logical operators, .T. and .F., which might be confusing.

Because a logical field can either be true or false, it's easy to display one of two responses. Here, *mcorrect* is a logical type memory variable:

```
IF mcorrect   && That is, if mcorrect is .T.
   ? "That's correct!"
ELSE          && It's .F.
   ? "Sorry, that's not correct -- try again!"
ENDIF
```

Chapter 6
COMMUNICATING WITH THE USER



Because you can't assist your users personally, your program must communicate clearly, helpfully, and efficiently so they can work with the program easily. The most important part of any program is the user interface — the communication with the user.

What This Chapter Covers

The topics covered in this chapter are:

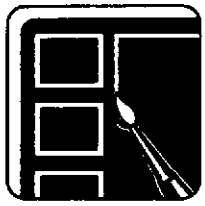
- What screen coordinates are
- How to position prompts and messages on the screen
- The commands for receiving and controlling responses (input) from the user
- How to use the SET CONFIRM command in @...GET lines
- The basic pattern for using screen forms in programs

Preparing for This Chapter

By now you should have a general knowledge of dBASE programming, including how to convert numeric and date type information to character strings, discussed in Chapter 5. It's preferable to study this chapter in conjunction with the next two, which continue the discussion of designing screen forms.

Using Screen Forms — The Recommended Way

The primary function of the user interface is to make the program easy to use. The phrase *user-friendly* describes exactly how your program's user interface should be. Whenever possible, design customized screen forms that mimic the user's own forms. These screen forms also provide assistance and guidance so that the user is never lost or unsure about what to do next. Supply easy-to-understand messages and prompts to guide users through the program. At the end of the next chapter are some pointers for designing screen forms.

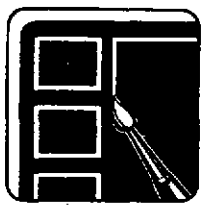


NOTE
The discussion in this chapter relates to entire screen forms and to shorter, one-line prompts and messages. The setup techniques are the same for both. These methods are also used for obtaining and displaying information.

Although you can show actual field contents on the screen, you might not want the user to be able to enter information directly from the keyboard to a database file. Remember that the full-screen commands, APPEND, BROWSE, CHANGE, and EDIT, allow the program to do this.

For the sake of database integrity, you may want to take firm control over how the user changes database information. For instance, if your users are inexperienced, you wouldn't want to give them access to the full-screen commands. Using on-screen forms is the recommended way to have users enter or change data in a database file. The discussion in the present chapter and the following ones stresses this approach.

Using custom-designed screen forms, the program gets input from the user and STOREs this input temporarily in memory variables. When the user verifies that the input is correct, the program REPLACEs the current information in a database field with the new information. Using memory variables throughout your screen form design ensures maximum protection of the user's database file.



NOTE
There is one exception to this practice: memo fields, which can only be changed directly. See Chapter 8 for a discussion of this exception.

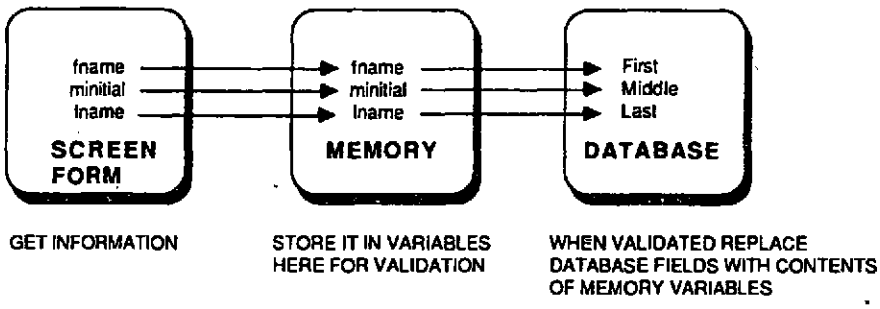
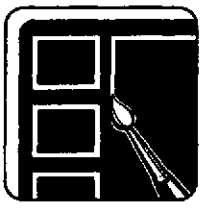


Figure 6-1 How a screen form works

You already know how to initialize memory variables. Chapters 9 and 11 deal with verifying user input and updating the database file. This chapter and the next concentrate on screen forms.

Controlling the Screen

Before you investigate the ways to implement the above pattern, you have to be able to position prompts, messages, and places for the user input exactly where you want them on the screen. This means learning how the computer works with the screen area.



Screen Coordinates

The screen display is a grid of evenly spaced horizontal and vertical lines. dBASE III PLUS refers to the horizontal positions on the screen as rows and the vertical positions as columns. The intersection of a given row and column is called a screen coordinate. Every coordinate can display information. You can put any printable character on the screen by telling dBASE III PLUS the location of the coordinate with the @...SAY command.

Microcomputer screens generally have 25 horizontal rows and 80 vertical columns. These rows and columns are numbered from the top of the screen down to the bottom, and from left to right. A screen coordinate consists of two numbers: the horizontal row number is listed first, and the vertical column second, such as: 1,5.

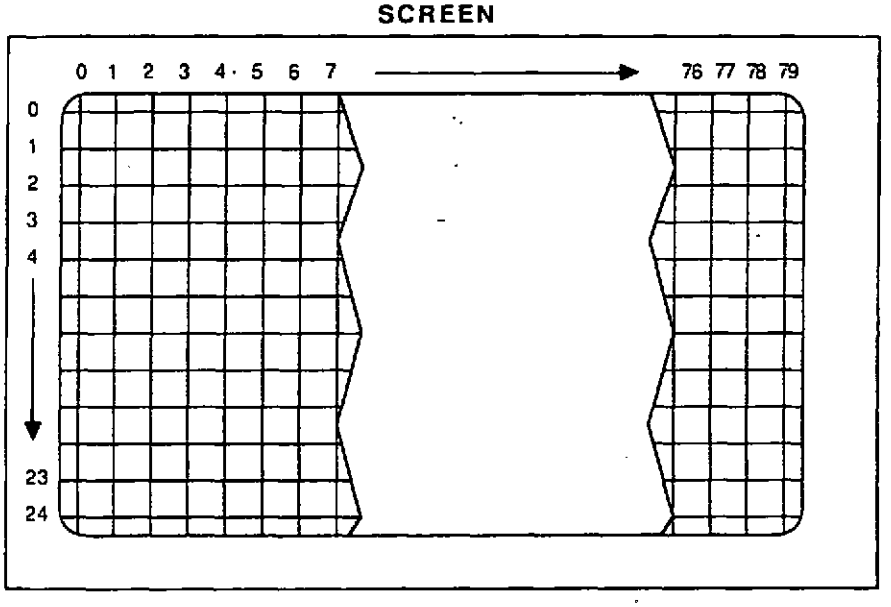


Figure 6-2 Screen coordinates

Every character position on the screen has row and column coordinates.

Coordinates start with 0, rather than 1. The top row of the screen is row 0, and the bottom is row 24. Similarly, the leftmost column number is column 0, while the rightmost column is column 79. So the coordinates 1,5 refer to the second row on the screen and the sixth column.

> **TIP**
> Remember that with SET STATUS OFF, the top row of the screen, row 0, is the scoreboard. If you decide to display information on this row and don't want certain dBASE III PLUS messages to disrupt your screen, make sure that you have SET SCOREBOARD OFF.

The @...SAY command requires that you supply correct screen coordinates after the @ command and what you want to display following SAY. For example, if you want to display the current First_name field of a database file in USE on row 5, column 10 of the screen, here is the way to do it:

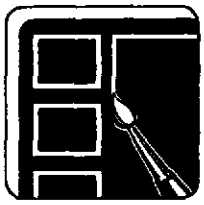
```
@ 5,10 SAY First_name
```

If you want to display a memory variable on the screen, make sure that the variable is initialized before using @...SAY.

```
PUBLIC message1
STORE "That's not a proper response" TO message1
@ 10,7 SAY message1
```

If the program is to display a string, enclose the string in correct delimiters:

```
@ 15,15 SAY "Do you want to correct your mistake? (y/n)"
```



dBASE III PLUS paints the screen in the order that the @...SAY commands appear in the program file. It's best to have the program show information on the screen from the top down and from left to right. However, for some special effects you can arrange the order of the @...SAY command lines in your programs to display information in any way you want.

Screen and Printer Coordinates

In Chapter 12, you'll learn how to print from within a program using @...SAY lines to position output on the printed page. In this case, you can have coordinates beyond those allowed for screen displays. However, the highest coordinate you can use for either horizontal or vertical positioning in printouts is 255, and in screen displays, dBASE III PLUS gives you an error message if you use a coordinate, such as (24,205), that is off the screen.

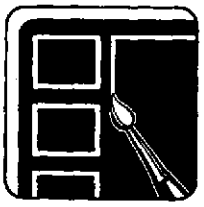
Ways of Clearing the Screen

The screen display doesn't change unless you change it with instructions in your program. Make sure that your program clears away the previous screen display before it shows the user another screen, or whenever you want the display to disappear. The CLEAR command blanks out the entire screen. If you want to clear only a part of the screen, there are ways to do so.

To clear the screen from a specific row to the bottom of the screen, use @...CLEAR with the beginning row's coordinates. For instance, if you want to clear the screen from row 19, but leave rows 0-18 intact, use:

```
@ 19,0 CLEAR
```

WARNING
Don't confuse the CLEAR command with the CLEAR ALL and CLEAR MEMORY commands. Issue CLEAR by itself to clear the screen.



If you want to clear only one row at a time, just supply the coordinates of that row after @. The following command lines clear the screen from row 19 to the bottom of the screen, one row at a time:

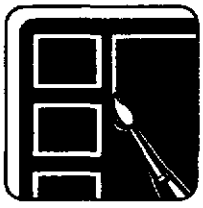
```
@ 19,0
@ 20,0
@ 21,0
@ 22,0
@ 23,0
@ 24,0
```

You can also clear a rectangular area of the screen by providing its upper left and lower right coordinates:

```
@ 5,1 CLEAR TO 10,75
```

The checkbook management system uses another technique to clear part of a row. It fills that part with spaces, deleting the display in that section:

```
@ 6,15 SAY SPACE(20)
```



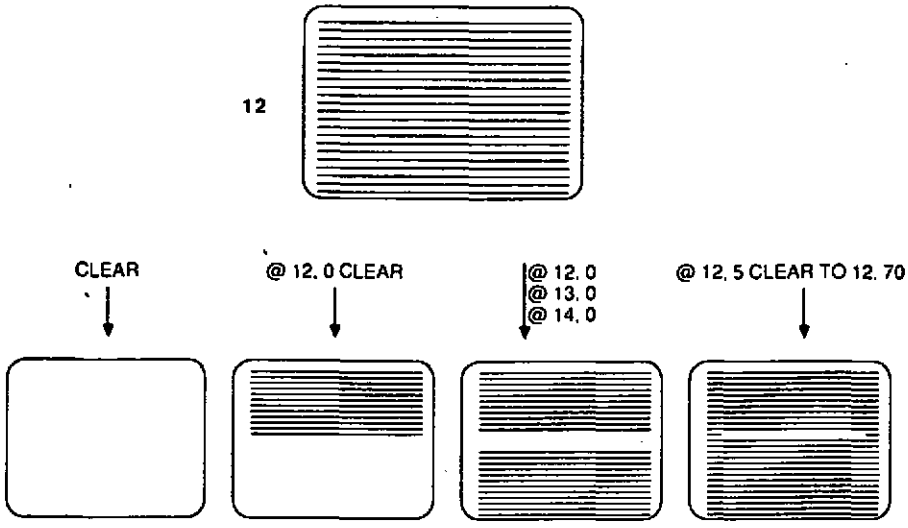


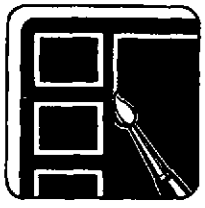
Figure 6-3 Ways to CLEAR a screen

You can CLEAR the entire screen from a specific row to the bottom of the screen, certain rows only, or only parts of a row.

The TEXT ...ENDTEXT Construction

If you want to display large amounts of text on the screen — for example, in a help screen — you don't have to set up coordinates or use strings enclosed in delimiters for each and every row. Instead, use the TEXT...ENDTEXT construction. TEXT...ENDTEXT instructs dBASE III PLUS to display the contents between the commands exactly as shown.

Keep in mind when you prepare your text that the screen is 80 columns wide. Make sure your display is centered correctly on the screen. Take a look at the Help.prg file, which contains the help screens for the checkbook management system. The very beginning of the file looks like this:



```
CLEAR
TEXT

Welcome to the Checkbook Management System. This system is
designed to keep track of your deposits, your withdrawals,
your checks, balancing your checkbook, and printing reports
of several combinations. Below are listed the menu selections
and a description.
(...)
ENDTEXT
```

Notice how the program sets up each screen with TEXT...ENDTEXT and the blank lines for spacing between paragraphs. The TEXT command never uses the SCOREBOARD, even if you SET SCOREBOARD OFF. You can't display field or memory variable information within a TEXT...ENDTEXT construction.

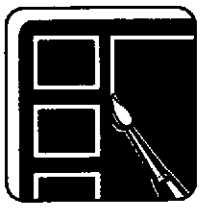
How to Get User Input

There are several commands for receiving input from the user. Which one you choose depends on the kind of input that the program needs. Generally, if the user is to supply more than one piece of information, use @...GET and READ. If the user is to respond to a one-line prompt, use ACCEPT, INPUT, or WAIT.

@...GET ...READ

Use @...GET followed by READ in on-screen forms to obtain user input. Each @...GET line displays a blank for user input, similar to the way APPEND works. If you want two blanks on the same screen row, issue two separate @...GET commands.

> **WARNING**
> The @...GET instructions work only if the program has previously initialized all the memory variables it uses in the @...GET commands.



With INTENSITY ON (the default), each @...GET line presents the user with a blank form to fill in, just as if the user were working in APPEND or EDIT. @...GET uses the enhanced display to show each blank in inverse video, unless you've changed the enhanced display with SET COLOR TO. The fill-in blank is the exact length of the memory variable. You can also use SET DELIMITERS to show the boundaries of the fill-in blanks (see Chapter 8).

The way you want the user to give the program input determines how many READ lines you will need. Most of the time you present the user with a screen form that contains several blanks for filling in information. This type of setup enables the user to enter all the information and backtrack with the ↑ key to make any necessary changes. You only need one READ statement to handle all the individual GETs. Alternatively, you can have the user enter just one piece of information at a time if you use READ after each GET line.

Here's how the two situations differ in practice. You are working with three character type memory variables called *mfirst*, *mmiddle*, and *mlast*. Each is of a different length. They correspond to the actual fields in a name and address database file. You have initialized these variables in the program:

```
mfirst = SPACE(15)
mmiddle = SPACE(2)
mlast = SPACE(20)
```

When you want the user to input information for each variable, you set up on-screen instructions, which look something like this:

```
CLEAR
@ 1,0 SAY "Enter first name:"
@ 1,25 GET mfirst
@ 3,0 SAY "Enter middle initial:"
@ 3,25 GET mmiddle
@ 5,0 SAY "Enter last name:"
@ 5,25 GET mlast
READ
```

When you use many GETs and one READ, all the fill-in blanks appear at one time.

```
Enter first name:
Enter middle initial:
Enter last name:
```

Figure 6-4 How the above GETs look on the screen

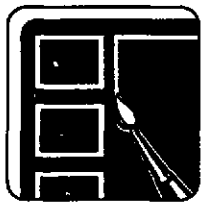
The user can enter each variable and backtrack with the ↑ key to correct a mistake. When the user has entered the last variable, the READ statement in the program STOREs all the input in the correct memory variables.

If you want the user to see only one fill-in blank at a time and enter each variable individually before going on to the next, set up the screen in this fashion:

```
CLEAR
@ 1,0 SAY "Enter first name:"
@ 1,25 GET mfirst
READ
@ 3,0 SAY "Enter middle initial:"
@ 3,25 GET mmiddle
READ
@ 5,0 SAY "Enter last name:"
@ 5,25 GET mlast
READ
```

In this case, the GET lines appear individually. The user must fill in the previous GET information and press the ↵ key before the next blank appears. However, the user cannot use the ↑ key to back up to previous memory variables, unless you use the READ SAVE option. READ SAVE lets you back up to previous GETs after the READ SAVE. You could rewrite the above example with READ SAVE instead of READ.

In the next chapter, you'll see how to set up templates with the GET command to ensure that the input from the user is correct. You'll also see how to include ranges for numeric or date input.



The default number of @...GET commands that you can use at one time is 128. This is more than sufficient for most purposes. You can't send the results of @...GET...READ operations directly to the printer. You must first use @...GET and READ to get input in memory variables. Then you use @...SAY lines to print the contents of the variables. You'll learn more about this in Chapter 12.

Make sure that the coordinates for the GET lines don't overlap the coordinates for the @...SAY lines. You can combine @...SAY and GET lines, but the SAY command must precede the GET command. Unless you provide other spacing within the string following the SAY command, the GET blank is separated by one space from the end of the string on the screen:

```
m_fname = SPACE(15)
@ 10,10 SAY "Enter first name:" GET m_fname
```

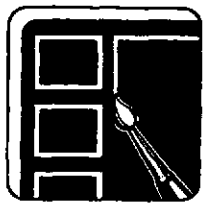
looks like this:

Figure 6-5 Position of a GET blank

You can have several different GETs appear on the same screen row, but you must still establish the GET instructions separately in your program:

```
@ 1,0 GET m_fname
@ 1,20 GET m_mi
@ 1,25 GET m_lname
READ
```

Figure 6-6 Several GETs on one row



Clearing the GETs

After the READ instruction STOREs the user input from the @...GET lines in the correct variables, it automatically clears the GETs. If you want the information to remain in the GETs, as in the example of individual GET blanks appearing on the screen, you can use the SAVE option on the READ line. If you do use this option, make sure that you also use the CLEAR GETS command in your program when you are finished with the GETs.

Multiple-Page Screens

You can design multiple-page screens for requesting user input by putting in READ commands to go from one screen to the next. You can have one READ line govern all the GETs in the multiple-page screens.

In Chapter 8, you'll learn another way to set up format files for screen forms, which also can have multiple screens.

Pressing ↵ to Continue

When the user is entering information in a screen form, dBASE III automatically positions the cursor at the next entry if the user has filled in the entire blank. Normally the user presses ↵, because the fill-in blank is slightly larger than most entries. The same thing happens when you're entering information in fields with APPEND or EDIT.

If you want the user to press the ↵ key to end each entry before going on to the next, use the command SET CONFIRM ON. This method gives the user a reminder to check the entry and make necessary corrections.

ACCEPT and INPUT

When the user is to enter information in several blanks, you usually use @...GET and READ. However, the program may need just one input, such as a response to a particular query. You can then use the ACCEPT, INPUT, or WAIT commands, which don't require a corresponding READ line. These commands also allow you to present a one-line prompt without an @...SAY construction.

ACCEPT and INPUT work in a similar fashion. They request a response from the user. After the user types in the response and presses ↵, the program STOREs the response in a memory variable. The memory variable can hold a maximum of 254 characters.

Unlike the @...GET setup, when you use ACCEPT or INPUT you don't have to initialize the variable that is to hold the user's response. ACCEPT and INPUT do this automatically. However, they don't show a blank form on the screen, as @...GET does.

The major difference between ACCEPT and INPUT is in the data type of the memory variable. ACCEPT receives only character strings and creates a character type memory variable. Even if the user types in a number, ACCEPT treats it as a character string. INPUT can accept character, numeric, date, or logical information and initialize a corresponding memory variable. However, with ACCEPT, the user doesn't have to delimit the input string with quotes; this is mandatory with INPUT. What's more, INPUT is a very flexible command; you can respond to it with the name of a field or memory variable.

You must include a prompt line in both these commands so that the user knows what the program is requesting. dBASE III PLUS displays this prompt line on the screen and waits for the user's input. Make sure that the prompt line is less than 80 characters long, the maximum width of the screen.

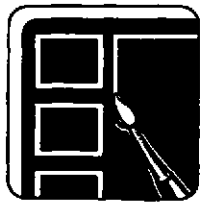
Here are examples of the ACCEPT and INPUT commands:

```
ACCEPT "What is your first name? " TO fname
```

This command requests a first name from the user and then STOREs the user's response in the memory variable *fname*. The user doesn't have to enter the response in delimiters.

```
INPUT "Enter the amount of the check " TO mcheck
```

This line requests a numeric answer from the user. dBASE III PLUS STOREs the input in a numeric type memory variable named *mcheck*.



Program code

ACCEPT 'Enter your choice ' TO mchoice

Screen	***Result in Memory***		
Enter your choice **A** ↵	MCHOICE	pub C "A"	
		1 variables defined,	3 bytes used
		255 variables available,	5997 bytes available
Enter your choice **2** ↵	MCHOICE	pub C "2"	
		1 variables defined	3 bytes used
		255 variables available,	5997 bytes available
Enter your choice **09/12/86** ↵	MCHOICE	pub C "09/12/86"	
		1 variables defined,	10 bytes used
		255 variables available,	5990 bytes available

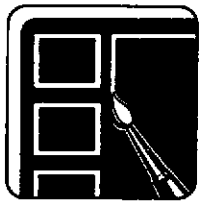
Program Code

INPUT 'Enter your choice ' TO mchoice

Screen	***Result in Memory***		
Enter your choice "A" ↵	MCHOICE	pub C "A"	
		1 variables defined,	3 bytes used
		255 variables available,	5997 bytes available
Enter your choice **2** ↵	MCHOICE	pub N 2	(2.00000000)
		1 variables defined	9 bytes used
		255 variables available,	5991 bytes available
Enter your choice CTOD('09/12/86') ↵			
	MCHOICE	pub D 09/12/86	
		1 variables defined,	9 bytes used
		255 variables available,	5991 bytes available

Table 6-1 The difference between ACCEPT and INPUT

When requesting numeric, date, or logical information with INPUT, give the user a prompt describing what kind of information to enter. This ensures that the information is in the correct form and that the memory variable initialized with the INPUT command is correct. So you could amend the above INPUT situation to look like this:




```
@ 10,0 SAY "Please enter the amount of the check"
?
INPUT "using the correct decimal places .... $ " TO mcheck
```

Note the use of the ? command to space the INPUT line from the @...SAY line. The screen looks like this:

```
Please enter the amount of the check
using the correct decimal places .... $
```

Figure 6-7 Using INPUT with @...SAY

You include the dollar sign in the prompt line to indicate that the user need not enter it.

> **NOTE**
> A special case occurs when the user presses only the ↵ key in response to any prompt for input, whether from an ACCEPT, INPUT, or GET...READ construction. You'll learn more about this in Chapter 9.

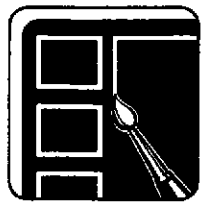
Wait

The WAIT command puts the following message on the screen:

Press any key to continue...

WAIT gives you the option to put pauses in your program so that the user can read information on the screen or decide when to go on. The program won't continue until the user has pressed a key. It's good practice to SET ESCAPE OFF so that the user doesn't accidentally end the program by pressing the **Esc** key.

If you don't like dBASE III PLUS's WAIT message, you can set up your own message. For example, in the Cancl.prg module for canceled checks, the program includes this WAIT line:



```
WAIT SPACE(19)+"Press any key to return to the Main menu "
```

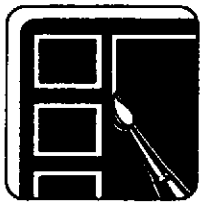
Normally, the WAIT message starts at column 0. Here, the SPACE() function is used to center the message on the screen:

```
                    Press any key to return to the Main menu
```

Figure 6-8 Setting up the WAIT prompt

Unlike ACCEPT or INPUT, WAIT doesn't have to STORE the user's input to a memory variable. The command simply allows the program to wait for the user to press a key before continuing. However, you can have WAIT initialize a variable:

```
WAIT "Press R to return to the Main menu, any other key " + ;
     "to continue" TO mchoice
IF UPPER(mchoice) = 'R'
   RETURN
ELSE
   * (commands)
ENDIF
```



NOTE
You can't use the screen coordinates with ACCEPT, INPUT, and WAIT. As you've seen, you can use the ? command to position the ACCEPT, INPUT, and WAIT lines relative to other lines on the screen.

Chapter 7
USING TEMPLATES AND RANGES

Your program will seem easy to use only if you make it easy to use. One way to do that is to supply helpful prompts and messages. Another way is to use screen templates and ranges to control user input.

What This Chapter Covers

In this chapter you will learn:

- Why it's advisable to use templates for user input
- How templates work in PICTURE clauses
- The difference between template symbols and template functions
- How to restrict the range of numeric or date input
- How to change data displayed with ?, DISPLAY, LIST, and REPORT with the TRANSFORM() function
- Points to consider when designing screens

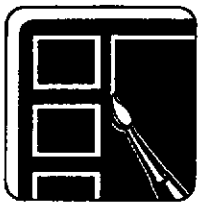
Preparing for This Chapter

Besides having a general knowledge of dBASE programming, understand how to set up screen forms and how to use string, numeric, and date functions.

How Templates Work

A template acts as a filter to ensure that the user types in acceptable data. It lets the program restrict the display of data and check for valid user input. For example, if the user is to type in a numeric amount, the template can restrict user input to numbers only.

A template is contained in a PICTURE clause, which is an extension of @...SAY and @...GET command lines. A PICTURE clause contains a combination of one or more template symbols and functions. A template symbol restricts the display or input of individual keystrokes. Because each symbol represents one position in the @...GET blank, you must include a symbol for every position in the blank. A template function, which begins with the @ sign, need only be typed once at the beginning of the PICTURE clause.



NOTE
When used with @...SAY lines, templates do not change the actual information in fields or memory variables. They only change their screen appearance. When used with GET lines, templates restrict user input. You can't use templates with the ACCEPT, INPUT, or WAIT commands.

Template Symbols

You can see template symbols at work in the Add.prg module of the checkbook management system. They're used to control the screen display of the *subtotal* memory variable:

```
@ 16,32 SAY subtotal PICTURE "9999999.99"
```

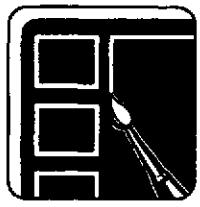
The number 9 here isn't a value. It's the template symbol that instructs dBASE III PLUS to show only numbers. There must be a template symbol for every position in the PICTURE clause. The decimal point in the template instructs dBASE III PLUS to show the *subtotal* variable with two decimal places.

For numeric variables, the default size of the GET is ten characters. For longer numbers, use the correct PICTURE clause:

```
number = 0
* get the number with 13 places
@ 10,10 GET number picture "9999999999999"
```

Another frequently used template symbol is the !, which automatically converts the user's input to upper case. In the following lines, the program requests that the user enter y or n and converts this input to upper case:

```
STORE " " TO choice
@ 10,5 SAY "Do you want to add more records? (y/n)"
@ 10,44 GET choice PICTURE "!"
READ
```



You limit the size of a GET response by including the maximum number of characters allowed. In the above example, the user can type only a one-character response. The size of user input is determined by the size of the PICTURE template. In this next example, the program accepts input of up to ten characters for a memory variable that is to contain first name data:

```
@ 10,10 GET mfirst PICTURE "AAAAAAAAAA"
```

Because names normally don't contain numbers, the A symbol tells dBASE III PLUS to accept alphabetic characters only. However, if you have a street variable, you would not want to filter out numbers. You can use the X symbol instead:

```
@ 10,10 GET mstreet PICTURE "XXXXXXXXXXXXXXXXXXXXXXXXX"
```

This symbol accepts any character in the PICTURE clause.

You can control the way numeric fields or variables look on the screen with a variety of template symbols. For example, the $ symbol displays dollar signs in the place of leading zeros in numeric fields or variables only:

```
@ 10,10 SAY mamount PICTURE "$$$$$$$.$$"
```

would display the amount 45.00 as:

$$$$$45.00

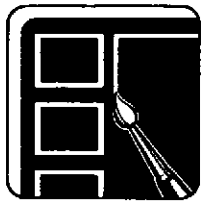
To display just one dollar sign, you would have to convert the numeric variable to a string. Because this variable may contain numbers of different lengths, the length of the string will vary. The following module assumes that the *mamount* variable contains a number with five total places and two decimal places:

```
@ 10,10 SAY "$"+ STR(mamount,5,2)
```

This would display the amount 45.00 as:

$45.00

Suppose you want to supply commas in PICTURE clauses with @...GET commands. Here's how:




```
mcost = 0.00
@ 10,10 GET mcost PICTURE "999,999.99"
READ
```

When using date fields or variables, dBASE III PLUS automatically validates that the user's input is a correct date and prompts for a correct date if the user has typed an incorrect date. If you want to use a string that you'll later convert to a date, employ a template like this:

```
mtoday = SPACE(8)
@ 10,10 GET mtoday PICTURE "99/99/99"
```

The above example supplies the slashes and indicates to the user where the entries should be. It ensures that the user types in only numbers, but it won't validate this as a correct date. Your program has to do this. For this reason, avoid using character variables for dates.

Another useful template symbol is Y, which restricts input into logical fields or variables to Y for yes or N for no. In this example, *decision* is a logical variable:

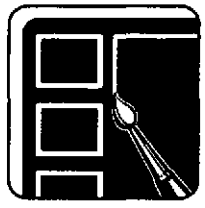
```
@ 10,10 SAY "Are you finished? (Y/N)" GET decision;
    PICTURE "Y"
READ
```

Template Functions

Template functions govern the display of the entire PICTURE clause, not just individual positions in the PICTURE clause. You can use a mixture of template symbols and functions in a PICTURE clause, but the function or functions must be the first item in the clause. Use an @ sign directly in front of the function, and separate the function from the rest of the PICTURE clause by a space. To avoid confusion with the other @ command on the line, you can use the word FUNCTION as a substitute for the function designator, @.

For example, dBASE III PLUS normally right-justifies numeric data, but you can display numeric information as left-justified on the screen with the B function:

```
@ 10,10 SAY mamount PICTURE "@B 9,999,999.99"
```



The , symbol shows commas when the figures are over three digits, that is, for thousands and millions. You can write the same line like this:

```
@ 10,10 SAY mamount FUNCTION "$" PICTURE "9,999,999.99"
```

If you have several functions, you only need one @ sign for all of them. For instance, if you're writing an accounting program, you can have dBASE III PLUS display amounts as debits or credits. If the *mamount* variable contains the number −45.00, the command line:

```
@ 10,10 SAY mamount PICTURE "@XC 999.99"
```

gives the result:

45.00 DB

Similarly, the result for *mamount* containing 55.00 would be:

55.00 CR

Two other frequently used template functions are (, which encloses negative numbers in parentheses, and Z, which displays zero numeric values as a blank string.

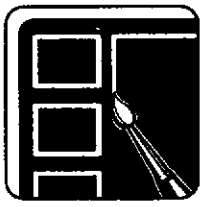
There are template functions that allow you to change the display of dates to either American or European format:

```
mdate = CTOD("09/12/86")
@ 10,10 SAY mdate PICTURE "@E"
```

This returns *mdate* in BRITISH format, that is, **12/09/86**.

The R function instructs dBASE III PLUS that there are literal characters in the PICTURE clause. These characters are not interpreted as symbols. For example, you have a variable called *mtitle* which contains the following string, **DEPOSITS**, and you want to display this variable differently, with each letter separated by a space. Here's how:

```
@ 10,10 SAY mtitle PICTURE "@R X X X X X X X X"
```



dBASE III PLUS interprets the spaces literally as spaces. It substitutes each letter in the contents of *mtitle*, DEPOSITS, only when it sees a valid template symbol. The result is:

D E P O S I T S

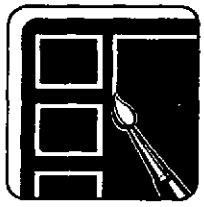
With GETs, the literals in the PICTURE clause do not become part of the variable.

The new S function allows horizontal scrolling in @...GET lines. This is most helpful when you only have a certain amount of space on the row for a blank, but the information the user types into the blank may be longer. The S function scrolls the input in the blank to make room for more input, without increasing the size of the blank. You supply a number next to the S, which determines the width of the scrolling region, and template symbols to restrict the input further.

For example, the following module displays a prompt and a blank that is ten characters in length. The scrolling region is eight characters wide. Because the variable *mlast* is 20 characters long, the user can still enter a longer string into the blank:

```
CLEAR
* Initialize variable
mlast = SPACE(20)
@ 10,20 SAY "Enter a last name of any length:" ;
   GET mlast PICTURE "@S8 !!!!!!!!!!!!!!!!!!!!"
READ
```

When the user enters a name that is more than eight characters long, the input scrolls left to make room in the blank for the extra characters.



Blank is 10 characters wide

van Beetho

Entry scrolls to make space

van Beethoven

Figure 7-1 The scrolling function

The name is too long for the blank, so it scrolls left as the user types more characters.

In the next chapter, you'll see how to use this function to restrict user input within a box drawn on the screen.

NOTE
Template symbols and functions relate to specific data types. If you use a template symbol or function with an incorrect data type, dBASE III PLUS disregards your instructions. For instance, the A symbol works only with character type data. In addition, some symbols and functions are only applicable for displaying data. For example, it makes no sense to use the C and X functions for validating input.

There are many more template symbols and functions listed under the @ command in Chapter 5 of *Using dBASE III PLUS*. Experiment a little with all of them to see how they work.

Templates cannot possibly check for all input errors, however, so they do not take the place of thorough error-trapping routines after the user has entered data. At the very least, you'll want your program to request the user to verify that all newly input information is correct. This is the topic of Chapter 9.

Limiting the Range of Numeric and Date Input

You can also limit the entry of numeric and date information with the RANGE option. This works with @...GET commands to provide inclusive upper and lower limits. Recall that dates are actually special types of numbers. RANGE instructs dBASE III PLUS to accept only numeric or date information that is within the stipulated RANGE.

For example, you want to restrict the user's input to an amount variable to be between 25.00 and 100.00 for the price of an item:

```
@ 10,10 GET amount RANGE 25,100
```

Similarly, you may want to restrict merely the lower range:

```
@ 10,10 GET amount RANGE 25,
```

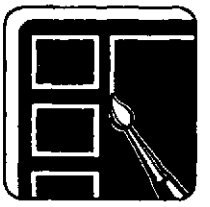
Notice that you still have to add the comma at the end of the RANGE command. The Check.prg module in the checkbook management system uses this form of the RANGE clause. A lower RANGE of 0 prevents the input of negative check numbers.

If you want to restrict only the upper range:

```
@ 10,10 GET amount RANGE ,100
```

For date variables, you must convert the date ranges from character strings to actual dates first. The following example allows entry of dates in the month of April 1985 only into the variable *m_date* :

```
@ 10,10 GET m_date RANGE CTOD("04/01/85"), CTOD("04/30/85")
```



Once you've set up ranges for numeric or date information, if the user types numeric or date data that doesn't fall within the RANGE, dBASE III shows the acceptable range on the status line or scoreboard, depending on the SET STATUS and SET SCOREBOARD commands, and instructs the user to press the **Spacebar** to clear the entry and try again.

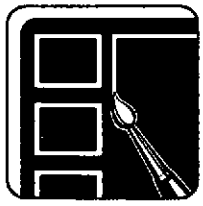
Transforming Displays

Because they are part of PICTURE clauses, templates only work with @...SAY and @...GET lines. There is also the TRANSFORM() function, which works with the following dBASE III PLUS commands to display data: ?, ??, DISPLAY, LABEL, LIST, and REPORT.

You can use the template symbols and functions with TRANSFORM(), but you don't need the word PICTURE. Include the field or variable name in the parentheses first, followed by a comma, then the symbols or functions. In the following example, Client is a character field containing client names. Using the R function, the @ symbol, and X symbols with the LIST command displays the names with each letter separated by a space. (Client is a character field containing client names.)

```
USE Accounts
LIST TRANSFORM(Client,'@R X X X X X X X X X X X X X X X')
```

Don't forget to enclose the symbols and functions in delimiters. Because it allows for the display of fields with special effects, the TRANSFORM() function is very useful when you're setting up reports with dBASE III PLUS's CREATE REPORT command. As with templates, the TRANSFORM() function does not change the actual data in the database file.



Designing Screens

Now that you know about screen coordinates and templates, here are some points to keep in mind when you design your screen forms and reports:

- Do screen forms mimic users' actual paper forms?
- Do the screens present a consistent appearance?
- Are the prompts and messages displayed in the same area of the screen for each form?
- Is enough information given?
- Are the help screens helpful?
- Have you eliminated technical jargon?
- Can users change their minds after making a menu choice?

In the next chapter, you'll take a look at ways to make your screen forms as appealing as possible.

Chapter 8
FANCIER SCREEN FORMS AND FORMAT FILES

How you design your screens is up to you, but remember that the friendliness of the user interface affects how the end user reacts to your program. This chapter gives you some useful tips to make your screen forms aesthetically pleasing. You'll also learn about format files and how to work with memo fields in programs.

What This Chapter Covers

This chapter discusses the following:

- How to change the appearance of @...GET blanks on the screen
- What relative addressing does
- How to center, right-justify, or insert strings within a string
- How to save yourself typing by using memory variables for prompts and the REPLICATE() function to repeat characters
- How to draw special characters and boxes on the screen
- How to turn the screen off temporarily
- What format files do and how to use them
- How to deal with memo fields and fast typists

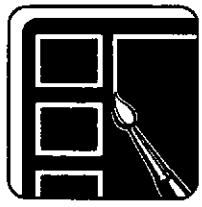
Preparing for This Chapter

You should have a basic understanding of dBASE programming, conversion functions, and how to set up screen displays.

Changing the Appearance of GET Blanks

When you enter information in a field or on-screen form with a full-screen edit command such as APPEND, EDIT, or GET, dBASE III PLUS shows the field or the variable using the enhanced display.

Unless changed by SET COLOR or SET INTENSITY, the enhanced display is inverse video. The size of the blank delimits the size of the field. If you prefer dBASE III PLUS to delimit the fields with colons, use SET DELIMITERS ON. The default is for DELIMITERS to be OFF.



You can SET other characters, such as square brackets, [], or braces, {}, as delimiters, but you must enclose the new delimiters in single or double quotes:

```
SET DELIMITERS TO "[]"
```

Once you have SET other DELIMITERS, you have to turn them on with the command SET DELIMITERS ON. So you will probably use these two commands together. If you wish to revert to dBASE III PLUS's default delimiter, type SET DELIMITERS TO DEFAULT.

Even if you change the delimiters, dBASE III PLUS still shows the fields using the enhanced display. To use the standard display for @...GET fields, SET INTENSITY OFF. The default is ON.

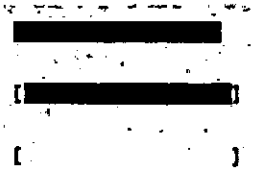
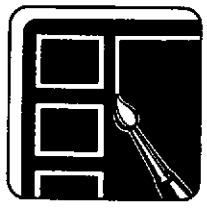


Figure 8-1 Three different GET blanks

The first GET blank uses the enhanced display only. With SET DELIMITERS TO '[]', the second GET blank uses the enhanced display and is delimited with brackets. With SET INTENSITY OFF, the third GET blank is delimited with brackets only.



Relative Addressing

Usually you know in advance where to position @...SAY or GET lines on the screen. However, occasionally the screen coordinates may depend on the position of other screen rows, which may change during the program. Say, for example, that you want the program to show certain fields and then a prompt two rows below them. You don't know exactly how many fields will be displayed; there may be two or ten. So how do you know on what row your prompt is to appear?

You can use a technique known as relative addressing. This means that the position of one screen coordinate is dependent on, that is, relative to, the position of a previous coordinate.

dBASE III PLUS uses the two functions ROW() and COL() for relative addressing. The ROW() function returns the value of the row on which the cursor is currently located, while the COL() function returns the value of the column position. If you have a command like this:

```
@ 5,10 SAY "Hello there!"
```

then the current ROW() is 5 and the current COL() is 22, the end of the displayed line. You can then do this:

```
@ ROW()+2,10 SAY "How ARE you?"
```

The new coordinate here is two rows down from the previous row. Similarly,

```
@ ROW()+5,COL()-2 SAY "Well, I hope"
```

```
Hello there!

How ARE you?


  Well, I hope
```

Figure 8-2 How relative addressing works

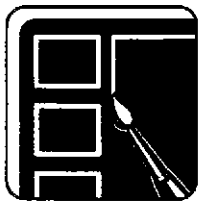


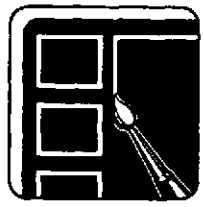
Figure 8-2 shows what the three examples look like on the screen.

No matter what number ROW() contains, the command ROW()+5 always positions the prompts five rows down. The only time you can't use the ROW() and COL() functions is directly after a READ statement. READ always resets the value of ROW() to 23 and the value of COL() to 0.

The checkbook management system uses a different type of relative addressing to put an asterisk next to the the user's choice in several of the submenus:

```
* put asterisk on screen next to choice
@ 6+VAL(choice2),24 SAY "*"
```

Note the use of the VAL() function to change the variable *choice2* to a number for the column coordinate.



TIP
You can also use numeric memory variables in @...SAY lines, which can make your programs more readable. For example, in the Taxcodes.prg module, the program initializes a memory variable called *line* and then uses a DO WHILE loop to display all the tax codes from the Tax.dbf file, with each code on a new line. The line number is incremented by one before another repetition of the loop:

```
* display all tax codes and titles
line=4
DO WHILE .NOT. EOF()
   @ line,28 SAY Code
   @ line,32 SAY Title
   line=line+1
   SKIP
ENDDO
```

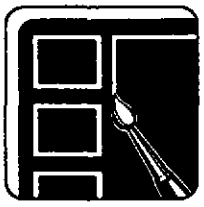
Because the program doesn't know how many tax codes are in the file, it uses the relative addressing technique and a simple counter to display them.

Centering a String

To center a string with a known length, subtract the length from 80, the total number of columns on the screen, then divide by two to find the starting coordinate. Occasionally, you may need to center a string in your program's display, without knowing what the string length will be. An example would be when you want to show what the user has typed. Here's how you can center any string.

First, get the user's input with any standard input command, such as @...GET or ACCEPT:

```
ACCEPT "Enter the client's first name " TO mfirst
```



Then, trim the trailing blanks from the variable and take its length, divide by two, and subtract it from 40, which is the exact center coordinate of the line, to get the center position. STORE this into a new memory variable, *mcenter*:

```
mfirst = TRIM(mfirst)
mcenter = 40 - LEN(mfirst)/2
```

You can then use the contents of *mcenter* in a screen coordinate:

```
@ 20,31 SAY "You have entered:"
SET COLOR TO N/W
@ 22,mcenter SAY mfirst
SET COLOR TO W/N
```

Notice the special effect of inverse video display that the example uses.

Right-Justifying a String

You can use a similar technique to right-justify a string of any length. Here is a simple module that right-justifies a memory variable called *string*. The variable *width* contains 80, the total width of the screen, and the variable *currentrow* contains the row on the screen where the text is displayed:

```
STORE 80 TO width
STORE 5 TO currentrow
@ currentrow,width - LEN(string) SAY string
```

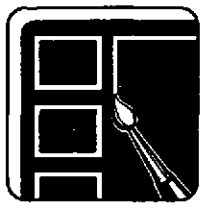
The vertical coordinate of the position is determined by adding the starting column position, 0, to a number which represents the remaining number of blank columns. This number depends on the width of the screen and the length of the string.

Stuffing a String

Use the STUFF() function to insert a string within a string. STUFF() either adds a new string to the existing string without changing it, or replaces part of the string with the new string.

For example, you use a memory variable called *prompt2*, which contains the following string:

```
Type X to Exit
```



and you want to change it to:

```
Type C to Continue or X to Exit
```

First, STORE the new string to a memory variable:

```
extra = "C to Continue or "
```

Note the space at the end of the string. You could then do this with the STUFF() function:

```
STUFF(prompt2, 6, 0, extra)
```

The first argument in the parentheses, *prompt2*, is the original string. Next follows the starting position, 6, in the old string where the new string will be inserted. The second number, 0, is the number of characters to remove from the old string. Here, 0 means that you don't want to replace the first string with the second but merely to add the second to the first. The last piece of information, *extra*, is the string that is inserted in the first string. The above line is equivalent to:

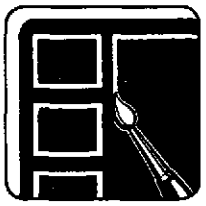
```
SUBSTR(prompt2,1,5) + extra + SUBSTR(prompt2,6)
```

STUFF() is also useful when you want to REPLACE only part of a character field without having to reconstruct the entire field.

Graphics and Other Special Characters

For special visual effects, you can use @...SAY with any ASCII character. The checkbook management system uses graphics characters in the ASCII set to give its menus a little pizzazz.

The best way to draw boxes in dBASE III PLUS is to use the @...TO command with an upper left and lower right range for the box:




```
@ 1,0 TO 5,79
```

draws a box from column 0 on row 1 to column 79 on row 5. You must supply the two corners. The box contains a single border, but you can draw a double-bordered box with:

```
@ 1,0 TO 5,79 DOUBLE
```

dBASE III PLUS draws the box from top to bottom of the screen. If you use MODIFY COMMAND to look at the Menumask.prg, you can follow how the program displays the boxes in the main menu.

If you use a range with the same row, then dBASE III PLUS draws a horizontal line:

```
@ 1,0 TO 1,79
```

Similarly, using the same column in the range instructs dBASE III PLUS to draw a vertical line:

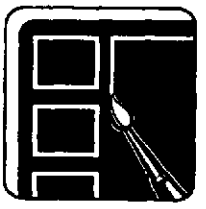
```
@ 1,0 TO 23,0 DOUBLE
```

To clear a box from the screen, use @...CLEAR TO with the top left and bottom right coordinates:

```
@ 1,0 CLEAR TO 5,79
```

With the S template function, you can restrict user input to a box on the screen and have long input scroll, as in this module:

```
SET TALK OFF
name = SPACE(25)
CLEAR
@ 5,10 TO 8,29
@ 6,12 GET name PICTURE "@S15 XXXXXXXXXXXXXXXXXXXXXXXXX"
READ
```



NOTE
There are some special characters in the ASCII range 1 through 31 that can't be displayed with @...SAY. When you want to ring the bell in a program, you can't use @...SAY. You must use the ?? command:

```
@ 15,0 SAY "What do you want to do next?"
?? CHR(7)
@ 17,0 SAY "(A)dd a record, or (D)elete the record"
```

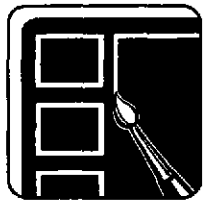
Because you don't have to position the bell on the visible screen, using ?? does not pose a big problem. The computer works very fast, and when the above line appears on line 15, the user hears the bell immediately. If you plan to use special characters from the ASCII set in reports, make sure that your printer can print them.

Although you can use the **Alt** key with the numeric keypad to type special ASCII characters on the screen, don't use this method to type them in program files. When you edit these files with MODIFY COMMAND, dBASE III PLUS won't save the special characters. Use the CHR() function instead. However, you can STORE the actual characters in memory files (see the next section).

Don't Retype Screen Prompts

If you use the same screen prompts consistently, you can set them all up in a memory file. Then, after the program RESTOREs the file, they will reside in memory throughout the program. When you're writing the program, you won't have to type each prompt every time it appears. For instance, you will use the following prompt frequently:

Is everything correct? (y/n)



You can STORE it in a memory variable and call it *mprompt1*. Whenever you need to display the prompt on the screen, use @...SAY:

```
@ 10,10 SAY mprompt1
```

You may have noticed that the Chkbook.mem file contains horizontal and vertical lines made up of special characters. The program uses these memory variables, *svert* for single vertical lines and *shoriz* for single horizontal lines, in many screen displays. Recall that the main program module RESTOREs this file at the beginning of the run. In the Clrdep.prg module, for example, you'll see these lines:

```
@ 1,20 SAY "Clear Deposits with Bank Statement"
@ 2,20 SAY LEFT(shoriz,34)
@ 3,7 SAY "Date       Amount"
@ 4,5 SAY LEFT(shoriz,8)+"   "+LEFT(shoriz,10)
```

They appear like this on the screen:

```
                    Clear Deposits with Bank Statement
  Date      Amount
  ----      ------
```

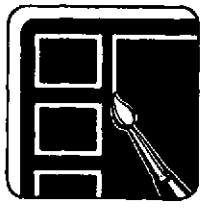
Figure 8-3 Using part of a memory variable for displays

The program uses the LEFT() function to display only a part of the *shoriz* variable as underlining for the title lines. This is a great way to reuse the same variable any number of times.

Repeating Characters

Another useful function is REPLICATE(). This function repeats the same character a stipulated number of times. For example, in Menumask.prg, which presents the main menu at the beginning of the checkbook management system, the program uses REPLICATE() to display CHR(176) 75 times:

```
@ 22,2 SAY REPLICATE(CHR(176),75)
```



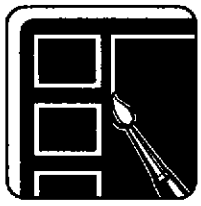
You must first type the character that you want to repeat, either with the CHR() function, a character string, or any valid character expression, then a comma, and finally the number of repetitions.

Pseudo-Passwords

You may wish to have the user enter a password before being allowed to use your program. Generally, you wouldn't want this password to appear on the screen when the user types it, so you must use the SET COLOR command to hide the display. Because GET blanks are governed by the enhanced display, you must change its color. A sample module to request a password and temporarily disable the screen might look like this:

```
CLEAR
STORE SPACE(5) TO password
* set enhanced display to black on black
SET COLOR TO ,N/N
@ 5,0 SAY "Please enter your password: " GET password
READ
* (... commands to check for correct password)
* revert to normal colors
SET COLOR TO
@ 7,0 SAY "Thank you"
```

WARNING
This method of password protection is not foolproof. Furthermore, if you are programming in dBASE III PLUS for users on a local area network, you need much more protection than this. There are special commands for maintaining security on networks. Refer to *Networking dBASE III PLUS* for more information.



Format Files

If you plan to use the full-screen commands APPEND, EDIT, CHANGE, or INSERT in your programs, you could also set up format files, which determine how the field data looks on the screen. These format files merely change the position of the information to mimic the user's own forms. They offer an alternative to dBASE III PLUS's default presentation.

Format files work only with the above full-screen commands, not with other commands that you might include in your programs. Format files only contain @...SAY and @...GET commands for the various fields in the database file. A format file automatically CLEARs the screen when you issue a full-screen command, such as APPEND, after SETting the FORMAT TO the format filename.

Creating Format Files

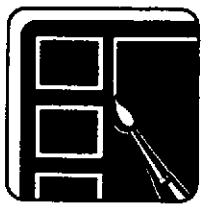
There are several ways to create and edit format files. The easiest method is to use CREATE SCREEN <screen filename>. This command allows you to USE a database file, presents you with the fields in the file, and lets you choose which fields to display and position anywhere on the screen. It then sets up a format file for this screen.

CREATE SCREEN actually creates two files, a screen file with the extension .scr and a format file with the extension .fmt. You can use MODIFY SCREEN to make changes to the screen file, which then updates the corresponding format file.

You can also use MODIFY COMMAND or another word processor to write format files, but you must provide the file extension .fmt. For example, the following command would be used to create or edit the format file called New.fmt:

```
MODIFY COMMAND New.fmt
```

If you use your own word processor, make sure that you save the format file as an ASCII text file, or a DOS text file in Framework II.



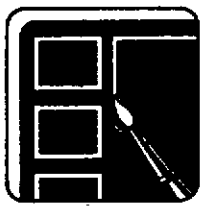
You must first type the character that you want to repeat, either with the CHR() function, a character string, or any valid character expression, then a comma, and finally the number of repetitions.

Pseudo-Passwords

You may wish to have the user enter a password before being allowed to use your program. Generally, you wouldn't want this password to appear on the screen when the user types it, so you must use the SET COLOR command to hide the display. Because GET blanks are governed by the enhanced display, you must change its color. A sample module to request a password and temporarily disable the screen might look like this:

```
CLEAR
STORE SPACE(5) TO password
* set enhanced display to black on black
SET COLOR TO ,N/N
@ 5,0 SAY "Please enter your password: " GET password
READ
*( ... commands to check for correct password)
* revert to normal colors
SET COLOR TO
@ 7,0 SAY "Thank you"
```

WARNING
This method of password protection is not foolproof. Furthermore, if you are programming in dBASE III PLUS for users on a local area network, you need much more protection than this. There are special commands for maintaining security on networks. Refer to *Networking dBASE III PLUS* for more information.



Format Files

If you plan to use the full-screen commands APPEND, EDIT, CHANGE, or INSERT in your programs, you could also set up format files, which determine how the field data looks on the screen. These format files merely change the position of the information to mimic the user's own forms. They offer an alternative to dBASE III PLUS's default presentation.

Format files work only with the above full-screen commands, not with other commands that you might include in your programs. Format files only contain @...SAY and @...GET commands for the various fields in the database file. A format file automatically CLEARs the screen when you issue a full-screen command, such as APPEND, after SETting the FORMAT TO the format filename.

Creating Format Files

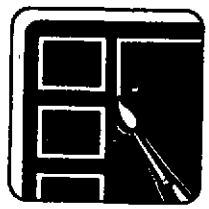
There are several ways to create and edit format files. The easiest method is to use CREATE SCREEN <screen filename>. This command allows you to USE a database file, presents you with the fields in the file, and lets you choose which fields to display and position anywhere on the screen. It then sets up a format file for this screen.

CREATE SCREEN actually creates two files, a screen file with the extension .scr and a format file with the extension .fmt. You can use MODIFY SCREEN to make changes to the screen file, which then updates the corresponding format file.

You can also use MODIFY COMMAND or another word processor to write format files, but you must provide the file extension .fmt. For example, the following command would be used to create or edit the format file called New.fmt:

```
MODIFY COMMAND New.fmt
```

If you use your own word processor, make sure that you save the format file as an ASCII text file, or a DOS text file in Framework II.



NOTE
If you use CREATE or MODIFY SCREEN to set up a format file and later edit this file with MODIFY COMMAND or another word processor, the screen (.scr) file is not updated.

Here is the New.fmt file, which displays the First, Middle, Last, Street, City, State, and Zip fields in a database file called Names.dbf, with screen prompts and @...GET lines for receiving user input:

```
* NEW.FMT - format file for Names.dbf file
@ 1,5 SAY "First name:"
@ 1,20 GET First
@ 3,6 SAY "Last name:"
@ 3,20 GET Last
@ 5,9 SAY "Street:"
@ 5,20 GET Street
@ 7,11 SAY "City:"
@ 7,20 GET City
@ 9,10 SAY "State:"
@ 9,20 GET State
@ 11,12 SAY "Zip:"
@ 11,20 GET Zip
```

Here is how this format file looks on the screen:

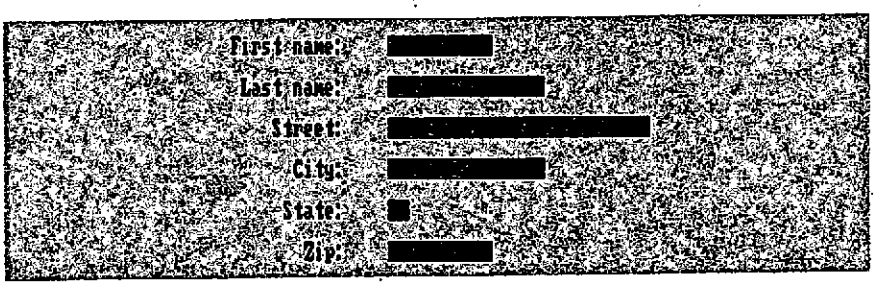


Figure 8-4 Screen appearance of the New.fmt file

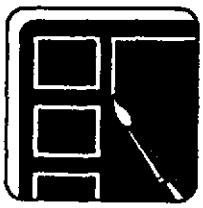
Because the entire format file is governed by whatever full-screen editing command you're using at the time, you don't need a READ line for the GETs unless you plan to have multi-page screen formats, discussed below.

Using Format Files

Once you have set up a separate format file, you can use it from within a program with the SET FORMAT command. So, to open a database file called Names.dbf and use the New.fmt file with the APPEND command, you write the following code in your program:

```
USE Names
SET FORMAT TO New
APPEND
```

Similar to the way MODIFY COMMAND assumes a program file, the SET FORMAT command assumes the file extension .fmt. The program opens the format file, but the screen form doesn't appear until the program also issues one of the full-screen commands, such as APPEND. To close a format file, use CLOSE FORMAT or SET FORMAT TO without a filename.



TIP
Because the total number of files that can be open at one time is restricted, get in the habit of inserting a CLOSE FORMAT command to close every format file immediately after the program is finished with the file.

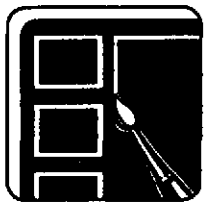
Multiple-Page Screen Forms

dBASE III PLUS also allows you to set up multiple-page screen forms with format files. This way, you can spread the information over several screen forms, allowing you to create more aesthetically pleasing screen displays.

Use the READ command in a format file wherever you want your program to CLEAR the screen and show a new on-screen form. Recall that normally you don't use READ in single-page format files. Make sure, of course, that you establish the correct screen coordinates for the next form.

With multi-page screen forms, dBASE III PLUS automatically shows the next page of the form when the user has filled in all GET lines in the previous page. The user can also page backward through the forms with the **PgUp** key and forward with the **PgDn** key.

NOTE
This use of READ for multi-page screen forms only works with format files. In program files, READ functions normally. No matter how many format file screens you string together with READ, you still can have only a maximum of 128 total GETs.



Working with Memo Fields

Because memo fields are maintained by dBASE III PLUS in a separate file, they are treated differently. For instance, you can't STORE the contents of a memo field in a memory variable, and you can't display memo fields with @...SAY lines. Although full-screen commands such as EDIT or APPEND work with memo fields, the user sees only the field name and may not know how to open the memo field.

So, a good way to add to or change the contents of a memo field from within a program is with a format file and either the EDIT or CHANGE command with the scope option. The format file guides users by instructing them to press **Ctrl-PgDn** to open the memo field for editing and **Ctrl-PgUp** to finish. Here is an example of a setup for changing a memo field. The database file, Names.dbf, has a memo field named Notes. The format file to add to or change this memo field, Notechng.fmt, looks like this:

```
* Notechng.fmt - format file for changing memo field
@ 8,21 SAY "Press <Ctrl><PgDn> to edit Notes"
@ 10,17 SAY "To save your changes, press <Ctrl><PgUp>"
@ 14,16 SAY " <RETURN> takes you back"
@ 12,34 GET Notes
```

The commands in the program for changing the memo field Notes are below. The user has already supplied a record number in the *mrecnum* memory variable.

```
USE Names
GOTO mrecnum
SET FORMAT TO Notechng
CHANGE NEXT 1 FIELD Notes
CLOSE FORMAT
USE
```

After typing in the corrections and pressing **Ctrl-PgUp** to leave the memo field, the user then presses ↵ to leave the format file. The program updates the memo field, saves the database file, closes the format file, and continues.

TIP
Because the total number of files that can be open at one time is restricted, get in the habit of inserting a CLOSE FORMAT command to close every format file immediately after the program is finished with the file.

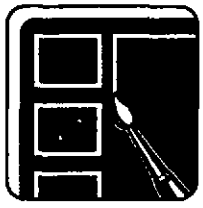
Multiple-Page Screen Forms

dBASE III PLUS also allows you to set up multiple-page screen forms with format files. This way, you can spread the information over several screen forms, allowing you to create more aesthetically pleasing screen displays.

Use the READ command in a format file wherever you want your program to CLEAR the screen and show a new on-screen form. Recall that normally you don't use READ in single-page format files. Make sure, of course, that you establish the correct screen coordinates for the next form.

With multi-page screen forms, dBASE III PLUS automatically shows the next page of the form when the user has filled in all GET lines in the previous page. The user can also page backward through the forms with the **PgUp** key and forward with the **PgDn** key.

NOTE
This use of READ for multi-page screen forms only works with format files. In program files, READ functions normally. No matter how many format file screens you string together with READ, you still can have only a maximum of 128 total GETs.



Working with Memo Fields

Because memo fields are maintained by dBASE III PLUS in a separate file, they are treated differently. For instance, you can't STORE the contents of a memo field in a memory variable, and you can't display memo fields with @...SAY lines. Although full-screen commands such as EDIT or APPEND work with memo fields, the user sees only the field name and may not know how to open the memo field.

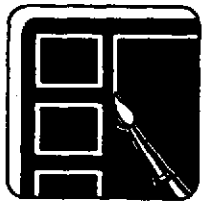
So, a good way to add to or change the contents of a memo field from within a program is with a format file and either the EDIT or CHANGE command with the scope option. The format file guides users by instructing them to press **Ctrl-PgDn** to open the memo field for editing and **Ctrl-PgUp** to finish. Here is an example of a setup for changing a memo field. The database file, Names.dbf, has a memo field named Notes. The format file to add to or change this memo field, Notechng.fmt, looks like this:

```
* Notechng.fmt - format file for changing memo field
@ 8,21 SAY "Press <Ctrl><PgDn> to edit Notes"
@ 10,17 SAY "To save your changes, press <Ctrl><PgUp>"
@ 14,16 SAY "<RETURN> takes you back"
@ 12,34 GET Notes
```

The commands in the program for changing the memo field Notes are below. The user has already supplied a record number in the *mrecnum* memory variable.

```
USE Names
GOTO mrecnum
SET FORMAT TO Notechng
CHANGE NEXT 1 FIELD Notes
CLOSE FORMAT
USE
```

After typing in the corrections and pressing **Ctrl-PgUp** to leave the memo field, the user then presses ↵ to leave the format file. The program updates the memo field, saves the database file, closes the format file, and continues.



You can DISPLAY or LIST the contents of a memo field on the screen. You can also use the ? command to display memo fields. Remember that these commands don't allow you much flexibility in positioning the memo information on the screen. However, SET MEMOWIDTH is one command that can help you. It changes the default line length of a memo field, 50, to another length for output only. Thus,

```
SET MEMOWIDTH TO 65
```

DISPLAYs or LISTs the contents of a memo field with 65 characters per line. This command also works in a printed report. You can set MEMOWIDTH in the Config.db file. See *Using dBASE III PLUS* for more information.

Help for Fast Typists

If your users complain that they type too fast for dBASE III PLUS when they are entering information, you can change the number of characters that dBASE III PLUS stores in its type-ahead buffer. This buffer zone catches characters and retains them until dBASE III PLUS is ready to interpret them. The default is 20, but you can SET TYPEAHEAD TO any number from 0 to 32,000, provided you have enough memory in your computer.

NOTE
SET TYPEAHEAD will not work unless SET ESCAPE is ON, so it's best not to change the type-ahead buffer unless absolutely necessary. You can clear out the type-ahead buffer with CLEAR TYPEAHEAD. This command is particularly useful in a program where you don't want the user to input information before continuing with the program. See *Using dBASE III Plus* for more information.

Chapter 9
EVALUATING USER INPUT

Trapping users' mistakes, not only in screen forms but also after the user has typed in responses, is a very important part of programming. Make sure that the program knows what to do when the user types an incorrect response. The program should also ask the user to verify that the input is correct, and be able to handle situations in which the user presses the special keys on the keyboard.

What This Chapter Covers

This chapter discusses the following:

- How to restrict input to certain characters in prompts and messages
- How to check for users' mistakes
- How to filter data to get only the information that your program needs
- How to manage the special keys on the keyboard from within a program.
- How to use the ON command to test for certain conditions

Preparing for This Chapter

Have a general understanding of the basics of dBASE programming and of the information in Chapters 5 through 7.

Filtering the Input Line

In Chapter 7, you learned how to restrict input in on-screen forms by using templates and ranges. Whenever you can, try to limit the user's choices and then set up a filter so that the program only accepts a correct response. You do this most often in menus, or when you're prompting the user to hit particular keys, such as **y** for yes, or **n** for no. In the checkbook management system, for example, the main menu choices are limited to the letters **A** through **L** and **X**.

Using Numeric Choices

You can also set up the choices as numbers and then limit these choices to a particular numeric range. Say you have a memory variable of numeric type named *choice*, which is for user input. If you set up the choices in the range 1 through 8, you can easily restrict the user's input like this:

```
choice = 1
@ 10,10 GET choice PICTURE '9' RANGE 1,8
READ
```

The program doesn't accept a number less than one or greater than eight. This technique also prevents the program from accepting a letter key.

Because the program can't continue until the user enters a valid choice, a setup like this one catches the user's mistake before it can disrupt the rest of the program.

Anticipating the Correct Response

After receiving a response, your program should always ask the user to verify that the input is correct. The checkbook management system uses this conventional prompt:

Is this correct? (Y/N)

The program employs a DO WHILE loop to allow the user to correct mistakes on an **n**, or no, answer. Only after the user types **y** for yes does the program continue. It's easy to check for numeric ranges, but how do you set up the program to check for string input such as **y** or **n**?

First, think about the possible choices here. They may be only **y** or **n**, but the user may type **Y** or **N**. The best way to get around this situation is to use a template in the GET line to convert the user's input to upper case:

```
STORE ' ' TO answer
@ 5,0 GET answer PICTURE '!'
READ
```

If the user types **y** or **n**, the program will convert this response into **Y** or **N**. That still doesn't ensure that the user types only **y** or **n**. For that you need an instruction that loops back and requests a correct response as long as the user doesn't type **y** or **n**. You are excluding incorrect answers and including correct ones.

dBASE III PLUS uses the $ operator for what it calls a substring search. This feature looks for the input in a string which contains all the possible choices and makes sure that the user's response is one of these choices. Here is an example of how to code the program to evaluate the above yes/no situation:

```
DO WHILE .NOT. answer $ 'YN'
```

Read this command line as: Do the loop as long as the contents of the variable *answer* are not found in the string **YN**. The string is enclosed in delimiters. As long as the user doesn't type **y** or **n**, which are converted to upper case at the time of input, the program loops until the user types a response that is in the string. Here is a pattern for filtering string input so that it's either **y** for yes or **n** for no.

```
row = 5
column = 10
choice = ' '
DO WHILE .NOT. choice $ 'YN'
   choice = ' '
   @ row,column GET choice PICTURE '!'
   READ
ENDDO
```

The line below DO WHILE is important: it tells the program to reinitialize the memory variable *choice* with a blank if the user doesn't press either **y** or **n**. The program returns to the beginning of the loop if the user presses another letter, for instance, **t**. The **t** would be the current contents of the memory variable *choice* after the first try. It would then appear in the @...GET blank and perhaps confuse the user. So, the module reinitializes *choice* with a blank at the beginning of every loop. This module of code efficiently handles the input by converting it to upper case and making sure that it is either **Y** or **N** before the program continues.

Another example of the substring search technique is in the CASE lines in the main program module of the checkbook management system, Cbmenu.prg. Here the program uses the $ operator in a slightly different fashion. For example:

```
CASE CHR(i) $ "Aa"
   DO Check
```

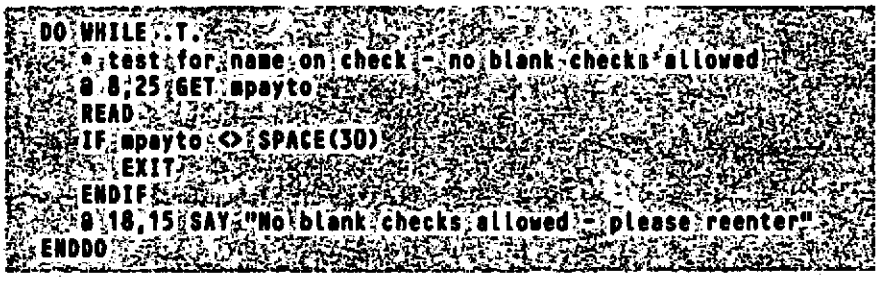
This CASE line uses the substring search operator to look for the ASCII character equivalent of the input variable, *i*, to be either **A** or **a**.

> **NOTE**
> Don't confuse the substring search operator, $, with the SUBSTR() function. The $ operator merely looks for a substring within another string, while the SUBSTR() function returns a portion of a character string.

Covering All Possibilities in Their Turn

Two of the most difficult aspects of programming are testing for all possible input conditions and determining the order in which to have your program test these conditions. As your programming experience grows, you'll develop ways to handle user input. Here are some suggestions.

Most evaluations center around a logical condition. This means they can be set up in one of the standard dBASE constructions, DO WHILE...ENDDO, IF...ENDIF, and DO CASE...ENDCASE. For example, in the Check.prg, which enters checks in the checkbook management system, the following section of code tests for the user's typing of the check's payee by means of a variable called *mpayto* and a simple IF...ENDIF construction:



```
DO WHILE .T.
  * test for name on check - no blank checks allowed
  @ 8,25 GET mpayto
  READ
  IF mpayto <> SPACE(30)
    EXIT
  ENDIF
  @ 18,15 SAY "No blank checks allowed - please reenter"
ENDDO
```

The program uses a negative approach. It evaluates the contents of the variable *mpayto* to see if it is not equal to 30 blanks, SPACE(30). If the line

```
IF mpayto <> SPACE(30)
```

evaluates as true, no other action is needed and the program EXITs the DO WHILE loop. If the line evaluates as false, that is, *mpayto* variable is equal to SPACE(30), the program gives an error message and returns to the beginning of the loop. In the above example, the program could have used a positive approach:

```
IF mpayto = SPACE(30)
```

How you have your programs evaluate the user's input is up to you. What is important, though, is that you try to anticipate all possible conditions relating to the input. Think carefully about the order in which your program deals with the possibilities. The next example, from the same program module, Check.prg, evaluates the user's input of the check amount, the variable *mamt*:


```
DO WHILE .T.
   * test for amount less than or equal to 0
   @ 8,66 GET mamt PICTURE "999,999.99"
   READ
   @ 18,0
   IF mamt = 0.00
      @ 18,15 SAY "Check must have an amount - please reenter"
   ELSE
      IF mamt < 0.00
         @ 18,15 SAY "Check must be a positive amount - "+;
                     "please reenter"
      ELSE
         EXIT
      ENDIF
   ENDIF
ENDDO
* check to make sure there are sufficient funds to cover check
IF balance < mamt
   @ 18,10 SAY "There are not sufficient amount of funds "+;
               "to cover this check."

* option to enter check anyway
   @ 20,15 SAY "Do you wish to enter this check anyway? (Y/N)"
   ans = " "
   DO WHILE .NOT. ans$"YyNn"
      ans = " "
      @ 20,61 GET ans
      READ
   ENDDO
* (...)
ENDIF
```

The program first tests to see if the amount is equal to zero and, if that is the case, presents the user with a message. It then checks whether the amount is a negative number, which would mean that the user has made an obvious mistake. The program handles these two possibilities in the same DO WHILE loop, because they are related input mistakes.

The third possibility is more serious. If the user enters a correct number, but there is not enough money in the account to cover the check —

```
IF balance < amt
```

— the program will generate a negative balance. The program warns the user of this and lets the user decide what to do. In Chapter 14, you'll look at another program module to get more ideas about how to anticipate all possible conditions.

Special Keys

You may want your program to branch to certain subprograms depending on what key the user presses. As well as providing ways to filter and evaluate input of numeric and string information, dBASE III PLUS gives you methods to evaluate whether the user has pressed a standard alphanumeric key or one of the non-printing keys, such as the ↑ or the **Backspace** key.

The INKEY() Function

The INKEY() function tests for input of most keys on the keyboard, including the cursor movement and other special keys. INKEY() returns an integer value which corresponds to the ASCII code number of the last typed key.

```
i = 0
DO WHILE i = 0
  i = INKEY()
ENDDO
? i
```

The above program loops until a key is pressed. If the user presses **A**, the program exits the loop and displays 65, the ASCII code for A.

All the non-printing keys on the keyboard return INKEY() values between 1 and 31.

WARNING
Don't be confused by the input values of ASCII codes for the non-printing keys and their display value between 1 and 31. For example, if the user has pressed the → key, the INKEY() response would be the integer 4. But if you do this:

```
? CHR(4)
```

you'll get a little diamond shape on the screen. The ASCII codes refer to different keys when you are checking for input than they do for displaying screen output. INKEY() only checks keyboard input. See *Using dBASE III PLUS* for a list of INKEY() values.

Getting Fancy

In Chapter 4, you were promised an explanation of how the checkbook management main program uses the INKEY() function in the main program, Cbmenu.prg. Take a look at this module now:

```
i=0
DO WHILE i=0
   i=INKEY()
   @ 17,63 SAY TIME()
   @ 22,58 SAY ""
   IF UPPER(CHR(i))$"ABCDEFGHIJKLX"
      EXIT
   ENDIF
   i=0
ENDDO
```

This module sets up a continuous loop that waits for the user to type a key. The variable name *i* stands for keyboard input. The loop waits for the user to press a key —

```
i=INKEY()
```

— and continues to initialize *i* to 0 and count the time as long as the user doesn't press a key. That's why you see the seconds count off on the menu. Although this command displays nothing:

```
@ 22,58 SAY ""
```

the command positions the cursor at the bottom of the screen between the two colons and makes it appear as if it is prompting the user for an entry. It replaces the @...GET and READ construction, because in @...GET and READ lines, nothing happens until the user enters something. In the INKEY() example, however, the time is ticking away, and the use of INKEY() in this loop is an active situation.

When the user types a letter choice from the string "ABCDEFGHIJKLX," it activates the INKEY() function, which instructs the program to EXIT the loop. Otherwise, the program continues the loop and ticks off the seconds.

Initially, the variable *i* is a counter, and its value is numeric. However, the INKEY() function allows any numeric or alphabetic keyboard input into *i*. The program reinitializes *i* to be 0 for each repetition of the loop if the user either doesn't press a key or presses an incorrect key.

When the user presses an acceptable key, the program displays the input in upper case:

```
@ 22,58 SAY UPPER(CHR(i))
```

and the program then evaluates which CASE command to execute for the menu choice.

The READKEY() Function

INKEY() works in menus and other situations such as prompts, where the program waits for the user to enter a response by pressing a key. In full-screen commands, such as in APPEND or @...GET lines, you can determine the user's input of a key with the READKEY() function. This function works in a similar fashion to INKEY(), but only in full-screen situations, and it uses a different table of key values.

For example, if you want to know whether the user has changed any information in an on-screen form, you can use READKEY() to test if the user has pressed **Ctrl-End**, the integer value for which is 270. If the user has not changed any field, READKEY() returns a 0 value, and the program need not alter the database information. A list of the READKEY() values is in *Using dBASE III PLUS* under the READKEY() function.

The ↵ Key

Under certain circumstances, you may want the user to press ↵ alone as a valid choice. For instance, if the user presses ↵ instead of a letter choice, the program returns to the main menu. Because ↵ has a null value, its length is 0.

How do you test for this? If you want to have the program check if the user has pressed ↵, you can set up a module like the example below. Here, *select* is a character variable which the program initializes with a space. The user has just pressed a key:

```
IF LEN(TRIM(select)) = 0
   DO Something
ELSE
   DO Another
ENDIF
```

Another way to check for the same condition is:

```
IF "" = TRIM(select)
   DO Something
ELSE
   DO Another
ENDIF
```

The " " means that the input string contains nothing, not even a space. This is the way dBASE III PLUS indicates a null string. There is no easy way to set up this test with the substring search operator, $, so always keep this point in mind when you're designing your program. For instance, test for the ↵ key first and then for the other choices.

↵ = "" 0 characters long

Spacebar = " " 1 character long

Figure 9-1 The ↵ key has a null value.

Note the difference between the value of the ↵ key and the value of the **Spacebar**.

The Spacebar

You will probably use the LTRIM(), RTRIM(), and TRIM() functions to trim the leading and trailing blanks from the user's input. However, it may be important that the user not type any spaces in the middle of an input string. You can test for this potential problem with the AT() function. Here, the variable *select* holds the user's input:

```
DO WHILE .T.
   ACCEPT "What is the account number? " TO select
   STORE LTRIM(TRIM(select)) TO select
   * if a space is in the string
   IF AT(" ",select) > 0
      CLEAR
      ? CHR(7)
      @ 10,10 SAY "Don't type spaces in the account number!"
      ?
      WAIT "          Press any key to try again."
      RELEASE select
      LOOP
   ELSE
      EXIT
   ENDIF
ENDDO WHILE .T.
```

After the user types the input, *select* is TRIMmed of leading and trailing blanks. The AT() function then checks for any blanks within the string *select*. Because AT() returns the numeric value of the substring's position in the string, or 0 if the substring is not in the string, you must test if AT() returns a value greater than 0.

Checking for Type

You can test whether certain user input is of a specific type with the three functions ISALPHA(), ISLOWER(), and ISUPPER(). These functions return a logical true or false. For example, if you want to check that the user has input a letter of the alphabet, you can say:

```
@ 20,10 GET select
READ
IF ISALPHA(select)
   DO Something
ELSE
   DO Error
ENDIF
```

Similarly, the functions ISLOWER() and ISUPPER() test for lower case and upper case. These functions test the first character in the string only.

The TYPE() function, unlike the above three functions, returns the specific data type of a memory variable, that is, character, numeric, logical, date, or undefined. When using TYPE(), enclose the name of the variable in delimiters:

```
. STORE 123 TO memvar
. ? TYPE('memvar')
N
. ? TYPE('memvar2')
U
```

Here, the TYPE() function returns N for the numeric variable *memvar*, but U for the undefined variable *memvar2*. You can also use TYPE() to determine a field data type, which can be character, numeric, logical, date, or memo.

Using the ON Command

The ON command allows your program to branch to subprograms depending on whether the user presses the **Esc** key or any other key, or if a dBASE III PLUS error has occurred. There are three versions of ON:

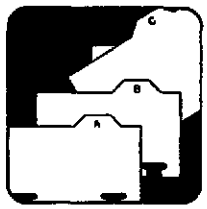
ON ESCAPE — tests for **Esc** key
ON KEY — tests for any other key
ON ERROR — tests for a dBASE error condition

The ON ESCAPE command allows you to set up a way to handle the user's pressing the **Esc** key while your program is running without ending the program prematurely:

```
ON ESCAPE DO Warning
```

These three commands, like the INKEY() and READKEY() functions, should be set up in continuous loops to evaluate keyboard input. If you put them at the beginning of the main program file, they'll be in effect throughout the program, or you can turn them off when they're no longer needed. To turn off ON ESCAPE, type ON ESCAPE without any other command on the line.

Chapter 10
WORKING WITH THE DATABASE



In addition to getting information from the user through screen forms and prompts, your program will make use of data in database files. It has to open, or USE, the database file, with its related index files, and locate the correct records. These are the topics of this chapter.

What This Chapter Covers

In this chapter you'll learn:

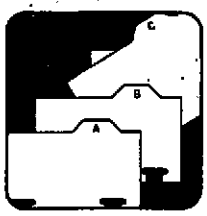
- How to design a database
- How to establish the database file in USE for a program
- How to set up work areas and ALIASes
- How to set up multiple-field INDEXes
- How to change the master index file
- How to locate data in the database file
- What the end-of-file and beginning-of-file conditions mean and how to work with them
- How to filter database information

Preparing for This Chapter

Understand the essentials of dBASE programming, how to work with field and memory variable types, and the basics of getting user input.

Designing the Database

The designing of a database significantly affects the design of a corresponding program. dBASE III PLUS really shines when you follow the relational pattern for building database files. You will probably design several different, yet related, database files as part of your database, along with corresponding index files.



TIP
One cardinal rule to follow when designing database file relationships is: don't overlap fields. The relational concept works best with one field as the bridge, the relation, between different files. When you have superfluous fields in several database files, you waste disk space and make the program run more slowly.

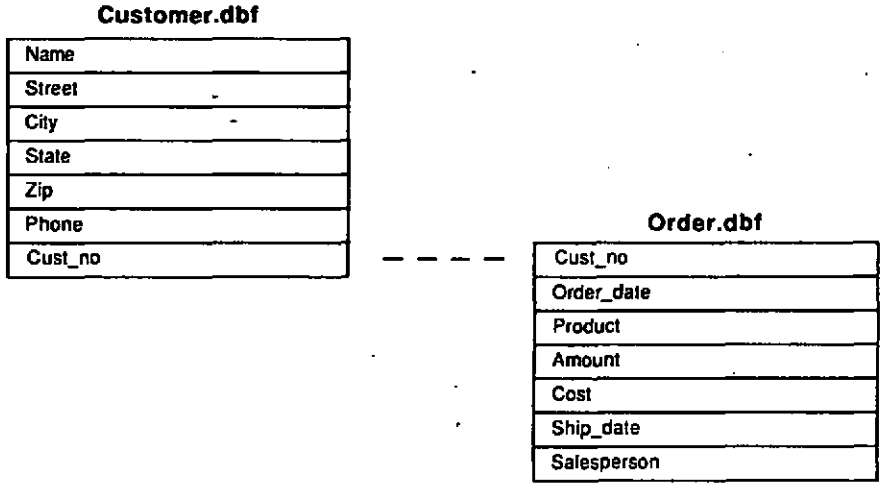
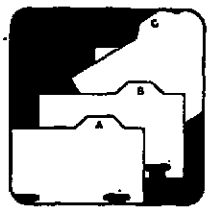


Figure 10-1 A relational database

In this example, the Cust_no field is the key that relates, links, two separate files.



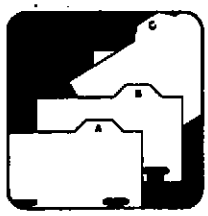
Sometimes the programming project uses an existing database, but you should consider restructuring the database if you feel that you can improve the speed and efficiency of your program. dBASE programs run very fast, but if the program uses a large database, you may find that performance is slower when the program has to access files on the disk.

To lessen disk access times for a large database, break the database into smaller relational database files. For example, instead of maintaining all personnel data in one database file, keep less frequently used data in a file that is related to the main personnel database file on a key field, such as employee number. The program doesn't have to sift through this information all the time and runs faster because the main database file is smaller. The relational model lets you isolate sensitive data in restricted-access database files. For example, you may require passwords to identify authorized personnel. This is especially important if you are using dBASE III PLUS on a local area network. See *Networking dBASE III PLUS* for more information.

Opening the Database File

Depending on the size and complexity of your program, you can open, or USE, a database file and any related index files at the beginning of the program, or wait until the program actually needs the database file before putting it in USE. The checkbook management system follows the latter course. There are five database files USEd by the program: Checks.dbf, which is INDEXed ON Chkno TO Chkno.ndx; Deposits.dbf; Recon.dbf; Bank.dbf; and Tax.dbf. The program doesn't USE a database file until absolutely necessary.

Keeping a file closed until absolutely necessary maintains the integrity of the database file. If databases are left open, they can be corrupted by power failures or by the user accidentally turning off or resetting the computer. Whenever possible, have the program close files with the USE command as soon as it's finished with them.



If your program USEs two or more database files that are related to each other on a field, you must open files in different work areas with the SELECT command. For example, this module sets up two work areas:

```
SELECT 1
USE Accounts INDEX Accts
SELECT 2
USE Balance INDEX Bal
```

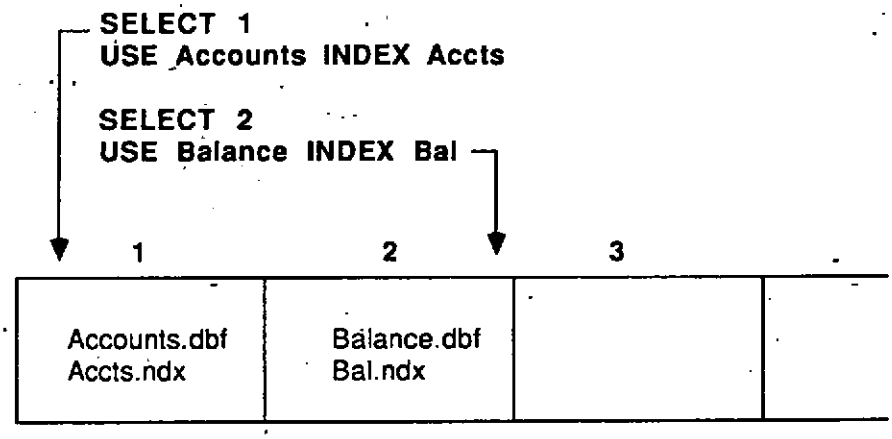
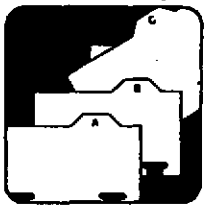


Figure 10-2 Work areas

However, note that since you have USEd the files, they are open until you close them. Issue the USE command to close the currently SELECTed work area only, or the CLOSE DATABASES command to close all work areas simultaneously.

In the next chapter, you'll learn how to manipulate field information between work areas with the SET RELATION command.



Using ALIAS Names

Another method for selecting database files is to establish ALIASes for the database filenames. Suppose you have a database file that you use continually throughout the program with a related index file. You could establish an ALIAS as in the following example:

```
SELECT 1
USE Accounts INDEX Accts ALIAS Acctsdue
```

Whenever you need this database file and its index in your program, you can insert the command:

```
SELECT Acctsdue
```

Depending on the name you use, an alias can make it clearer what database file is in what work area. Once you establish an alias, you must use it to refer to the database file until you close it.

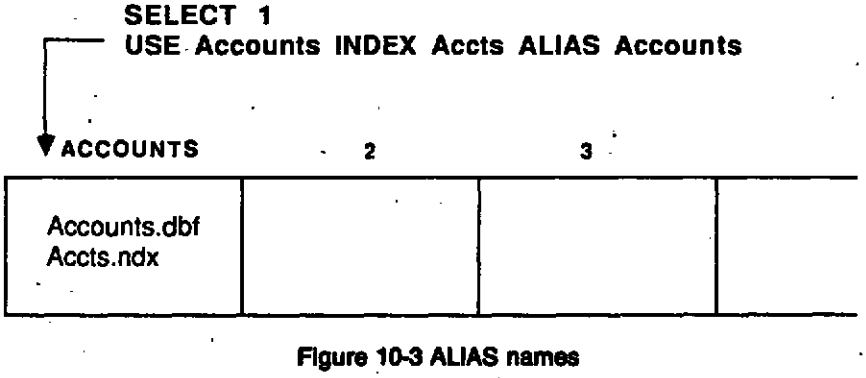
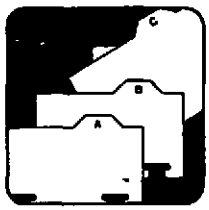


Figure 10-3 ALIAS names

WARNING
Don't use the single letters A through J for ALIAS names, because these letters also represent work areas 1 through 10, respectively.



Elaborate Indexes

You can have multiple-field indexes if you want records to be arranged according to several fields. One popular example is a name-and-address database file, where the first and last names and middle initial are in separate fields. The database file is INDEXed ON the last name, first name, and middle initial in that order. If several fields in the database file contain the same last name, dBASE III PLUS arranges the records according to the first name. If there are duplicate last and first names, dBASE III PLUS uses the middle initial to arrange the records.

To set up a multiple-field index file, set up an expression that shows the fields. Use the + operator to concatenate the fields. List the main field first and then the other fields in the order that you want dBASE III PLUS to INDEX them:

```
USE Names
INDEX ON Last + First + Middle TO Namein
```

NOTE
dBASE III PLUS INDEXes in ascending order.

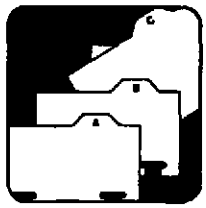
Here is another example. The database file Accounts.dbf is INDEXed ON Order_date, a date field, and Client, a character field. However, to get correct chronological order in the dates, you must use separate character string conversions of the year, month, and day portions of the dates:

```
USE Accounts
INDEX ON STR(YEAR(Order_date),4) + STR(MONTH(Order_date),2) ;
   + STR(DAY(Order_date),2) + Client TO Clientin
```

Using Several Index Files

You can have several index files set up for one main database file and call them all up when you USE the database file. In this example, the program opens three existing index files when it USEs Accounts.dbf:

```
USE Accounts INDEX Amountin, Clientin, Payin
```



The first index file is the master index. The master index controls the order in which dBASE III PLUS displays and accesses the database file. However, all index files in USE will be updated by the program whenever you update information in the main database file. You can have seven open index files related to a single database file.

Sometimes you'll want to change the order of the index files to make another index the controlling one. For example, if you're finished working with the Amount_due field and your program has to update the Client field, you may want to display the client information on the screen. Because this information is INDEXed by client name in the Clientin.ndx file, make that file the master index.

You can change the order of the index files currently in USE with the SET ORDER command, which takes a numeric argument.

```
SELECT Accounts
SET ORDER TO 2
```

The above module makes the second index, Clientin.ndx, the controlling one. You can use SET ORDER TO 0 to make the file appear in its unINDEXed order while index files are still open.

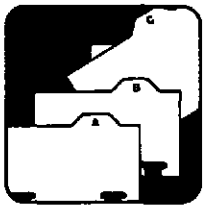
If you want to open another index file that is not currently in USE, use the SET INDEX command:

```
SELECT Accounts
SET INDEX TO Salesin
```

Unless you specify the index files already in USE when you SET another index, you'll have to repeat the entire list, because SET INDEX establishes the entire index list:

```
SELECT Accounts
SET INDEX TO Salesin, Clientin, Amountin, Payin
```

To close all index files but leave the database file in USE, type CLOSE INDEX or SET INDEX TO with no filename after it.



> ***TIP***
> Set up all the index files that you need once and then change the order with the SET ORDER command whenever necessary. Look in *Using dBASE III PLUS* for more about multiple index files.

Disk File Management

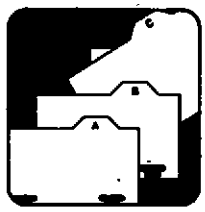
The program should take complete control of the database and related files it USEs. Whenever possible, set up these files when you design the program. However, occasionally the program won't know the name of a file in advance. It requests a filename from the user and checks whether such a file exists on the disk. dBASE III PLUS has functions that help the program to control disk space and directories. You'll learn about them in Chapter 13.

Finding Records

Most dBASE programs use either GOTO or one of the search commands, LOCATE, FIND, or SEEK, to isolate a record needed by the program. Which method you use depends on the way you've designed your program.

The program may ask the user for a specific record number and then instruct dBASE III PLUS to GOTO this number. More often, however, the user won't know the record number and your program must find the record. Once the program isolates the correct record, it can display the record on the screen using either @...SAY commands, if you've developed your own screen forms, or ?, DISPLAY, or LIST.

To find a record, your program ascertains what data the user needs from the database file. Then, it isolates the data. For example, if the user wants to look at all records in which the last name field contains the name **Smith**, the program isolates these records, if they exist.



There are three commands to search for information: LOCATE, FIND, and SEEK. A fourth command, CONTINUE, works with LOCATE. These three commands and the GOTO command move the record pointer through the file and stop at the very first record that fits the stipulated condition. However, there are ways to repeat the procedure for any other records that meet the same condition. The record pointer stays at this record until the program isolates another record.

When working with large database files, be sure to use a related index file, because then dBASE III PLUS can search through the database file very quickly. The FIND and SEEK commands only work with INDEXed files.

LOCATE and CONTINUE

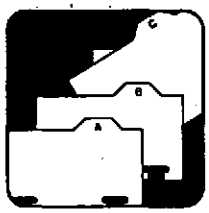
LOCATE works with any database file, INDEXed or not, but it is the slowest of the three search commands. The LOCATE command also must include a condition. For example, the Clrcash.prg looks for a withdrawal number equal to a control number:

```
LOCATE FOR Numwith=cntr
```

LOCATE is most useful when you are looking for a record that fits a specific condition. In the example below, from Clrdep.prg, the program searches for outstanding deposits in the Bank.dbf file. It isolates those records that contain an amount over 0 in the Num field and .F. in the Clear field:

```
LOCATE FOR Num > 0 .AND. .NOT. Clear
```

Once the program has LOCATEd a record, you can use the CONTINUE command to locate the next record, if any, that fits the same condition. CONTINUE only works with LOCATE. It starts at the previous record number found with the LOCATE command and continues the search.



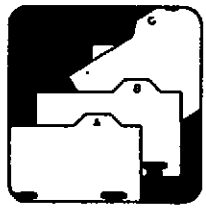
NOTE
You can set up a separate LOCATE command for each work area, but be careful that you know which is the currently SELECTed work area when you use CONTINUE.

FIND and SEEK

FIND and SEEK work in a similar manner but much more quickly than LOCATE. However, you should exercise care when using them, because they only function properly under the following conditions:

- The database file must be INDEXed on the field that you are FINDing or SEEKing.
- No matter where the record pointer is, FIND and SEEK always begin at the TOP of the database file and locate the first applicable record only.
- There is no continuation command, like CONTINUE, that works with FIND and SEEK.
- FIND and SEEK will match any string to be found, starting at the first character and continuing only for the length of the string. dBASE III PLUS stops at the first record that contains the string, which may not be the string you want. For example, you want to find the name Smith, but FIND positions the record pointer to a record containing the name Smithe. You can remedy this situation with SET EXACT, discussed below.

Think of FIND as a subset of SEEK, a more powerful and versatile command. FIND can only locate a character string which doesn't have to be delimited with quotes, or numbers. SEEK can evaluate an expression and locate its result. The values of this expression can be character, date, or numeric. If you're using SEEK to locate a string, remember to delimit the string. Here are some examples:



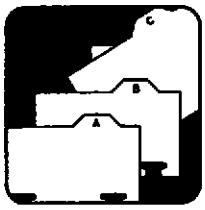
```
* to find a character string:
FIND Smith
* is the same as:
SEEK "Smith"

* to find an amount:
SEEK 124.50

* to find a date:
SEEK CTOD('09/12/86')
```

You can use both FIND and SEEK with memory variables directly by STOREing the user's input into a memory variable and then using this variable in FIND and SEEK commands, but you must use macro substitution with FIND.

```
INPUT "What is the customer's code number? " TO ccode
FIND &ccode
SEEK ccode
```

Here is an example of a program module using FIND:

```
SET TALK OFF
CLEAR
* Request last name from user
m_last = SPACE(20)
@ 10,10 Say "Enter the last name:"
@ 10,31 GET m_last PICTURE "@A"
READ
* Remove leading and trailing blanks
STORE LTRIM(TRIM(m_last)) TO m_last
* Open database file which is INDEXed on the last name field
USE Names INDEX Last
* Find it
FIND &m_last
* Display certain fields in the record
CLEAR
@ 1,10 SAY TRIM(First) + " " + TRIM(Last)
@ 3,10 SAY Street
@ 5,10 SAY TRIM(City) + ", " + State + " " + Zip
SET TALK ON
RELEASE m_last
RETURN
```



There is a major fault with this module: it doesn't consider the possibility that there is no last name that matches what the user types in. When there is no record found, the record pointer is at the end of the file. You'll see how to deal with this below.

FIND and SEEK Save Time

You can use FIND and SEEK together with a DO WHILE loop to find similar records. This method is like using LOCATE and CONTINUE. The trick is to position the record pointer at the first record with FIND or SEEK. This is a much faster method than LOCATE. Then use a loop to show all similar records. For example, if the user wants all records with the last name of Smith, you could change the previous example to:

```
SET TALK OFF
CLEAR
* Request last name from user
m_last = SPACE(20)
@ 10,10 Say "Enter the last name:"
@ 10,31 GET m_last PICTURE "@A"
READ
* Remove leading and trailing blanks
STORE LTRIM(TRIM(m_last)) TO m_last
* Open database file which is INDEXed on the Last_name field
USE Names INDEX last
* Find it
FIND &m_last
* Once it's found, show all other records with the same name
CLEAR
* Set up first row for screen display
r = 1
DO WHILE TRIM(Last) = m_last
   * Show information on three rows
   @ r,10 SAY TRIM(First) + " " + TRIM(Last)
   @ r+1,10 SAY Street
   @ r+2,10 SAY TRIM(City) + ", " + State + " " + Zip
   * Add blank row
   ?
   * Skip to next record that fits the condition
   SKIP
   * Increment row for next record
   r = r + 4
   * Loop back
```

```
ENDDO
SET TALK ON
RELEASE m_last
RETURN
```

This program uses relative addressing to display information on the screen. The scrolling doesn't stop when the entire screen display is filled up with database information. An error message will occur when it tries to DISPLAY on row 25. So, you must add a counter in the program to make sure that the display doesn't reach row 25. See the Listnames.prg module in the next chapter for a way to handle this situation.

The End-of-File Condition

Whenever you use GOTO or one of the search commands, dBASE III PLUS positions the record pointer at the specified record. Because the record pointer is at an existing record, the end-of- file condition is false, .F.

However, if the program tries to GOTO a record number that is higher than the last record in the file, or if LOCATE, FIND, or SEEK can't come up with a record that matches the condition, the record pointer reaches the end of the file. dBASE III PLUS then automatically sets the end-of-file condition to true, .T.

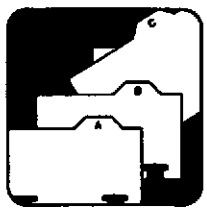
> **NOTE**
> The last record in the database file, that is, the BOTTOM of the database, is not the end-of-file. The end-of-file is directly after the last record. Similarly, the beginning-of-file, discussed below, is immediately before the first record. The terms TOP and BOTTOM do not refer to the beginning or the end-of-file conditions but rather to the first and last records, respectively, in the database file.

Your program must work with this end-of-file condition, and dBASE III PLUS provides you with a special function, EOF(). EOF() always returns a logical value, either true or false, depending on where the record pointer is.


```
. USE NAMES
. GO BOTTOM
. ? EOF()
.F.
. SKIP
Record No.      8
. ? EOF()
.T.
```

Whenever you use either SKIP, LOCATE, FIND, or SEEK to move the record pointer, you should make sure that your program tests for the end-of-file condition. Here is the original FIND example rewritten to include this test. Note the use of the DO WHILE...ENDDO construction.

```
SET TALK OFF
CLEAR
* Request last name from user
m_last = SPACE(20)
@ 10,10 SAY "Enter the last name:"
@ 10,31 GET m_last PICTURE "@A"
READ

* Remove leading and trailing blanks
STORE LTRIM(TRIM(m_last)) TO m_last
* Open database file which is INDEXed on the Last_name field
USE Names INDEX Last
* Find it
FIND &m_last
* End-of-file?
IF EOF()
   CLEAR
   @ 10,1 SAY "Sorry, there is no" + m_last + ;
   " in the database file."
   @ 13,0
   WAIT
ELSE
   CLEAR

   * Set up first row for screen display
   r = 1
   DO WHILE TRIM(Last) = m_last .AND. .NOT. EOF()
```



```
    * Show information on three rows
    @ r,10 SAY TRIM(First) + " " + TRIM(Last)
    @ r+1,10 SAY Street
    @ r+2,10 SAY TRIM(City) + ", " + State + " " + Zip
    * Add blank row
    ?
    * Skip to next record that fits the condition
    SKIP
    * Increment row for next record
    r = r + 4
    * Loop back
    ENDDO
ENDIF
USE
SET TALK ON
RELEASE ALL
RETURN
```

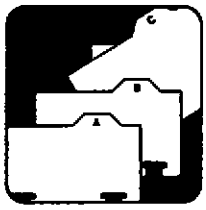
There are two points in the the module where the program must test for the end-of-file condition:

- When the program attempts to FIND the last name
- In the DO WHILE loop when the program SKIPs to the next record

Other File Functions

Similar to the EOF() function is the BOF() function, which returns the logical value .T. if the record pointer is at the beginning of the file. Use this function in the same way as you use the EOF() function. Use BOF() when the record pointer SKIPs backward and you want the program to test when or if the record pointer has reached the very beginning of the file.

```
. USE NAMES
. ? RECNO()
          1
. ? BOF()
.F.
. SKIP -1
Record No.       1
. ? BOF()
.T.
```



You can also use the RECNO() function to return the current record number, for instance, to STORE it in a memory variable so that you can GOTO it later:

```
FIND &m_last
IF .NOT. EOF()
   STORE RECNO() to record
ENDIF
```

LOCATE, FIND, and SEEK all work with the end-of-file condition. In the above examples, the program is looking for specific data and locates a record which contains the data. If no record exists, the end-of-file condition is set to true, and the program returns an appropriate message.

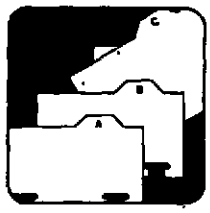
Sometimes, however, all you'll need to know is whether there is a match or not. For example, in the Check.prg, the following section of code checks for duplicate check numbers:

```
SEEK mchkno
IF FOUND()
   @ 18,15 SAY "Check number already exists. Please reenter."
ELSE
   EXIT
ENDIF
```

When the program SEEKs the check number, *mchkno*, it only needs to know whether another check with that number already exists. It uses the FOUND() function, which returns a logical value, true or false, depending on whether the record pointer stops at a record or not. In this example, if FOUND() is true, the program advises the user to enter another check number. It doesn't matter what record contains the check number, only that there is an exact match.

Filtering Commands

There are several commands that are useful filters for information in database fields. Programmers use these filters with such commands as LOCATE or FIND to bypass unnecessary data. These filtering commands are SET FILTER, SET DELETED, SET EXACT, and SET UNIQUE.



SET FILTER

If your program USEs a very large address database file, and all you want are listings for California, you can have dBASE III PLUS filter out all other states from the State field when looking for records:

```
SET FILTER TO "CA" $ State
```

SET FILTER makes the database file appear to dBASE III PLUS as if it contained only CA in the State field. The filter can also be a more complicated expression:

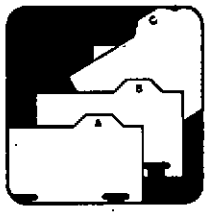
```
SET FILTER TO "CA" $ State .AND. Zip < "91400"
```

If you want to SET FILTER TO a date, you must convert a character string to a date:

```
* Filter only those records added after January 1, 1986
* Add_date is a date field in the database file
SET FILTER TO Add_date > CTOD('01/01/86')
```

The database file is not smaller when you use SET FILTER TO, so it still takes dBASE III PLUS the same amount of time to scan the entire file. Because it may affect the working of your program later, make sure that you remove the filter with SET FILTER TO without a condition after you're finished with this command.

TIP
It's a good idea to GOTO the TOP of the database file after you SET FILTER and before you start another operation, such as SORT or SUM. However, the INDEX command indexes all records in the file, whether or not they meet the filter condition.



Skipping Deleted Records

If you want to exclude records already slated for deletion in a FIND, LIST, SEEK, or DISPLAY operation, you can use the SET DELETED ON command. This command, normally OFF, filters out the deleted records. Another way to do the same thing would be:

```
SET FILTER TO .NOT. DELETED()
```

The SET DELETED ON command is also helpful when you are using the COPY TO command to copy only current and active records to a new database file, or the APPEND FROM command to copy in records from another database file. Note that with APPEND FROM, SET DELETED ON is not equivalent to SET FILTER TO .NOT. DELETED(). Filters only apply to the active database file while SET DELETED refers to the entire dBASE operating environment.

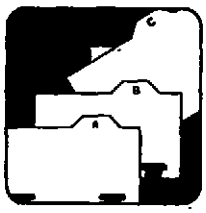
Checking for Exactness

Use SET EXACT ON to ensure that a comparison between two character strings is done exactly, character for character. Normally in comparisons, dBASE III PLUS starts on the left and does a character by character comparison until the character string on the right side of the operator runs out of characters. If all characters up to that point are the same, the strings are equal. Otherwise, they are not. So, the following commands:

```
STORE '123' TO number1
STORE '12345' TO number2
? number2 = number1
```

would be evaluated by dBASE III PLUS as being .T., true, although as far as you're concerned, it's false.

If you SET EXACT ON, however, dBASE III PLUS compares the strings exactly, and the answer to the above query is .F. Similarly, if you want your program to search for information that matches what the user types exactly, then you should SET EXACT ON immediately before the operation.

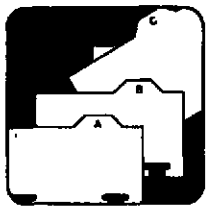


WARNING
dBASE III PLUS finds exact matches of character strings only with SET EXACT ON. Make sure that you SET EXACT OFF as soon as you are finished with it to prevent this command from interfering with the rest of the program.

Avoiding Duplicates

A database file may contain records with duplicate information in a field. If you prefer not to see duplicate information in a field, first SET UNIQUE ON, then create an index file using that field as the key field. With SET UNIQUE ON, dBASE III PLUS includes only the first record with the key information in the index file. Then when you use APPEND, BROWSE, CHANGE, DISPLAY, EDIT, LIST, or REPLACE, you will not see records with duplicate information in the key field because they are not part of the index file.

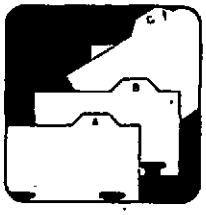
In addition, that index file will always be updated as a unique index file. However, make sure that you SET UNIQUE OFF before creating any index files that are to include all records.




```
. USE Names
. LIST
Record#  FIRST      LAST       STREET                 CITY           STATE  ZIP
      1  George     Jones      404 Main Street        Hightown       NY     10301
      2  Steven     Smith      33 Geary Street        San Francisco  CA     91405
      3  Sue        Smith      125 West Avenue        Rochester      NY     14605
      4 *Jean       Walker     Route 6                South Fork     ND     77632
      5  John       Smithe     303 14th Street, N.E.  Seattle       WA     98021
      6  Melanie    Thomas     4102 Los Feliz Blvd.   Los Angeles    CA     90027
      7  Melanie    Thomas     4102 Los Feliz Blvd.   Los Angeles    CA     90027
. SET FILTER TO State $ 'CA'
. LIST
Record#  FIRST      LAST       STREET                 CITY           STATE  ZIP
      2  Steven     Smith      33 Geary Street        San Francisco  CA     91405
      6  Melanie    Thomas     4102 Los Feliz Blvd.   Los Angeles    CA     90027
      7  Melanie    Thomas     4102 Los Feliz Blvd.   Los Angeles    CA     90027
. SET FILTER TO
. SET DELETED ON
. LIST
Record#  FIRST      LAST       STREET                 CITY           STATE  ZIP
      1  George     Jones      404 Main Street        Hightown       NY     10301
      2  Steven     Smith      33 Geary Street        San Francisco  CA     91405
      3  Sue        Smith      125 West Avenue        Rochester      NY     14605
      5  John       Smithe     303 14th Street, N.E.  Seattle       WA     98021
      6  Melanie    Thomas     4102 Los Feliz Blvd.   Los Angeles    CA     90027
      7  Melanie    Thomas     4102 Los Feliz Blvd.   Los Angeles    CA     90027
. SET DELETED OFF
. SET UNIQUE ON
. INDEX ON Last TO Lastname
      5 records indexed
. LIST
Record#  FIRST      LAST       STREET                 CITY           STATE  ZIP
      1  George     Jones      404 Main Street        Hightown       NY     10301
      2  Steven     Smith      33 Geary Street        San Francisco  CA     91405
      5  John       Smithe     303 14th Street, N.E.  Seattle       WA     98021
      6  Melanie    Thomas     4102 Los Feliz Blvd.   Los Angeles    CA     90027
      4 *Jean       Walker     Route 6                South Fork     ND     77632
. SET UNIQUE OFF
. FIND Smit
. DISPLAY
Record#  FIRST      LAST       STREET                 CITY           STATE  ZIP
      2  Steven     Smith      33 Geary Street        San Francisco  CA     91405
. SET EXACT ON
. FIND Smit
No find
```

Figure 10-4 The difference between SET FILTER, SET DELETED, SET EXACT, and SET UNIQUE

Chapter 11
WORKING WITH DATA IN THE DATABASE



After the program isolates the data it needs, it uses the data in a variety of ways. The program displays the data, puts it into memory variables, manipulates it with functions, and updates it. The program may also use data from two or more related database files.

What This Chapter Covers

In this chapter you'll learn:

- How to work with database information
- Ways to update database information from within a program
- How to work with information in two or more work areas
- How to set up relationships between work areas

Preparing for This Chapter

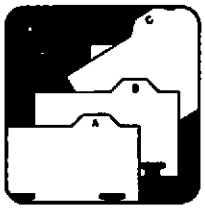
Make sure you understand the basics of dBASE programming and that you have read the previous chapter before working with this chapter.

Manipulating Data

You have learned that there are many commands and functions for manipulating the data in a database file. These commands and functions work either from the dot prompt or from The Assistant. They work exactly the same in dBASE programs.

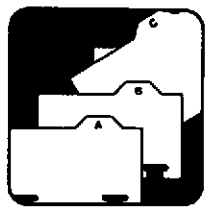
For instance, the Reconcil.prg module in the checkbook management system reconciles the bank statement to the records in the database files Checks.dbf and Bank.dbf. It also uses the SUM command and, if the balance is a negative amount, the ABS() function:

```
* total outstanding checks
USE Checks
SUM Amt TO outstand FOR !.NOT. Can

* total deposits and cash withdrawals in
* transit
USE Bank
SUM Amt TO notclear FOR Num > 0
SUM Amt TO notcash FOR Numwith > 0
notcash = ABS(notcash)
```



There are countless other ways to use database files. Your program may show field information with the ?, ??, DISPLAY, LIST, or @...SAY commands in customized screen forms. The following module displays the first name, middle initial, and last name fields in the regular columns of a screen form. It keeps a counter of the row number. When the display reaches row 21, it stops and requests the user to press a key to see more names. When the user presses any key, it begins a new screenful of names.

```
* Program...: Listname.prg
* Author...: Vincent Alfieri, Ph.D.
* Date.....: 06/26/85
* Notes....: This program lists the First, Middle and Last
*            fields from Names.dbf. Change file and field
*            names for use with other database files.
*
SET TALK OFF
CLEAR
* Open database and index files
USE Names INDEX Names
* Top of screen display
@ 1,11 SAY 'First Name'
@ 1,30 SAY 'Middle Initial'
@ 1,52 SAY 'Last Name'
* Display broken line across screen
@ 2,1 SAY REPLICATE('-',73)
* Set up initial screen row
r = 4
* Loop for all records in database file
DO WHILE .NOT. EOF()
   @ r,12 SAY First
   @ r,37 SAY Middle
   @ r,53 SAY Last
   * Skip to next record
   SKIP
   * Check for last row on screen
   IF r = 21 .AND. .NOT. EOF()
      ?
```

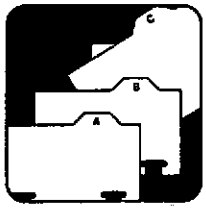


```
    WAIT REPLICATE("*",20) + " Press any;
      key to view more names " + REPLICATE("*",20)
    * Leave top of screen display for new
    * screen
    @ 4,0 CLEAR
    * Return to row 4 for new screen
    r = 4
  ENDIF
  * Increment row number for next iteration of loop
  r = r + 1
ENDDO WHILE .NOT. EOF()
USE
RELEASE r
SET TALK ON
RETURN
```

TIP
Learn how to use the various scope options in such commands as DISPLAY and LIST. You can specify four scopes:

- Just one record: DISPLAY RECORD 2
- A number of records beginning with the current record: DISPLAY NEXT 10
- All the records in a database file: DISPLAY ALL
- The rest of the file starting at the current record: DISPLAY REST

Many more examples of using database information are in this and the following chapters, as well as in the checkbook management files. Here are some particularly important operations for working with database information from within dBASE programs.



Changing the Database Information

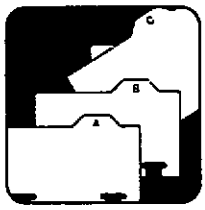
This section deals with how to update database files from within other programs. In Chapter 13, you'll see how to modify a database file structure.

Updating Data

After your program has allowed the user to change or add data in an on-screen form and has verified the correctness of the data, it must eventually add the new or updated data to the database file. If your program doesn't use the standard full-screen commands, APPEND, BROWSE, CHANGE, and EDIT, it must locate the correct record and REPLACE the old data with the new data.

In Chapter 2 of this manual, you learned the basic pattern for changing database data:

1. Initialize memory variables to hold the user's input. These memory variables correspond to the actual fields in the database file. Many programs give the variables similar names, such as *mfirst* to correspond to the First field. Make sure that the type and length of each variable are the same as the type and length of the corresponding field in the database file.
2. USE the database file, with INDEXes where appropriate, and GOTO, LOCATE, FIND, or SEEK the correct record.
3. If this is not a new record, STORE the contents of the fields in the corresponding memory variables.
4. Display a screen form to request the input from the user and store the input into the corresponding memory variables. This screen form shows the current contents of the fields. The user can then make the desired changes. Use templates to filter incorrect information. For a new record, the user is presented with blank fill-in forms, as in APPEND.
5. When the user types in the information, prompt the user to verify that all information is correct. If not, give the user a chance to correct it.
6. REPLACE the information currently in the fields with the new information in the memory variables.



7. If there are no more changes or additions to be made to the database file, close the file and clear the memory variables to use them again.

When you use the REPLACE command, there are two possibilities to consider:

- The record already exists.
- A new record must be created.

In the first case, the program REPLACEs the current contents of the record with the new data. Make sure, however, that the record pointer is at the correct record number. Remember that the pointer doesn't move unless you move it, so it should be at the last record used by the program.

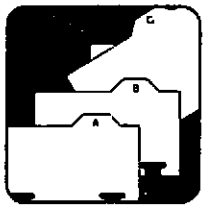
For example, the Clrdep.prg clears an outstanding deposit once that deposit has cleared at the bank. The program first finds the correct record number in the Bank.dbf database file:

```
LOCATE FOR Num=number
```

It checks to see if the number exists, that is, if the record pointer is not at the end-of-file, and determines whether the deposit has already been cleared. It then verifies from the user that the deposit is the right one to be cleared. When the user says yes, the program updates the database file:

```
IF UPPER(answer)= "Y"
   * if correct - replace fields in database
   *    file and erase deposit display from screen
   REPLACE Clear WITH .T.
   REPLACE Num WITH 0
* (...)
ENDIF
```

Because the record pointer hasn't moved since the program LOCATEd the record, dBASE III PLUS REPLACEs the contents of the current record with the new information.

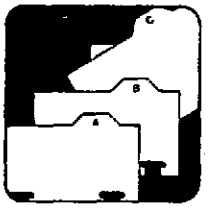


TIP
Your program can't know which fields may have been changed or updated by the user, so it's a good idea to REPLACE them all with the new information in the memory variables. For example, if a database file contains ten fields and the user is presented with all ten in an on-screen form, make sure that the program REPLACEs all ten fields with the ten memory variables.

The second possibility, that of a new record, is easily handled in a program without using the normal APPEND command. The program uses APPEND BLANK instead and REPLACEs the empty fields in the BLANK with the new information. In the Check.prg module, for example, after the user has verified that the new check information is correct, the program updates the database file in this fashion:

```
IF UPPER(answer)= "Y"
   * data is correct - add to database
   APPEND BLANK

   REPLACE Chkno WITH mchkno
   REPLACE Date WITH mdate
   REPLACE Payto WITH mpayto
   REPLACE Amt WITH mamt
   REPLACE Memo WITH mmemo
   REPLACE Can WITH mcan

   * update balance and lastchk
   balance = balance - Amt
   lastchk = Chkno
   * (...)
ENDIF
```

Because this information also changes with each new check, the program updates the current balance and the last check number at the same time.



TIP
In both cases, the program uses the CLOSE DATABASES command at the end of the module to close the database file properly. Don't allow your programs to leave database files open.

The Posting Method

Many accounting firms use the posting method to update data. They keep a temporary posting file with additions or changes, but they don't update the main database file every time they make an addition or change. Periodically, usually daily, weekly, or monthly, depending on the amount of business they do, they incorporate all the changes from the posting file to the main database file.

The UPDATE command performs this operation. If you are writing accounting programs, you may wish to use this command instead of or in addition to the REPLACE command. It's fully explained in *Using dBASE III PLUS*.

If your program has to check what date changes were last made to a database file, use the LUPDATE() function. This function returns a date value, so you can compare the date of the last update with any other date. See the example in the Commands and Functions sections of *Using dBASE III PLUS* for one practical use of LUPDATE().

Another method of updating information is to combine two database files with the JOIN command. Be aware of the difference between UPDATE and JOIN. UPDATE replaces information in the current database by copying in current information from a separate file. JOIN combines two files and creates a new, composite file. At the end of this chapter is an interesting program that simulates the JOIN command.

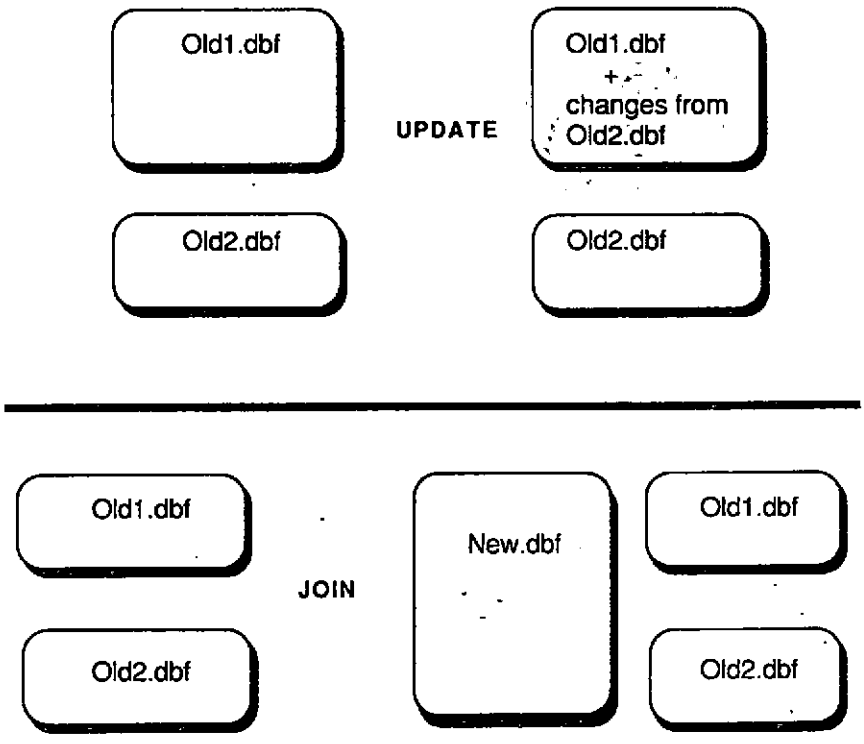


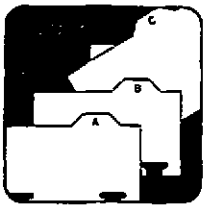
Figure 11-1 UPDATE and JOIN

Deleting Records

Deleting records is the same in dBASE programming as it is from the dot prompt or ASSIST. Position the record pointer to the correct record number, using GOTO or one of the search commands, and then DELETE the record. You can also use DELETE ALL with a condition, as the checkbook management program does in the Yearend.prg module:

```
DELETE ALL FOR YEAR(Date)=myear
```

You can, of course, RECALL a record marked for deletion as long as you haven't PACKed the database file. Don't forget to issue the PACK command to clean up the database file.

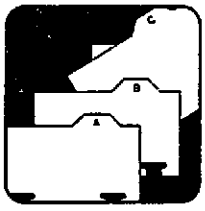


TIP
Some programmers choose not to PACK database files on a daily basis, but rather to maintain scratch files. They then consolidate files as part of their periodic housekeeping. This allows them to reinstate records if anything goes wrong. It's up to you how often you clean up your database files with PACK.

You can also use the quick and efficient ZAP command, but use it sparingly. This command DELETEs and PACKs all records at once. The checkbook program uses this command only in the Reinit.prg module, which clears the entire active database file to begin again for the next year:

```
* reinitialize all database files
USE Checks
ZAP
INDEX ON Chkno TO Chkno
USE Bank
ZAP
USE Deposits
ZAP

* (...)
CLOSE DATABASES
```

After the program ZAPs all the records from the Checks.dbf file, it must again INDEX the file in preparation for the next year's checks.



Copying Records

You can copy records to other database files, just as if you were working at the dot prompt. For example, in the Yearend.prg, before the program DELETEs ALL records from the active database files, it first COPYs the records to archive files with the year of transaction. Here are the commands that set up the three archive files:

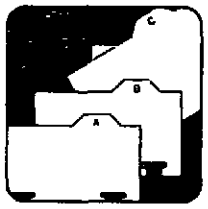
```
file1= "Dep"+STR(myear,4,0)
file2= "Bank"+STR(myear,4,0)
file3= "Chk"+STR(myear,4,0)
```

The three new files contain the last year's check records. They relate in name to the active files Deposits.dbf, Bank.dbf, and Checks.dbf, which are used by the program for each year's checkbook activities. At the end of the year, the program copies the past year's records from the active database files to archive files. Then it DELETEs the records from the active files so that it can use the three active files for the next year's activities:

```
USE Deposits
COPY TO &file1 FOR YEAR(Date)=myear
DELETE ALL FOR YEAR(Date)=myear

PACK
USE Bank
COPY TO &file2 FOR YEAR(Date)=myear
DELETE ALL FOR YEAR(Date)=myear
PACK

USE Checks INDEX Chkno
COPY TO &file3 FOR YEAR(Date)=myear
DELETE ALL FOR YEAR(Date)=myear
PACK

* (...)
CLOSE DATABASES
```

Different Work Areas

If you're working with two or more database files at once, you can work with fields from all the files in USE without switching between work areas. However, you must access them by filename or ALIAS name.



The Selected Work Area

It's important to know which work area is currently SELECTed. For example, say you have set up two areas like this:

```
SELECT A
USE First INDEX First
SELECT B
USE Second INDEX Second
```

You have opened the two database files, First.dbf and Second.dbf, with their respective index files, given them ALIASes, and placed them in two work areas. You then SELECT A, the First.dbf. This database file has two fields, First and Last. Second.dbf, currently unSELECTed, has a character field called Street. So you can do this:

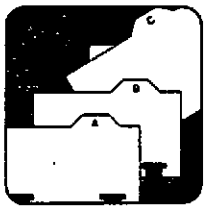
```
DISPLAY First, Last, Second->Street
```

This command DISPLAYs the First and Last fields in the current record in the First.dbf file and the Street field in the Second.dbf file. The pointer, ->, is dBASE III PLUS's way of referring to a field in a database file in another work area. The filename or ALIAS name precedes the pointer.

> **NOTE**
> If there is ever any confusion between field names and memory variable names, dBASE III PLUS always gives the field name. However, to reference a memory variable name, you can use the pointer M->. It's reserved to refer only to memory.

SET RELATION

The above examples show field information from the current record of the database files in two work areas, but they don't relate the two files together on a field. The Street field in the first record of Second.dbf may contain information that has nothing to do with the First and Last fields in the first record of First.dbf.



If two database files are related by a field, you can make the record pointer automatically go to related records in the second work area, as determined by the position of the record pointer in the first work area. The SET RELATION command establishes a link between two work areas.

You can only have one active RELATION for each work area. One good way to think of SET RELATION is as a way of temporarily linking two database files in USE by means of something they both have in common.

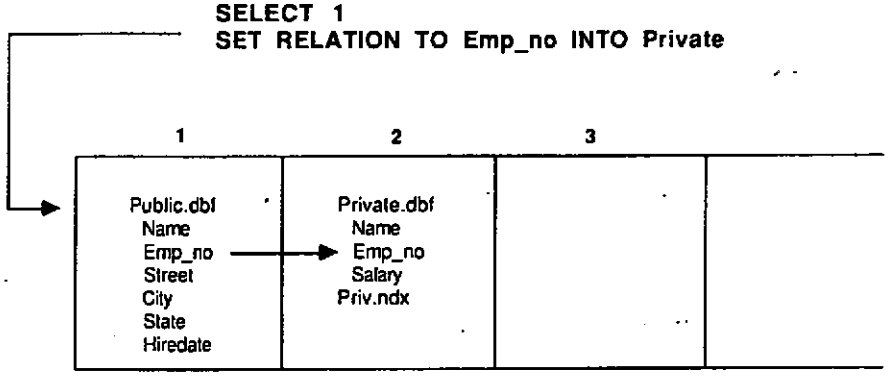
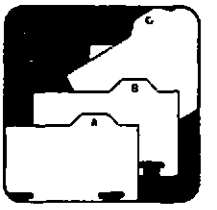


Figure 11-2 Establishing a relationship between work areas

NOTE
The SET RELATION command is most effective when both files have corresponding records. If there is no record in the linked file that matches the key field, dBASE III PLUS positions the record pointer at the end of the file.



The most common use of SET RELATION is with the key expression option. When the record pointer in the first work area moves, the record pointer in the second work area moves to a record whose INDEX key matches the value of the relation key expression. The second work area file must be INDEXed.

For example, you are working with two files, Public.dbf and Private.dbf, containing personnel information and related to each other by the field called Emp_no, an employee number. The Private.dbf file is INDEXed on Emp_no to Priv.ndx. You have set up the work areas like this:

```
SELECT 1
USE Public
SELECT 2
USE Private INDEX Priv
SELECT 1
```

You can then SET the RELATION from the currently SELECTed work area to the other work area on the employee number:

```
SET RELATION TO Emp_no INTO Private
```

Whenever the record pointer moves in the first work area, the record pointer in the second work area will be positioned to the related record:

```
DO WHILE .NOT. EOF()
   DISPLAY First, Last, Private->Salary
   SKIP
ENDDO
```

If you have more than one occurrence in the related file, dBASE III PLUS will only point you to the first occurrence.

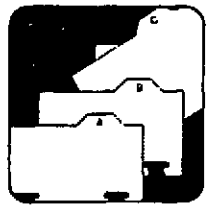
SET RELATION establishes a temporary relationship between two files. Using SET RELATION is not the same as INDEXing files, which creates and saves an index file for the database file. The relation disappears when you close the work areas or when you issue the command SET RELATION TO.

> **NOTE**
> When using a function, place the entire field reference, including the ALIAS name, in the parentheses:
>
> ```
> @ 10,10 SAY SUBSTR(Private->Address,1,25)
> ```

Advanced Relations

Below is an example of a more complicated relationship. This module program simulates the JOIN command and makes use of the SET RELATION feature. Notice that it uses the standard REPLACE technique that you investigated earlier in this chapter.

```
* Program.: S-JOIN.PRG
* Author..: Luis A. Castro, modified by David McLoughlin
* and Vincent Alfieri, Ph.D.
* Dates...: 01/11/83, 01/17/85, 06/24/84,11/16/85, 06/26/85
* Notes...: This program simulates the JOIN command. The
*           filenames may be requested from the user and
*           entered from the keyboard with ACCEPT statements
*
SET TALK OFF
*
firstfile = "OLDNAMES"
secondfile = "NEWNAMES"
joinfile = "TOGETHER"
key_expr = "Lastname+Firstname"
*
* Initialize macro to replace the joinfile from the
* secondfile. Only the Amount fields are to be replaced

mreplace = "Amount WITH B->Amount"
*
* Joinfile has all the fields to be JOINed and was created
* prior to the beginning of this program.
* It contains no records.
*
SELECT A
USE &joinfile
APPEND FROM &firstfile
```



```
GO TOP
SELECT B
* It is assumed that the secondfile and its INDEX have the
* same name. The INDEX file must be indexed on the key_expr
USE &secondfile INDEX &secondfile
SELECT A
* Set relation between the two work areas
SET RELATION TO &key_expr INTO B
DO WHILE .NOT. EOF()
   SELECT B
   IF .NOT. EOF()
      SKIP
      SELECT A
         * Add to joinfile from secondfile
      REPLACE &areplace
   ENDIF .NOT. EOF()
ENDDO WHILE .NOT. EOF()
CLEAR ALL
SET TALK ON
RETURN
* End-of-file: S-JOIN.PRG
```

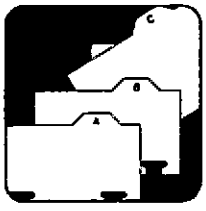
The program uses work areas and macros. The two files are joined according to the RELATION "Lastname+Firstname".

Setting Up Views and Fields

Two new features in dBASE III PLUS can help you quickly establish relations between work areas and narrow down the database fields you need. The CREATE VIEW command is a full-screen operation that allows you to choose from a list of the database, index, and format files and set up relationships between the files. CREATE VIEW also establishes any filtering options you may be using with the files. You then save the VIEW to a view file, which has the .vue extension.

Whenever you want to reuse the same relationships, instead of recreating the relationships and filters, use SET VIEW TO <view filename>. To change a .vue file, use MODIFY VIEW <view filename>.

Of help to programmers is the CREATE VIEW <view filename> FROM ENVIRONMENT command. This command establishes a view file from the current working relationships. It's an alternative to using the full-screen CREATE VIEW feature.



One aspect of a view may be a list of the specific fields you need. The new SET FIELDS command allows you to pick and choose only the fields necessary and keep them in a pool of fields. When you issue commands such as LIST, DISPLAY, and BROWSE, dBASE III PLUS automatically uses only the fields in the pool.

In the original example used to illustrate SET RELATION, after establishing the relation, you could have set up the fields you wanted to DISPLAY with SET FIELDS TO <field names>:

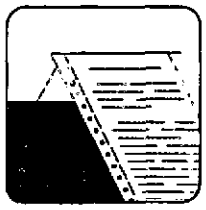
```
SET FIELDS TO First, Last, Private->Salary
```

Then, whenever you want to use only these three fields, you must first SET FIELDS ON, and then set up the appropriate commands:

```
SET FIELDS ON
DO WHILE .NOT. EOF()
   * No need to list the fields here,
   * because they're in the pool
   DISPLAY
   SKIP
ENDDO
```

You can add more fields to the current pool without removing the other fields by using SET FIELDS TO <fieldnames>. When you want to have access to all fields but keep the pool active, use SET FIELDS OFF. Use SET FIELDS TO or CLEAR FIELDS to remove all fields from the pool.

Chapter 12
PRINTING



Besides presenting information on the screen, your program may also provide printed output, such as reports. Therefore, you need to write program code tailored to your printer.

What This Chapter Covers

In this chapter you will learn:

- What things about your printer are important to dBASE III PLUS
- How to change the printer port from within dBASE III PLUS
- How to use printer coordinates for customized printed output
- The factors involved in determining page formatting
- How to do special printing features from within your programs
- How to set up a counter for printed lines and page numbering

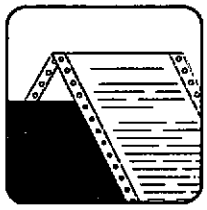
Preparing for This Chapter

Have an understanding of dBASE programming in general and a thorough understanding of screen coordinates. If you're not familiar with your printer already, have your printer manual handy for reference.

Your Own Reports and the dBASE III PLUS Report Feature

The word *report*, as used in this chapter, refers to any printed output. For many situations, you may wish to use the REPORT FORM <report filename> TO PRINT command in your programs. dBASE III PLUS's REPORT generator is useful for producing quick reports and for calculating totals for numeric fields. See *Learning dBASE III PLUS* for more information about the REPORT command.

If you want or need more elaborate printed output in your programs, dBASE III PLUS has other special commands and functions for printing that can make your programming tasks easier.



Printers: Some General Remarks

Unlike most application programs, dBASE III PLUS doesn't require you to run a special printer installation procedure. As far as dBASE III PLUS is concerned, your printer is just another output device, like the screen, that shows information.

There are three important rules about using the printer:

- The printer must be correctly connected to the computer.
- The printer must be turned on and paper must be properly inserted before you issue a printing command.
- You must tell dBASE III PLUS when to start and when to stop sending output to the printer.

Because dBASE III PLUS requires so little information about your printer, you must create any desired special printing effects in your programs. This will entail more careful planning on your part.

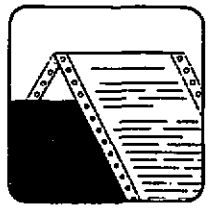
Connecting the Printer

For information on properly connecting your printer, read your printer manual. However, here are some general things you should know.

You must notify the operating system regarding the port to which the printer is attached and the specific configurations that it needs. This is especially important if you're using a serial printer connected to a serial port named by DOS — either COM1 or COM2. The computer's operating system assumes by default that you're working with a parallel printer connected to a parallel port called LPT1. For a serial printer, you have to set up a batch command to redirect the output to the proper port, COM1 or COM2, and change the type of output.

For most computers, you use the DOS MODE command with the specific settings for your serial printer. Here's an example for a serial printer connected to COM1 which operates at 1200 baud, uses odd parity, 7 data bits, and 1 stop bit:

```
MODE LPT1:=COM1:
MODE COM1:=1200,o,7,1,p
```



The first MODE line redirects the output from the default printer, LPT1, to the serial printer connected to COM1. The second line issues the settings for the serial printer. Note the p parameter, which stands for printer, at the end of the line.

Set these commands up in a batch file, such as the Autoexec.bat file, with the DOS Mode.com file on the dBASE III PLUS System Disk #1. Then they will run automatically when you start your computer. You can also issue them from dBASE III PLUS with the RUN command, but only if the Mode.com file is also on System Disk #2.

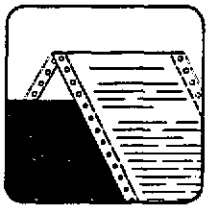
A new feature of dBASE programming allows you to change printer ports with the SET PRINTER command. For example, if you are working with two printers connected to COM1 and COM2, you can switch between them by typing SET PRINTER TO COM1.

Note that you don't need the colon at the end of COM1 in dBASE programming.

Sending Output to the Printer

There are two commands for sending output to the printer in dBASE III: SET DEVICE TO PRINT and SET PRINT ON. They are not the same command, and you should be thoroughly aware of their differences before you use them.

In dBASE III PLUS, the default output device is the screen. You use SET DEVICE TO PRINT when you have elaborately designed reports. Make sure that you immediately return the output to the screen after printing is finished, by issuing the command SET DEVICE TO SCREEN. When you SET DEVICE TO PRINT, you can use @...SAY and coordinates to position the output anywhere on the page, just as you do with output to the screen. This is called formatted output. Printed output is not simultaneously shown on the screen.



The SET PRINT ON command sends output to the printer and to the screen, unless you SET CONSOLE OFF. However, SET PRINT ON won't send the contents of @...SAY lines to the printer. It can only handle unformatted output, the kind produced by the DISPLAY, LIST, ?, or ?? commands. The checkbook management system, which requires a simple display, uses SET PRINT ON exclusively.

Command	Screen	Printer	Used with:
SET DEVICE TO PRINT	Freezes	Starts Printing	@...SAY
SET DEVICE TO SCREEN	Can Change	Stops Printing	@...SAY, ?, ?? DISPLAY, LIST
SET PRINT ON	Can Change	Starts Printing	?, ??, DISPLAY, LIST
SET PRINT OFF	Can Change	Stops Printing	@...SAY, ?, ?? DISPLAY, LIST

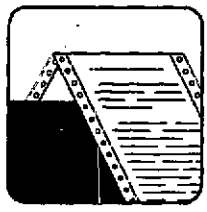
Table 12-1 The screen and print commands

Both SET DEVICE TO SCREEN and SET PRINT OFF, the defaults, allow you to use any display command.

SET PRINT ON has another useful function: it allows you to instruct the printer to do special effects, such as printing in italics or condensed type.

Printer Coordinates

Use @...SAY commands with the command SET DEVICE TO PRINT so that you can format the exact position of the printed output, with row and column coordinates. When designing reports, you can use copies of your on-screen forms as the models for printed reports, and make changes to the coordinates as necessary. This saves planning and design work. With one-page reports, you can often use the original screen form without any major changes.

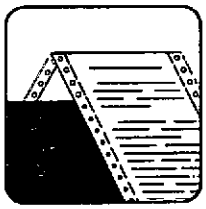


The alternative is to use ? and ?? commands with SET PRINT ON. For example, the checkbook management system uses a simple routine, Printer, to toggle output between the screen and the printer in all its reports. This routine is a PROCEDURE file. PROCEDURE files are the topic of Chapter 16. Here is the Printer routine:

```
@ row,6 SAY "Do you want the output sent to the " +;
            "printer or the screen? (P/S)"
pr = " "
DO WHILE .NOT. pr$"PpSs"
   pr = " "
   @ row,70 GET pr
   READ
ENDDO
CLEAR
IF UPPER(pr) = "P"
   SET PRINT ON
ENDIF
RETURN
```

WARNING
dBASE III PLUS can't make the printer go backwards or skip up to a previous line. You must issue the @...SAY commands in top-to-bottom and left-to-right order for printing output. This is decidedly different from setting up screen displays. If you issue a coordinate that is lower than the previously issued one, dBASE III PLUS causes the printer to eject a page. For example, if you've just printed text at coordinates 5,10 and then try to print something else at 4,10, dBASE III PLUS knows it just printed on line 5, so the printer ejects the paper to the next top-of-form setting and moves down to row 4 to print the next line.

You can't use the @...GET and READ commands in printed output. You must first GET the information in memory variables, SET DEVICE TO PRINT, and then use @...SAY lines to print the data.



How you set up printer coordinates also depends on several considerations that you don't need to know when you design screen coordinates: (1) where the paper is in the printer, (2) the size of the paper you're using, and (3) the size of the typestyle.

The Position of the Paper in the Printer

dBASE III PLUS does not know where the top-of-form setting is on your printer, nor does it know where the printer begins to print at the left edge of the paper.

After you have determined these two locations, always make sure that the paper is inserted at the same spot and that the print head begins at the same location. Consider putting a message in your program as a reminder before printing begins.

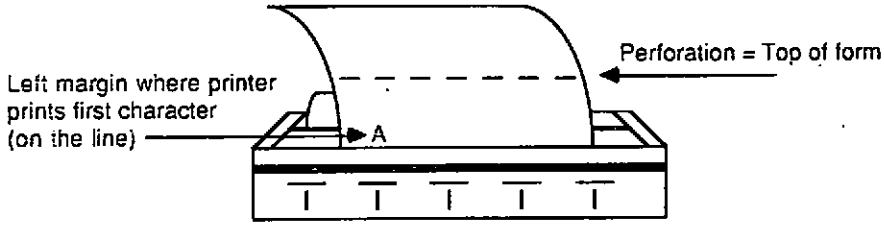
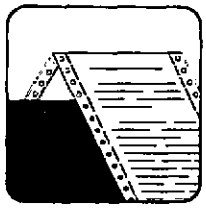


Figure 12-1 The top-of-form and left edge of paper settings

Paper Size

Just as you have to restrict your screen displays to the size of the screen, you must consider the size of the paper, labels, or envelopes. dBASE III PLUS assumes that you are printing on standard 8 ½ by 11 inch paper, which has 66 lines. You'll reserve some of these lines for blank top and bottom margins, but your program still has to keep count of them to ascertain when one page is filled and when to issue a form feed instruction for the next page. If you use a paper size other than 8 ½ by 11 inches, you must adjust your printer coordinates.



> **NOTE**
> This book refers to horizontal lines of printed output to distinguish them from horizontal rows of screen output, although they are really the same thing:
>
> ```
> @ 5,10 SAY 'Hello there'
> ```
>
> can mean either the sixth row on the screen or printed line six. The vertical coordinate is called a column in both cases.

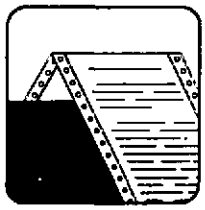
It's entirely possible to have printer coordinates that go beyond the normal screen coordinates. For example, the bottom of the screen is row 24, but you can have many more than 24 printed lines on a page. The highest coordinate you can use in either a line or column is 255.

Typestyle Size

Most printers use a pica typestyle that prints ten characters per horizontal inch. So, there are approximately 85 characters, or columns, across the page, although you'll need to reserve a few columns for the left and right margins. However, if you use a smaller typestyle, such as elite, which prints 12 characters per inch, you'll have to adjust your screen coordinates accordingly.

Switching Between Screen and Printer

When you SET DEVICE TO PRINT, the screen does not go blank; it stays the same as it was immediately before you routed output to the printer. Take advantage of this by prompting the user with messages on the screen, for instance, when the user has to insert a new sheet of paper. To do this, you must switch back and forth between the two devices, SCREEN and PRINT:




```
CLEAR
@ 10,25 SAY 'Insert paper correctly'
?
WAIT SPACE(20)+'Press any key to begin printing'
SET DEVICE TO PRINT
* Do printing
* (...)
* One page completed
SET DEVICE TO SCREEN
CLEAR
? CHR(7)
? CHR(7)
@ 10,22 SAY 'Insert another piece of paper and '
?
WAIT SPACE(25) + 'Press any key to begin again'
SET DEVICE TO PRINT
* (...)
```

Because the user may not be watching the screen, you've programmed the bell to ring twice to draw attention to the screen message.

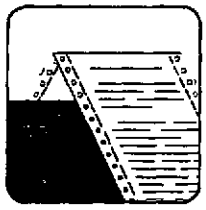
Page Formatting

Because you can't position characters on the screen border, the screen coordinates don't allow you to use the entire screen. However, you can take control of your entire printed page. Determining the look of the page is known as page formatting. Here are some observations about page formatting.

Margins

The white space bordering the top, bottom, left, and right sides of the text is called the margin. The top and bottom margins are just blank, unprinted lines. The left and right margins depend on the position of the paper in the printer, where the printing begins on the page, the length of the printed line, and the size of the typestyle used.

It's important to know where the vertical screen column 0 position prints on your printer. After you have determined that and adjusted your paper accordingly, you can set up your printer coordinates relative to this position. For example, use a simple program like this one to see where column 0 prints:



```
* TEST.PRG - tests where output appears on the printer page
SET DEVICE TO PRINT
@ 0,0 SAY 'Testing ...'
EJECT
SET DEVICE TO SCREEN
```

Then adjust the paper position accordingly. Another useful feature is the SET MARGIN command, which changes the beginning left margin for printed output. You must supply an integer number:

```
SET MARGIN TO 5
```

The printer starts printing at the new left margin, which is five columns to the right of the default printing position. However, the printer coordinates will still start at 0. Using SET MARGIN is therefore a good way to adjust where the printing is to start on the page without changing the numbering of the coordinates in your program or the positioning of your paper in the printer.

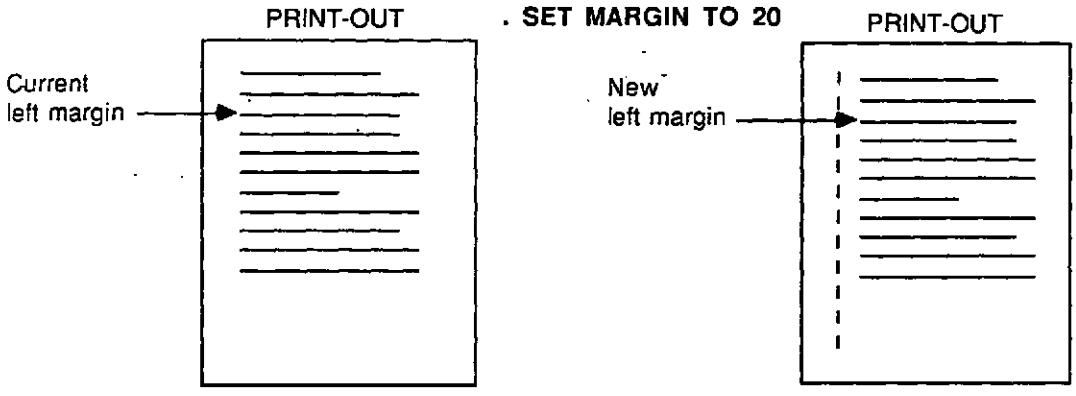
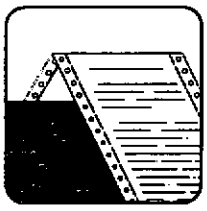


Figure 12-2 How the SET MARGIN command works

The printer prints the very same output starting at a different left margin on the page.



Paper Length

See where dBASE III PLUS prints screen line 0 on your paper and then adjust the top-of-form setting accordingly. You could use the same sample test program shown above. Then, for both top and bottom margins, you can skip some blank lines to start the printing on a specific line.

Keep track of how many lines are left on the page and how many blank lines you want for a bottom margin. If your printouts are always the same length, you'll have little trouble here. However, if the number of lines in the printout varies, you will have to set up counters to regulate printing, which you'll learn more about below.

Headers and Footers

You may wish to have running titles at the top or bottom of your printed pages. These titles are known as headers and footers. One common header or footer is a page number.

You must consider the total number of lines per page, the number of blank lines for the top and bottom margins, and the number of lines of your headers and footers when you set up your pages. At the end of the chapter is an example which shows you how to create a three-line header.

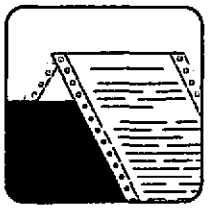
Starting a New Page

You can force the printer to begin a new page with the EJECT command. It issues what is known in the computing world as a form feed, which is ASCII code 12. This code instructs the printer to advance the paper the number of unprinted lines left on the page.

> **WARNING**
> This command can only work properly if the top-of-form has been set correctly before you begin printing.

Trouble Spots

The last line of the page and the position of the print head are two potential problem areas in your program. Here's how to handle them.



The Last Line

Printers often have a buffer area to store the individual characters in a line before printing the line. The print buffer is like a holding zone for these characters. A printer will print the entire line and clear the buffer only when it receives a carriage return code, ASCII 13. Most programs send the carriage return code at the end of the line, but dBASE III PLUS sends this instruction at the beginning of the line, so your printer may not print out the very last line as soon as you hoped. So, always include an EJECT command, which sends a carriage return code and a form feed code, at the very end of the print job. Alternatively, you could put this module at the end of your print job:

```
SET PRINT ON
?? CHR(13)    && Sends carriage return code
SET PRINT OFF
```

or this:

```
SET DEVICE TO PRINT
@ 23,0 SAY CHR(13)    && Sends carriage return code
SET DEVICE TO SCREEN
```

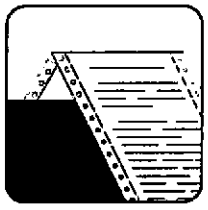
Realigning the Print Head

Make sure that you instruct the printer to realign the print head to its normal location at the end of each printout, because dBASE III PLUS doesn't do this automatically. Using an EJECT command at the very end of the printout forces the print head back to its home position.

Suppressing Initial Page Ejects

In certain cases, printing with @...SAY will result in an initial page eject. This situation often occurs when the last printer or screen row and column position is greater in value than the first print position. To work around the problem, do the following:

1. Check that the last print operation was terminated with an EJECT command.
2. Issue an @ 0,0 just prior to the SET DEVICE TO PRINT statement in your printing routine.
3. Issue an EJECT command prior to the SET DEVICE TO SCREEN command in your printing routine.



If you're using a REPORT FORM, use the NOEJECT option to suppress the intial page eject.

Special Effects

You can take advantage of most of your printer's special effects, such as italics on dot-matrix printers, from within dBASE III PLUS if you know what codes the printer needs to print them.

Escape Codes

Printers need a control code from the software to do a special printing effect. There is no standardization for these codes among printer manufacturers, so you'll have to refer to your printer manual to find the codes you need. However, most printer control codes begin with the **Esc** key (ASCII 27), followed by other ASCII codes, so they are often referred to as escape codes. Here's an example. Perhaps your dot-matrix printer will print in italics when it receives the control codes **Esc** F. You can write this in dBASE as:

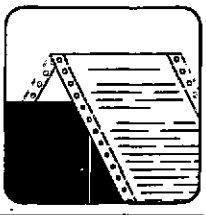
```
SET PRINT ON
?? CHR(27) + "F"
```

Note that the entire control code is a string. Similarly, the way to stop italic printing on your printer is with **Esc** G, which translates in dBASE to:

```
SET PRINT ON
?? CHR(27) + "G"
```

Setting the Form Length

If you are using a form that has a page length less than the standard 11 inches, and you are sending a report to the printer with the REPORT FORM or @...SAY statements, send a control code sequence that configures your printer for shorter page length. By doing this, an EJECT or form-feed will advance the paper to the proper top-of-form for the nonstandard page length.



For example, you may wish to print checks from a command file using @...SAY statements. Each check has a page length of between five and eight inches. To set up the print run:

```
SET PRINT ON
?? CHR(27) + "C" + CHR(45)
SET PRINT OFF
```

If you are using the REPORT FORM, be sure that you define the page length in the REPORT FORM to less than the number of lines that you have set for the form length. Otherwise, your page breaks may not occur at the line you expect.

Null Characters

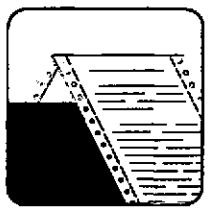
In dBASE III PLUS you cannot send null characters (00 hexadecimal) to the printer. This can be a problem for printers that require a null character to be sent as part of a string of control codes. For example, in order to change the form length or engage underlining with any of the Epson line of printers, you must send a null character to the printer.

There is a way around this. CHR(0) is a null character which cannot be sent to a printer or any device. However, some printers interpret CHR(128) as CHR(0). Notice that the binary representations of CHR(0) and CHR(128) are very similar:

```
CHR(0)   = 0000 0000
CHR(128) = 1000 0000
           ^—eighth bit
```

If your printer supports 8-bit characters and has an option to turn off the eighth bit, first send the printer control codes to turn off the eighth bit, then send CHR(128). Your printer will read CHR(128) as 0000 0000 and interpret it as CHR(0). If your printer does not support 8-bit characters, send only CHR(128) to substitute for CHR(0).

The following example will set the page length to seven inches on an Epson FX-80 printer. The first escape code sequence turns off the eighth bit, the second escape code sequence sets the page length to seven inches, and the third escape code sequence returns the printer to normal. The ? issues a line feed.




```
SET PRINT ON
?? CHR(27) + "="
?? CHR(27) + CHR(67) + CHR(128) + CHR(7)
?? CHR(27) + "8"
?
SET PRINT OFF
```

Printing Special Effects

Once you know the exact escape code sequences you need, you can instruct the printer to do special effects with the SET PRINT ON command and the correct escape codes, before you begin printing:

```
SET PRINT ON
?? CHR(27) + "F"
SET PRINT OFF
```

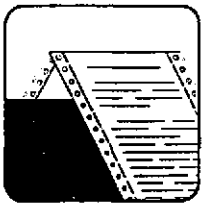
Because you don't want a superfluous carriage return/line feed instruction to be sent to the printer, which is what the ? command does, use the ?? command.

> **TIP**
> Set up the escape codes just before you begin printing and then return the printer to normal just after you've finished. Otherwise, the printer continues to print in the special mode the next time you use it.

If you want several special printing features at once, such as condensed, bold, and italic print, include all the correct escape codes on one command line:

```
SET PRINT ON
?? CHR(27) + "F" + CHR(27) + "P1" + CHR(27) + "Q"
SET PRINT OFF
```

Make sure to issue all the necessary cancellation codes at the end of the print job.



Different Printers

If your program has to control several different printers, set up separate memory files with the specific control code sequences for each printer. Use common names for the special printing effects. For example, here are the setup variables for printer model XYZ:

```
* Set italics on
STORE CHR(27) + 'E' TO italicon
* Set italics off
STORE CHR(27) + 'F' TO italicoff
* Set expanded print on
STORE CHR(27) + 'G' TO expandon
* Set expanded print off
STORE CHR(27) + 'H' TO expandoff
```

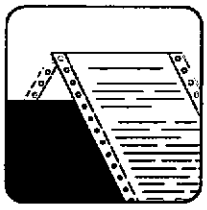
SAVE each set of codes in a memory file. Even if you have different control codes for printers, use the same memory variable names. Instruct your program to RESTORE the memory variable file containing the correct codes for the printer it needs. Then, whenever you want to assign an attribute in your program, use the memory variable name:

```
* Turn on italics
SET PRINT ON
?? italicon
SET PRINT OFF
```

This method allows you to use many different printers without altering the program code.

Relative Addressing

As with screen displays, if you know the exact length of fields and variables, you can set up exact printer coordinates. However, you'll have to use other techniques with data of differing lengths and for such changing situations as page numbering.



Just like the ROW() and COL() functions for the screen, you can use relative addressing techniques in printed forms with two special functions, PROW() and PCOL(). They return the current printer row and column coordinates. So, you can use these functions to advance the printing down the page:

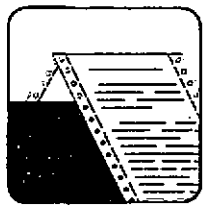
```
SET DEVICE TO PRINT
@ PROW(),PCOL() SAY TRIM(Last) + ', ' + First
* Next one is two lines down from the first
@ PROW()+2,PCOL() SAY Street
   * (...)
SET DEVICE TO SCREEN
```

When used with the SET MARGIN command, the value of the current print head's vertical position, PCOL(), depends on the new margin setting. If you use the ? and ?? commands to position output, you must use the SET PRINT ON command, not SET DEVICE TO PRINT. Refer to Chapter 8 for more about relative addressing techniques.

Determining Page Breaks

With multiple page reports, you'll probably want to write a program that keeps track of the number of printed lines on the page and then starts a new page when the previous page is filled. This program should also keep track of the current page number. You can use simple counters for both these numbers.

The following module is adapted from an example in the *Advanced Programmer's Guide*, pages 384-385. It keeps track of the line and page counters, and begins a new page when the line counter is greater than 60. It also uses a trick to avoid the EJECT command. When the line counter, *tline*, is greater than 60, the IF construction decreases *tline* to 1. Because dBASE III PLUS automatically starts a new page if the printer coordinates are less than the previous coordinates, the IF construction begins a new page without an EJECT command.



```
* Initialize counters to starting values
* Start tline high enough to take the branch
* for a new page just inside the first DO WHILE loop
SET TALK OFF
STORE 61 TO tline
STORE 5 TO tcolumn
STORE 0 TO pagenum
* Prepare the name and address file for printing
USE Names INDEX Last
GO TOP
* Route output to printer
SET DEVICE TO PRINT
* Start loop
DO WHILE .NOT. EOF()
   * Branch for new page
   IF tline > 60
      STORE 1 to tline
      * Increment page number counter
      STORE pagenum + 1 to pagenum
      * Start new page because tline is now less than
      * at the beginning of the IF construction
      @ tline,   tcolumn+66 SAY "Page" + STR(pagenum,3)
      @ tline+1, tcolumn+66 SAY DATE()
      @ tline+4, tcolumn+30 SAY "Names and Phone Numbers"
      STORE tline+6 to tline
   ENDIF
   * Show information from database file, in this instance
   * the First, Last, Area, and Phone fields
   @ tline,tcolumn SAY TRIM(First) + " " + TRIM(Last) + ;
       "   " + Area + " " + Phone
   * Go to next record and increment line counter
   SKIP
   STORE tline + 1 to tline
ENDDO
* Reset printhead
EJECT
* Route output to screen
SET DEVICE TO SCREEN
RETURN
```

Notice how this module also sets up a series of running header lines at the top of each new page.

Chapter 13
HOUSEKEEPING

In programming, housekeeping refers to the finishing touches put at the end of your program. In the world of computers, housekeeping also means keeping track of files, regulating available disk space, making backup files, copying files or records, and related operations.

What This Chapter Covers

In this chapter you will learn:

- What to do before ending your program
- How to reinstate the dBASE III PLUS working environment
- Important issues of day-to-day file maintenance
- How to manipulate files without knowing their names
- How to deal with disk space from within your program
- How to modify the structure of a database file from within your program

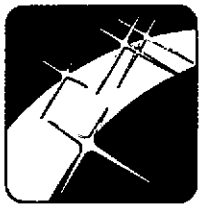
Preparing for This Chapter

You should have a general understanding of the basics of dBASE programming.

Completing the Program

If you follow the one entry/one exit rule when designing your programs, your user will have one, and only one, way to end the program and return to the dot prompt. However, before the program officially ends, it should tidy up the work space that it has used and make sure that there are no loose ends, such as open database or format files.

The housekeeping your program has to do at the end of the run depends on how you have designed it. In the checkbook management program, for example, there are no open database files that must be closed when the user chooses **X** to exit. All database files are closed by the subprograms that USE them.



However, the main program module does have to check if any transactions have been made during the run of the program, before returning to the dot prompt. Here's the section of code that handles this chore:

```
* test for exit condition
CASE CHR(i) $ "Xx"
  * retain variables -'balance','lastchk','lastwth' and
  *  'lastdep' only if changes were made.
  IF balance<>mbalance .OR. lastchk<>mlastchk .OR. ;
     lastwth<>mlastwth .OR. lastdep<>mlastdep
     RELEASE ALL LIKE m*
     RELEASE i,today
     SAVE TO Chkbook.mem .
  ENDIF
```

The program ascertains if any changes have been made to the original amounts in the Chkbook.mem file by comparing them to the contents of the four variables *mbalance*, *mlastchk*, *mlastwth*, and *mlastdep*. If so, it RELEASEs the variables used by the program so that only the four original variables, *balance*, *lastchk*, *lastwth*, and *lastdep*, are left in memory. It then SAVEs these variables to Chkbook.mem for the next time.

Closing Files

Make sure that the program closes all database and format files before returning to the dot prompt. The checkbook management system does this in the subprogram modules, by adding this command to the end of the subroutine:

```
CLOSE DATABASES
```

This useful command closes not just database files, but INDEX and FORMAT files too.

When you design your programs, remember the importance of database integrity and have your program close database files as soon as the program is finished with them.

Clearing Memory

With the RETURN command, dBASE III PLUS automatically clears PRIVATE memory variables but not PUBLIC variables. However, if you want to clear all memory variables at the end of the program, use the CLEAR MEMORY command. Because it erases everything from the memory buffer and the next highest program level may need these variables, using CLEAR MEMORY is a potentially hazardous situation. Make sure that you understand what you're doing.

Returning to the Default Environment

It is important that the user return to the standard dBASE III PLUS working environment, the dot prompt. Always reset the working environment to the way it was immediately before your program began. This means changing all the SET commands that your program uses. For example, you probably SET TALK OFF, so at the end of the program just SET TALK ON. The checkbook management program does this immediately after determining whether any transactions have been made. SET STATUS ON is left optional; if you want the status bar to appear, remove the asterisk at the beginning of that line:

```
* clear variables and return to calling program
* or dBASE system
SET TALK ON
SET ESCAPE ON
SET BELL ON
SET HEADING ON
SET HELP ON
SET SAFETY ON
* SET STATUS ON
CLEAR ALL
CLEAR
RETURN
```

Similarly, if you've changed any of the function keys for other purposes in your program, reset them to their default settings. A list of these defaults is under SET FUNCTION in the Commands and Functions reference sections of *Using dBASE III PLUS*.

Notice that the program also uses the CLEAR ALL command to clear out memory variables and close all files. Finally, the program CLEARs the screen and RETURNs the user to the dot prompt. The dot prompt is always on line 21 of your screen.

Working with the Disk

Without knowing the names of files, your program may have to choose among many files on the current disk drive, or to determine how much disk space is available. Here are some standard operations for dealing with the disk.

Finding a Database File

Most of the time, your program works with specific database files that you set up during the design of your project. You open the file with USE, just the same as at the dot prompt. Because filenames and relationships are determined during the program design stage, this method affords you the most control over the application. However, if the program must request a filename from the user, use the FILE() function.

The FILE() function returns a logical value, true or false, depending on whether or not the file specified in the parentheses exists on the current disk in the current directory. You supply the filename as a memory variable (containing the filename character string) or directly, enclosed in quotes. In addition, FILE() must have the complete filename, including extension. So, make sure that you set up the filename correctly by using a memory variable.

Here's a typical program module that requests a dBASE III PLUS database file, converts it to a full name with extension, and then ascertains whether it exists on the current drive:

```
* Initialize memory variable to hold file name
* without extension
mfile = SPACE(8)
CLEAR
* Get file name converted into uppercase
@ 10,5 SAY 'Enter the filename:' GET mfile PICTURE '@!'
READ
* Convert to full filename with extension
mfile = LTRIM(TRIM(mfile)) + '.DBF'
* Check to see if file is on the disk
IF FILE(mfile)
   * More commands
   * (...)
ENDIF
```

Unless otherwise instructed, the FILE() function searches only the current drive and directory, even if you SET PATH TO another directory. If you want to search another directory, include the path name with the filename in the FILE() function:

```
STORE 'C:\DBASE\WORK\' TO path
STORE 'FILE.DBF' TO file
IF FILE(path + file)
* (...)
  ENDIF
```

Examining the Work Area

Your program may need to know what database and index files are open in the currently SELECTed work area. This is especially helpful if files are open before the program begins and the program has to USE these files later, or if the program doesn't know the names of the database files.

The DBF() function returns the name of the current database file in USE, if any, or a null string if no file is in USE. If there are several work areas in USE, you must SELECT each one and use DBF() in each to determine the name of the database file currently in USE in that work area.

Similarly, the NDX() function lists the name of the index file or files in USE in the currently SELECTed work area. Because dBASE III PLUS allows up to seven index files to be open for each database file, the syntax of this function is slightly different than for DBF(): you must use the number of the index file that you're looking for. If no index file is open, dBASE III PLUS returns a null value.

You can use the NDX() function to manipulate the index files without knowing their real names:

```
SELECT 1
STORE NDX(1) TO firstndx
USE File INDEX &firstndx
```

You can also determine the names of fields in a database file from a program, without knowing the structure of the database file, with the FIELD() function. This function requires an integer from 1 to 128, the number of possible fields in any dBASE III PLUS database file, as its argument. It returns the field name for the position in the database file represented by that number:

```
STORE FIELD(1) to firstfld
GO TOP
DISPLAY &firstfld
```

Note that all three functions, DBF(), NDX(), and FIELD(), return strings. These three functions allow your program to work with any database and index files without knowing the specific file or field names in advance.

In addition, you can check when the database file in USE in the currently SELECTed work area was last changed or edited. The LUPDATE() function returns a date value. This function is handy if your program needs to prevent duplication of certain operations, such as a daily totaling of amounts.

File Size and Disk Space

Because dBASE III PLUS needs disk space to INDEX or SORT a file, the program's ability to determine available space can be crucial. There are three useful functions for dealing with available space: RECCOUNT(), RECSIZE(), and DISKSPACE().

The RECCOUNT() function gives the total number of records in a database file. This number always includes records marked for deletion, even if SET DELETED is ON. Use this function to see if a database file is getting too big:

```
USE Names
IF RECCOUNT() > 2000
   ? CHR(7)
   @ 10,10 SAY 'Time to delete a few records'
   * (...)
ENDIF
```

The RECSIZE() function gives the size, in bytes, of one record in the database file. It presents the sum of all fields in a record. One byte equals one character of information, such as a letter or punctuation mark. To get the total size of all records in the database, multiply RECSIZE() by RECCOUNT():

```
USE Names
STORE RECCOUNT() * RECSIZE() to size
```

This is not the total size of the database file, because dBASE III PLUS also maintains what it calls a header of information in the database. The header keeps track of field names, lengths, and types. You must add the size of the header to the calculation of total file size. To determine the size of the header, first you need to know the number of fields in the database file. To get this number, use the following program:

```
USE Names
numfields = 0
null = ""
DO WHILE null < FIELD(numfields + 1)
   numfields = numfields + 1
ENDDO
USE
```

The DO WHILE loop increments the number of fields variable, *numfields*, by one, as long as the value of the FIELD() function, which is a string, is greater than the null string. Recall that a null string has a length of 0. So, if there is a field corresponding to numfields + 1, the result of the FIELD(numfields + 1) function will be the name of the field, that is, a string that is larger than null. However, as soon as the DO WHILE loop gets to the field number after the last field in the database, FIELD(numfields + 1) returns a null value and the loop terminates.

Once you know the number of fields in a database file, you can find out the size of the database file header by multiplying the number of fields in the database times 32 and then adding 34:

```
header = (32 * numfields) + 34
```

Using all the above modules and memory variables, the program can compute the size of the entire database, that is, the total records plus header and the end-of-file marker, which takes up one byte:

```
totalsize = size + header + 1
```

The program can then compare the total size of the database file with the amount of space left on the disk. The DISKSPACE() function returns the available disk space in bytes. You can use DISKSPACE() with RECCOUNT() and RECSIZE() to ensure that there is enough room for a backup or temporary file.

For example, your program has determined the total size of a database file and STOREd this amount in the *totalsize* variable. Before the program SORTs this file, it can test to see if there is at least twice as much space as the size of the file on the disk to accommodate the sorted file:

```
USE File
* Determine size of file and STORE in variable totalsize
* See above examples using FIELD(), RECCOUNT(), and RECSIZE()
* (...)
* When that is done, check space
IF DISKSPACE() < totalsize * 2
   ? CHR(7)
   @ 10,10 SAY "There isn't enough room on the disk to sort ;
       this file"
ELSE
   SORT ON Amount DESCENDING TO Temp
ENDIF
```

NOTE
Depending on how complicated they are, index files may take up more disk space than the original database file. A SORTed file generally takes up the same amount of room, if all records are SORTed. However, a SORT requires that there be twice as much disk space available as the size of the file.

Another use for DISKSPACE() is to allow a program to back up a large file from a hard disk to several floppies by copying only a certain number of records. Check out the example of this operation under the RECSIZE() function in the Commands and Functions sections of *Using dBASE III PLUS*.

File Maintenance

You've probably devised your own methods and schedule for managing your disk files. Here are some additional suggestions.

Deleting and Renaming Files

You must know the complete name of a file in order to delete or rename it. So, use the technique outlined under the FILE() function to include the file extension in the name. Better yet, try to get your user to give you the complete filename. Use a template as a guide:

```
file = SPACE(12)
@ 10,10 SAY 'Enter name of file to delete'
@ 12,10 SAY ' including file extension:'
@ 14,5 GET file PICTURE '!!!!!!!!.!!!'
READ
STORE LTRIM(file) to file
IF FILE(file)
   ERASE &file
ELSE
   ? CHR(7)
   @ 18,10 SAY 'There is no file by that name on the disk'
ENDIF
* (...)
```

You can't use wildcards with the ERASE or RENAME commands to delete or rename groups of similarly named files. You must repeat the deleting or renaming step for each file.

Copying Files and Backups

You use temporary scratch files from ASSIST or the dot prompt when you COPY certain or all records from a database file. Your program can make use of scratch files for the same purpose. It is important that the program first ascertain whether there is enough space on the disk for the copied file. Periodically, your program should also make backups of all database files, including index files.

> **WARNING**
> If you have a hard disk, don't use the DOS RESTORE command to restore backup files without first uninstalling dBASE III PLUS.

Using Scratch Files

You've seen that the use of scratch files is not unique to programming. You have already learned that many accounting firms maintain all daily transactions in a temporary posting file. Periodically, they update the general ledger file with the information from the posting file. Consider using scratch files if your program makes many changes to the database file. Scratch files are merely copies of the original database files to which changes or additions are made. Be sure to provide for frequent and periodic updates of the scratch files into the main database file. Delete unnecessary scratch files when you're finished with them so that you don't get confused.

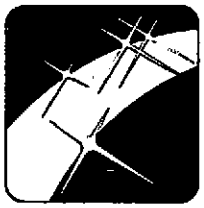
Importing and Exporting Files

Copying files usually refers to making an exact copy. However, if your program has to copy, or import, files that are in another format, such as files originally created in WordStar's non-document mode, you will either use the APPEND FROM <filename> DELIMITED or the APPEND FROM <filename> SDF command. Similarly, if you want to copy dBASE III PLUS files to another format, or export them, use the COPY TO DELIMITED command. There are also special commands, IMPORT and EXPORT, for working with files in the pfs:File™ format. Look in *Using dBASE III PLUS* for more information about these commands.

Modifying a Database Structure

One full-screen command to avoid using in your programs is MODIFY STRUCTURE. Think of what could happen if your users were allowed to change the structure of a database file. There is a dBASE programming alternative for MODIFY STRUCTURE which, again, gives you more control over what's happening and allows you to verify the user's request before actually changing the database.

To modify the structure of a database file from a program, first COPY the file's structure to a temporary database file, using the EXTENDED option. This option turns the structure into a series of records. Deleting or changing the records that you don't want actually deletes or changes the fields. You then use the CREATE FROM command to create a database file with the new structure and, finally, the APPEND FROM command to bring in the original records.



Start dBASE III PLUS, SET the DEFAULT drive, and insert the Sample Programs and Utilities disk in this drive. Then, USE the Checks.dbf file from the checkbook management system. The following exercise shows how to modify a database file structure from within a program.

> **WARNING**
> If you want to work along with this example, make a copy of the Checks.dbf file first. Do NOT use the original.

If you DISPLAY the STRUCTURE of the Checks.dbf file, here's what you would see:

```
Structure for database: B:checks.dbf
Number of data records:     0
Date of last update    : 06/10/85
Field  Field Name Type       Width Dec
    1  CHKNO      Numeric        4
    2  PAYTO      Character     30
    3  AMT        Numeric       10   2
    4  CAN        Logical        1
    5  DATE       Date           8
    6  MEMO       Character     25
    7  TAX        Numeric        1
** Total **                     80
```

To modify the structure from a program, first COPY this structure TO a temporary file STRUCTURE EXTENDED:

```
USE Checks
COPY TO Temp STRUCTURE EXTENDED
USE Temp
```

The fields of the original Checks.dbf file become a series of records in the Temp.dbf file, each record identified by four fields, Field_name, Field_type, Field_len, and Field_dec:

```
Structure for database: B:temp.dbf
Number of data records:     7
Date of last update    : 06/26/85
Field  Field Name  Type       Width  Dec
    1  FIELD_NAME  Character     10
    2  FIELD_TYPE  Character      1
    3  FIELD_LEN   Numeric        3
    4  FIELD_DEC   Numeric        3
** Total **                     18
```

If you LIST this file now, here's what you would see:

```
Record#  FIELD_NAME  FIELD_TYPE  FIELD_LEN  FIELD_DEC
      1  CHKNO       N                   4          0
      2  PAYTO       C                  30          0
      3  AMT         N                  10          2
      4  CAN         L                   1          0
      5  DATE        D                   8          0
      6  MEMO        C                  25          0
      7  TAX         N                   1          0
```

Request from the user which records to DELETE. You're actually deleting fields and thus modifying the structure of the original database file. For example, if you wish to eliminate the Tax field, have the program:

```
DELETE RECORD 7
```

You can also use the REPLACE command to REPLACE a field with another value to change the type or length. For instance, to reduce the length of the Payto field to 25 characters:

```
REPLACE Field_len WITH 25 FOR Field_name = "PAYTO"
```

After the program has made all necessary changes and asked the user to verify that the changes are correct, PACK the file, if any fields were DELETEd, and then save the changed file with USE. Finally, set up a new file from the extended structure of the Temp.dbf file with the CREATE FROM command. This command only works with extended structure files:

```
CREATE New FROM Temp
```

The records in the Temp.dbf file now become the fields in the New.dbf file. The program then APPENDs the records from the original Checks.dbf file to New.dbf:

```
APPEND FROM Checks
```

Finally, close the New.dbf file, delete the old files, Temp.dbf and Checks.dbf, and rename the new file to the original file, Checks.dbf:

```
USE
DELETE FILE Temp.dbf
DELETE FILE Checks.dbf
RENAME New.dbf TO Checks.dbf
```

Chapter 14
PUTTING IT TOGETHER

In this book, you've seen how dBASE programming accomplishes specific tasks, such as getting user input and working with database information. By looking at a completed program, you'll also get a larger view of how the tasks interact.

What This Chapter Covers

As an example of how an entire program module works, this chapter steps you through the Cancl.prg file, which registers canceled checks.

Preparing for This Chapter

Know the basics of dBASE programming, and have your printout of the Cancl.prg file handy.

The Setup

When the user chooses **D** from the main menu, the main program, Cbmenu.prg, branches to the subprogram, Cancl.prg. First, Cancl.prg opens the Checks.dbf database file and its corresponding INDEX, Chkno.ndx:

```
USE Checks INDEX Chkno
```

Cancl.prg then branches to Chkmask.prg, which draws a check display on the screen. The checkbook management system uses this display in two other subprograms that deal with checks: when it adds checks, Check.prg, and when it edits or voids checks, Editvoid.prg. Above the check representation, Cancl.prg then presents the title **CHECK CANCELLATION ROUTINE** on row 1 at column 26.

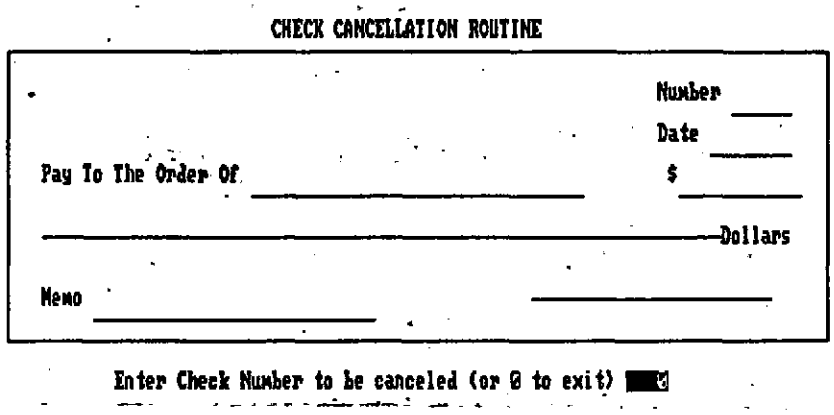


Figure 14-1 The beginning of the check cancellation program

Even before the user can enter canceled checks, Cancl.prg verifies that there are indeed checks that can be canceled. If not, it gives the user a message to that effect:

```
* check to see if there are any records in Checks.DBF
IF RECCOUNT()= 0
   @ 18,24 SAY "There are no checks in the file"
   WAIT SPACE(19)+"Press any key to return to the Main menu "
   CLOSE DATABASES
   RETURN
ENDIF
```

Cancl.prg uses the RECCOUNT() function to determine whether the number of records in Checks.dbf is equal to 0. It isn't concerned with the content of the records yet. It just checks to see if there are any records. If there are no records, Cancl.prg doesn't continue. To allow the user time to read this message and act on it, Cancl.prg uses the WAIT command. When the user presses any key, the program CLOSEs the DATABASE and RETURNs to the main menu.

If there are checks to be canceled, the program establishes the controlling loop for the rest of the check canceling routine:

```
DO WHILE .T.
```

The DO WHILE .T. loop continues to request a check number and repeatedly goes through the check cancellation process, so that the user can cancel more than one check.

Getting User Input

Cancl.prg is ready to request a check number from the user. It initializes a memory variable, *mcan*, to hold the user's input of the check number. It then requests the user to type the check number:

```
mcan = 0
* input check number to be canceled
@ 18,11 SAY "Enter Check Number to be canceled (or 0 to exit)";
  GET mcan PICTURE "9999" RANGE 0,
READ
```

The RANGE clause makes sure that the user doesn't input a negative check number.

> **NOTE**
> Even though *mcan* contains 0, the PICTURE clause regulates how large an integer number the user can type. The entire checkbook management system is set up for check numbers of four digits or less. You may wish to change the PICTURE clause for larger check numbers. Make sure, however, that you also MODIFY the STRUCTURE of the Checks.dbf file to increase the length of the Chkno field and then INDEX the database file ON Chkno TO Chkno. Change references to check numbers in the other subprograms, too.

Testing for Conditions

Eventually, the user finishes canceling checks and returns to the main menu. After the user enters a check number, Cancl.prg first ascertains whether this number is 0 and, if so, EXITs the loop:

```
IF mcan = 0
  EXIT
ENDIF
```

When determining what conditions to test, the order is important. First, test for check number 0. That way, Cancl.prg doesn't continue if the user just wants to return to the main menu.

Once the user has typed in a check number greater than 0, Cancl.prg CLEARs the screen at row 18 and looks for the check number in the database file:

```
@ 18,0 CLEAR
* search for canceled check number
SEEK mcan
```

The Checks.dbf file is INDEXed on the Chkno field, which is numeric. So, Cancl.prg uses SEEK. There are now four possible conditions:

- There is no check number at all in the Checks.dbf file, which means that the user has entered an incorrect check number.
- The check number exists, and the check has not been canceled.
- The check number exists and the check has not been canceled, but this isn't the right check.
- The check number exists, but the check has already been canceled.

Cancl.prg handles these conditions in the next IF...ENDIF construction. However, it breaks down these four conditions by nesting them within the main condition test, which checks to see if there is a check number at all:

```
IF FOUND()
```

If the check number, that is, the record containing this number in the Chkno field, is not FOUND(), Cancl.prg displays a message to that effect. This situation is handled in the ELSE command at the end of the IF FOUND() command:

```
* check is not in file
@ 20,21 SAY "Check "+LTRIM(STR(mcan,4))+" cannot be found."
WAIT SPACE(20)+"Press any key to continue"
```

Cancl.prg converts the *mcan* variable to a string to include it in the display. Because dBASE III PLUS pads a numeric integer with blanks when it converts the number to a string, the program uses the LTRIM() function to trim these leading blanks.

The second and fourth conditions are also handled with one IF clause. If there is a check number, then it must either be canceled or not canceled. Cancl.prg uses a logical field in Checks.dbf, Can, to govern this and an IF construction nested within the first IF construction to test for the condition. If the check has already been canceled, Cancl.prg displays a message:

```
IF Can
   @ 18,19 SAY "This check is already canceled"
   WAIT SPACE(22)+"Press any key to continue"
```

The final condition is whether the check number, which has been found and which is not canceled, is indeed the one that the user wants. The program handles this situation with a DO WHILE loop to accept only a yes or no answer from the user:

```
answer = " "
DO WHILE .NOT. answer$"YyNn"
   answer = " "
   @ 18,23 SAY "Is this the right check? (Y/N)" GET answer
   READ
ENDDO
```

After Cancl.prg has evaluated all the above conditions, if the uncanceled check is the correct one and the user presses **y** or **Y**, the program changes the logical field Can by REPLACEing .F. with .T. and displays a message. If the user types **n** or **N**, the program merely displays an appropriate message:

```
IF UPPER(answer)="Y"
   REPLACE Can WITH .T.
   @ 20,34 SAY "CANCELED"
ELSE
   @ 20,32 SAY "NOT CANCELED"
ENDIF
```

The Cleanup Section

The ELSE section of the IF FOUND() construction actually handles two different situations: it acts if no check number is found, and it clears the screen display if a check number is found and canceled. Because the screen doesn't change if the check number isn't found, the command lines to clear part of the screen seem inoperative. However, if a previous check was displayed, it is erased from the screen so that the user is not confused. These lines are:

```
@  4,71 SAY SPACE(4)
@  6,68 SAY SPACE(8)
@  8,25 SAY SPACE(30)
@  8,66 SAY SPACE(10)
@ 14,10 SAY SPACE(25)
```

Cancl.prg uses a little trick. The lines always execute, but for unfound checks, there's no screen display anyway. So, there's no visible change on the screen. For found checks, the lines clear the screen display of the check information without clearing the check border display. Moreover, because all messages appear on row 20, the command

```
@ 20,0 CLEAR
```

erases any message. With a thoughtful design, you can eliminate a great deal of unnecessary code.

At this point, the DO WHILE .T. loop repeats the entire procedure and asks the user for another check number until the user types **0**. As a convenience, Cancl.prg then shows the user a list of the currently uncanceled checks:

```
* display to the screen a list of remaining uncleared checks
CLEAR
? SPACE(17) + "LIST OF REMAINING UNCLEARED CHECKS"
?
? "CHKNO              PAID TO              AMOUNT     DATE        MEMO"
? LEFT(shoriz,5)+" "+LEFT(shoriz,30)+" "+ LEFT(shoriz,10)+" "+;
      LEFT(shoriz,8)+" "+LEFT(shoriz,14)
DISPLAY OFF ALL Chkno,Payto,Amt,Date,TRIM(Memo) FOR .NOT. Can
?
?
WAIT SPACE(13)+"Press any key to return to the Main menu "
```

Notice how Cancl.prg uses the LEFT() function to break the *shoriz* variable, which is a horizontal line in the Chkbook.mem file, to use as underscores for the various titles on the screen. This is a good example of how to save space by reusing the same display variable in several situations.

Finally, Cancl.prg closes the Checks.dbf and Chkno.ndx files, which are part of the databases, and then returns to the calling program:

```
CLOSE DATABASES
RETURN
```

Remember: open database files only as you need them, and close them as soon as you're finished. This cuts down on potential problems from power shortages, from rebooting, or from turning the computer off accidentally.

Summary

With the Cancl.prg module as an example, you have stepped through a typical dBASE program to see how the various components fit together to work as a whole program.

Chapter 15
TESTING AND DEBUGGING THE PROGRAM

Once you've completed your program, test it and correct the mistakes. Correcting mistakes is known as debugging the program. You test and debug programs simultaneously.

What This Chapter Covers

In this chapter you will learn:

- What the most frequent programming errors are and how to look for them
- The steps to testing and debugging a program
- The built-in dBASE debugging commands
- The two basic ways to debug programs during the run of a program: suspending the program and stepping through the program
- Other debugging techniques that you can use during program development

Preparing for This Chapter

Have a thorough understanding of dBASE programming before doing this chapter.

What to Look for

There are several types of programming mistakes: (1) misspelling commands, (2) forgetting to separate commands from their expressions with at least one space, (3) syntax errors, that is, using a command incorrectly, (4) issuing an incomplete command, (5) run time errors.

If you're good at spelling, spelling errors aren't difficult to catch. Printing your program files and proofreading them for spelling mistakes is the most effective way to catch them.

dBASE III PLUS groups errors in spacing and incomplete or incorrectly used commands under syntax errors. These errors are more difficult to pinpoint, but here are some general clues to guide you:

- Check the proper spacing between commands. Remember that every command must be separated by at least one space from the other information on the command line. You can catch this type of error when you proofread your files.
- Make sure that you haven't misspelled memory variable names, file names, and field names.
- Check to see that all PUBLIC variables are declared PUBLIC before you use them, and that you never declare a variable PUBLIC more than once.
- Watch out for data type mismatches. For example, don't attempt to SAY a date field and a string field with the same @ command. Use string conversions to correct this type of mistake. A data type mismatch also occurs when you attempt to perform a numeric operation, such as SQRT(), on a non-numeric field or memory variable.
- Check and double check the order of the arguments in a command line. For example, this command line is incorrect:

```
@ 10,10 GET mname SAY "Enter client's name:"
```

  The correct form is:

```
@ 10,10 SAY "Enter client's name:" GET mname
```

- Be careful about providing the correct options in commands when you use them. For example, this is incorrect:

```
DO FOR Name = "Smith"
```

  The correct form is:

```
DO WHILE Name = "Smith"
```

Run time errors are by far the most difficult bugs to find. They're called run time errors because they don't appear until you actually run the program. Most run time errors are mistakes in logic. You may think your code is correct, but the program doesn't do what you expect. Either you're not writing the commands as dBASE expects them, or your logic is different.

One of the most frequent types of mistakes in logic occurs when you've misused the logical operators themselves. It's important to remember the distinction between .AND. and .OR. Keep in mind that logical .AND. means that both the first condition and the second condition must be true. Logical .OR. means that either one or the other condition must be true.

When using these operators, it may help you to think graphically and to draw circles to illustrate how the conditions evaluate. For example, to show in a diagram what this line means:

```
DISPLAY FOR State = 'CA' .AND. Zip < 91400
```

the intersection of the circles, that is, the shaded area, illustrates where both conditions are met:

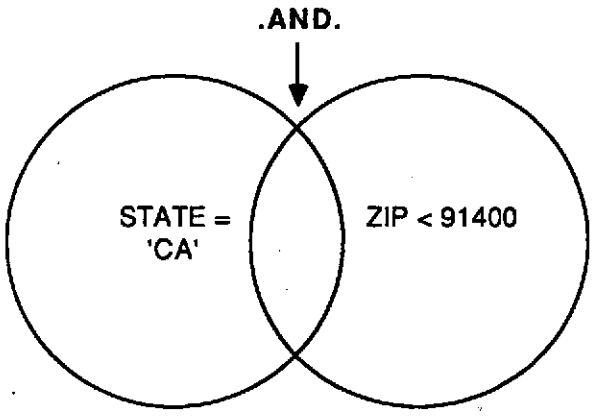


Figure 15-1 How logical .AND. works

However, this command

```
DISPLAY FOR State = 'CA' .OR. Zip < 80000
```

would look different in a graphic representation:

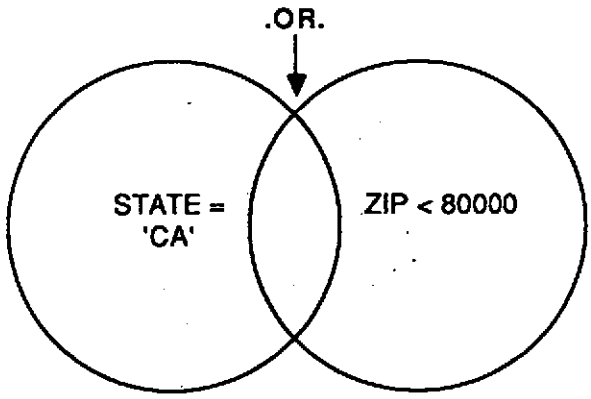


Figure 15-2 How logical .OR. works

The shaded area indicates where either one or the other condition is met.

Another logic mistake occurs when you misunderstand how to use logical .T. and .F. in situations calling for reverse logic. For example, this line:

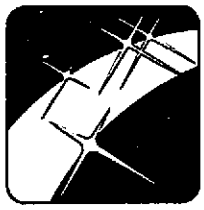
```
DO WHILE .NOT. EOF()
```

means: do while the end-of-file condition is false, that is, not true. Keep track of trues and falses. Check *Using dBASE III PLUS* for more help with logical operators.

Steps to Testing and Debugging

In Chapter 1, you learned that the modular approach to programming is preferred. If you get used to writing modular programs, you'll discover that they have another important advantage. You can test and debug modular files as you write them. Because these modules are as small as possible, and generally do one specific task, you can test and debug them quickly. dBASE III PLUS gives you a variety of commands that facilitate testing and debugging. Before you investigate these commands, get to know the basic steps to testing and debugging your programs:

1. Write each module, document it as you write it, and test it as soon as you've finished it.
2. Use the built-in debugging features with other techniques to debug your program modules as you go.
3. When you have tested a module, and it appears to work correctly, go to the next module.
4. After you've completed the necessary modules that go together, combine them into a composite program. Run and test this new, composite program, using the same features.
5. Continue to build your programs in larger units, gradually adding new, thoroughly tested modules.
6. When the entire project is completely assembled, do thorough testing of it, too.
7. Give your program to others for their independent testing. This is called alpha testing. It occurs before you actually begin using the program. Correct any mistakes, again using the built-in dBASE features.
8. When the program passes the alpha test stage to your satisfaction, let a limited number of users test it. This is the beta test stage. Make sure that those users running the program know how to document any problems that they discover during the testing.



9. After the program has passed the beta test stage, it's ready to be used by others. However, never consider a program bug-free. Testing and debugging never really end. A good program may not exhibit bugs for months, even years, after it's in general use. With good documentation, you will be able to find and fix even the most elusive bugs.

These steps mirror the top-down approach of the design stage. When you have finished your modules, assemble them in larger and larger units, starting from the bottom and working your way up to the top, until finally you have the finished application program.

> **TIP**
> Before attempting to correct a bug in a program that's in the alpha and beta test stages, make a copy of the program file, in case your newly debugged version doesn't work. You can then go back to the original if your attempt at debugging takes you too far astray. Occasionally, you may find that the best way to debug a program module is to rewrite it entirely.

The Debugging Commands

dBASE III PLUS has commands that allow you to test and debug programs as you run them. They are, in alphabetical order, RESUME, SET DEBUG, SET DOHISTORY, SET ECHO, SET HISTORY, SET STEP, and SUSPEND. How you use them depends on the way you want to debug your programs.

Using the History Buffer

One debugging method is to check the most recent commands that you issued. dBASE III PLUS automatically keeps track of the last 20 commands that you enter at the dot prompt. This is known as the history buffer. At any time, you can use the ↑ key to see the commands currently in the buffer to determine what led up to a mistake.

Using the history buffer also lets you repeat a command without retyping it. Just press the ↑ key to move to the command you want, and press ↵ to reissue the command.

You can set the number of command lines stored in the buffer to be more or less than 20 with the SET HISTORY TO command. For example,

```
SET HISTORY TO 30
```

instructs dBASE III PLUS to retain the last 30 commands in the buffer . You can also use DISPLAY HISTORY and LIST HISTORY to view the contents of the buffer. DISPLAY HISTORY pauses the screen, whereas LIST HISTORY does not.

Normally, dBASE does not record commands from programs in the history buffer because it is slow and there is usually no need to do this. However, sometimes a particular program has a problem and you can't figure out what's going on. If you SET DOHISTORY ON, that tells dBASE to record in the buffer the commands in the program in the order of their execution. You may discover by doing this that the statements are not executing as you expected. If you want dBASE III PLUS to retain more than 20 lines of code in the buffer, SET HISTORY TO another integer number at this point.

Then, when you test your program, if dBASE III PLUS finds a command that is incorrect, it stops execution of the program and gives you an error message, such as **Syntax error** or **Data type mismatch**. However, it also presents you with a choice about what to do next:

Cancel, Ignore, or Suspend? (C, I, or S)

When debugging programs, you would type S to SUSPEND the program. dBASE returns to the dot prompt, but leaves the program's working environment, including memory variables, intact in active memory, and keeps all database files open. To see what command line in the program caused the program to stop, you would type DISPLAY HISTORY at the dot prompt.

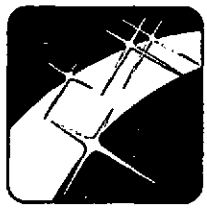
dBASE shows you the contents of the history buffer and allows you to edit the incorrect line. You would resume running the program by typing **RESUME** at the dot prompt.

This type of interactive debugging is an effective way to catch program errors and make corrections during the run of the program.

> **NOTE**
> When correcting errors with SUSPEND and RESUME, the changes don't alter the original program file. Therefore, it's important to keep track of your changes. Later, use MODIFY COMMAND to make your changes permanent in the program file.

If you choose to CANCEL the entire run, dBASE III PLUS returns to the dot prompt and clears memory. See below for more information about the CANCEL command. If you choose to IGNORE the error, dBASE III PLUS continues to run the program.

The history buffer only records the commands that were executed during the program run. For example, if you have a series of CASE lines, the CASE statements are executed in order until one evaluates as true. Then, the commands associated with that CASE statement are executed and the ENDCASE is executed. If there is a problem, dBASE shows you the commands in that CASE statement. Here is an example. Something has gone wrong when you press **C**, and the program has stopped. When you DISPLAY HISTORY, dBASE III PLUS shows you these lines:



```
DO WHILE .T.
CLEAR
@ 10,10 SAY 'What is your choice?' GET choice PICTURE '!'
READ
IF choice $ ('ABCD')
   DO CASE
      CASE choice = 'A'
         * There are commands here, but they don't pertain
         * to the problem, so they don't appear in
         * the history buffer
      CASE choice = 'B'
         * Same for this choice
      CASE choice = 'C'
         * This was the problem line:
         DO Cancel
```

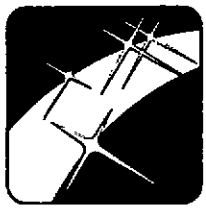
dBASE III PLUS shows you the command lines executed prior to the DO CASE statement and the command line for the C choice, but not the command lines for the other CASE statements.

Stepping Through the Program

The other way to debug programs allows you to see each program command line on the screen or the printer as you run your program. These commands are helpful if you can't isolate the suspected bug within the 20 or so lines of the history buffer. For example, if you have a lengthy IF...ENDIF construction, you might not find the mistake in the buffer.

With SET ECHO ON, dBASE III PLUS shows each program line on the screen. The program continues to run unless a major error occurs, so you can check what command does what operation. Using SET STEP ON, however, steps you through the program one command at a time. The program stops after each command and only continues when you tell it.

With SET ECHO ON, both the commands in the program and the program's own screen displays appear on the screen. The two will probably get in the way of each other. If you use SET DEBUG ON with SET ECHO ON, dBASE III PLUS routes the listings of commands to the printer rather than to the screen.



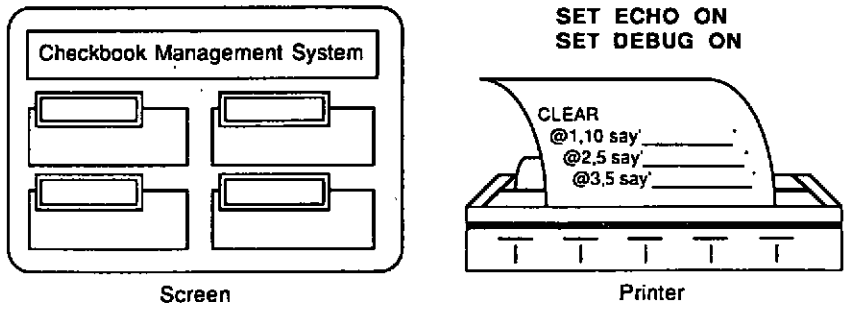


Figure 15-3 Using SET ECHO ON with SET DEBUG ON

So that the screen displays of the program are not disrupted, the actual command lines are routed to the printer.

So, to step through the program as it's running, first SET STEP ON and SET ECHO ON immediately before running the program. Use SET DEBUG ON if you want a printout of the debugging session. dBASE III PLUS shows each command line on the screen. Then it gives you this choice:

Press SPACE to step, S to suspend, or Esc to cancel...

If you wish to SUSPEND the program and check the history buffer, make sure that you also SET DOHISTORY ON before running the program. The normal way to use the SET STEP ON/SET ECHO ON procedure is to press the **Spacebar** to step through each command. When you get to the point where a problem occurs, you can examine your code with SUSPEND and DISPLAY HISTORY, make any changes and try continuing the program run, or cancel the entire program by pressing the **Esc** key.

TIP
During the initial stages of testing and debugging, use the DOHISTORY/SUSPEND/RESUME procedure for debugging. If you still have problems finding the mistakes, use the SET ECHO/SET STEP/SET DEBUG commands.

Other Debugging Tricks

Sometimes, even with the above debugging aids, you'll find that you just can't isolate a mistake. In this case, you could try some other debugging techniques.

Using the Esc Key

During program testing and debugging leave SET ESCAPE ON, the default. Then, with SET DOHISTORY ON, you can interrupt the program yourself by pressing the **Esc** key, and dBASE III PLUS will give you the same choices:

Cancel, Ignore, or Suspend? (C, I, or S)

The advantage to this approach is that you can check program code during the run, even though the code doesn't necessarily cause dBASE III PLUS to interrupt the program. Later, you can SET ESCAPE OFF so that the user can't stop the program with the **Esc** key.

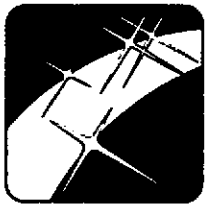
You can also use the command ON ESCAPE SUSPEND, which bypasses the above message and automatically suspends program execution when you press the **Esc** key.

DISPLAY MEMORY and DISPLAY STATUS

Many times a problem occurs because the memory variable is incorrect, nonexistent, or of the wrong type. To determine if this is so, insert the SUSPEND command into your program at the suspected trouble spot. When the program SUSPENDs, you would type DISPLAY MEMORY to view the variables. You could continue the program by typing RESUME, or you could type CANCEL and correct your errors.

This is a good way to work your way through a complex program file. When you've isolated and corrected your mistake, make sure that you delete the SUSPEND command from your program.

Because the DISPLAY STATUS command shows you the working environment, use it for debugging by substituting DISPLAY STATUS for DISPLAY MEMORY in the above instructions. If you're working with a database file, use one of the DISPLAY or LIST options to pinpoint a problem.



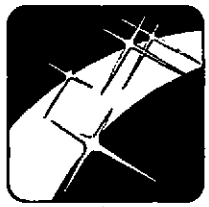
NOTE
The CANCEL command is abrupt. It closes all program files and RELEASEs all variables from active memory. This can hinder your efforts to find the problem in your program. For this reason, use SUSPEND and DISPLAY MEMORY. Neither CANCEL nor SUSPEND closes database files in USE, so be sure to CLOSE DATABASES either before or after you CANCEL your program.

Keeping a Record of What's Happening

Another useful trick is to use the SET ALTERNATE command to save what's happening on the screen in a disk file. SET ALTERNATE TO <filename> sets up the file in which this record will be kept. dBASE III PLUS appends the file extension .txt to the filename unless you specify another extension. The SET ALTERNATE ON command starts the recording by opening the file. SET ALTERNATE TO without a filename closes the file and stops recording.

The best way to use this trick is with the DISPLAY MEMORY and DISPLAY STATUS commands at those points in your program where you've discovered bugs. For instance:

```
* Set up file Problems.txt to record
* what's happening on the screen
SET ALTERNATE TO Problems
* Open file to receive screen output
SET ALTERNATE ON
DISPLAY MEMORY
DISPLAY STATUS
* Close file
SET ALTERNATE TO
WAIT
CANCEL
```



You can refer to this file later to check what was in the active memory. Use the TYPE command to view the file, or MODIFY COMMAND with the filename, including the .txt extension. In a similar fashion, you can get a printed list like this:

```
DISPLAY MEMORY TO PRINT
WAIT
CANCEL
```

Make sure the printer is on before you run the program. If you use the DOHISTORY technique, you can also SET PRINT ON immediately after the program run is SUSPENDed and before you LIST HISTORY to get a snapshot record of the debugging step and what's going on in the computer at that point.

The ON ERROR Command

You can also set up the ON ERROR DO <program filename> command to alert you when an error occurs by branching to a subprogram specifically designed to handle such errors. This command stays in effect throughout the program run, unless you turn it off with ON ERROR. The following module SETs DOHISTORY ON. If an error occurs, it branches to Error.prg, listed below, which gives you a message and automatically DISPLAYs HISTORY. Use this setup before you begin testing a program:

```
SET DOHISTORY ON
ON ERROR DO Error
```

Error.prg contains these lines:

```
* ERROR.PRG - tells you that an error occurs and
*              shows you the program code responsible
CLEAR
@ 10,10 SAY "An error has occurred!"
DISPLAY HISTORY
WAIT
RETURN
```

See also the discussion of the new ERROR() and MESSAGE() functions, and the RETRY command in the next chapter.

TIP
Whatever debugging techniques and commands you use, make sure that you turn these commands off, or delete the debugging commands from your program files when you're finished testing and debugging your programs.

Chapter 16
MORE ADVANCED FEATURES

What This Chapter Covers

This chapter discusses some of the more advanced features of dBASE III PLUS and serves as a bridge to your own independent study of dBASE programming. You will learn:

- A shortcut for IF...ENDIF constructions: the IIF() function
- What PROCEDURE files are and how to use them
- How to hide a PUBLIC memory variable temporarily within a subprogram
- How you can use parameter passing to make generic program modules that work in any situation
- How to retry commands after rectifying an error condition
- How to use assembly language routines in dBASE programs
- Ways to work with the operating system and exchange data between dBASE III PLUS and other applications
- How to set up a turnkey system

dBASE has programming features that speed up your programs and make them more efficient. These features allow your programs to deal directly with the operating system and with other application programs, such as word processors or spreadsheets.

Preparing for This Chapter

Besides having a general knowledge of dBASE programming, be familiar with the commands and terminology of your computer's operating system.

A Shortcut for IF...ENDIF

There is a shortcut to the IF...ENDIF construction: IIF(), the immediate IF function. It lets you set up the entire IF condition and resulting action in one line. You don't need the ELSE or ENDIF lines. The IIF() function requires a slightly different syntax. List the condition and the results in parentheses following the IIF. Separate the condition and results with commas.

Here's an example. Your birthday is April 26th, so you're testing whether the current date is your birthday. The program is to give you an appropriate message. This example uses the memory variable *mday* to hold the message:

```
IF DATE() = CTOD('04/26/86')
   mday = 'Happy Birthday!'
ELSE
   mday = 'Just another day'
ENDIF
? mday
```

This can be rewritten with the IIF() function as:

```
* If today's date is April 26, 1986, store the first
* message to the variable mday; if not, store the
* second message to mday
mday = IIF(DATE() = CTOD('04/26/86'), 'Happy Birthday!',;
   'Just another day')
* Show the message
? mday
```

Note that you first establish what variable will contain the result of the IIF() function's evaluation, *mday* =. The first expression in the IIF function is the condition to be tested. The second expression is what results if the condition is true, and the third expression is the result if the condition is false. Both expressions must be of the same type.

IIF() is a way to streamline your program coding, which in turn makes your program run faster.

More Efficient Program Code

Aside from the standard dBASE control structures, DO WHILE...ENDDO, IF...ENDIF, IIF(), and DO CASE...ENDCASE, dBASE has other commands and features that help you write more efficient code. These commands help you with program flow and handling memory variables.

Procedures

When you are designing and developing your program, you'll probably use certain key modules repeatedly. Each time you want to run a module, you use the DO command to call it. Because the module is in a separate program file, dBASE III PLUS has to go to the disk and open the file.

Using a procedure file reduces disk access time and delay. A procedure file contains modules of program code that stay in the computer's memory for the duration of your program. dBASE III PLUS doesn't have to access the disk each time the program needs the program module.

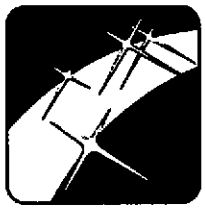
A procedure file contains only module PROCEDUREs. It has the same .prg extension as any normal dBASE program file. You create and edit a procedure file using MODIFY COMMAND. Within the file, each module PROCEDURE must begin with a PROCEDURE line, which includes the name of the PROCEDURE. Next come the actual commands. The last line of each PROCEDURE, as with all module programs, should contain a RETURN statement.

For example, the checkbook management system uses a procedure file, Rprtpro.prg, for all reports. The PROCEDURE called Printer handles whether output shows on the screen or is printed. This PROCEDURE begins like this:

```
PROCEDURE Printer
```

Each separate report is in a different PROCEDURE in the Rprtpro.prg file.

TIP
Write and test your modular PROCEDUREs separately. Then combine them in a procedure file later. To do this, issue MODIFY COMMAND and name the new procedure file. Press **Ctrl**-K R to read each module file into the procedure file. dBASE III PLUS prompts you for the name of each file to be read in.



If you plan to use a procedure file during the entire program, include the SET PROCEDURE TO <filename> command to open a procedure file at the beginning of the program.

The checkbook management system only uses a procedure file for reports. When the user types **H** from the main menu and program control branches to Reports.prg, the first line in this program opens the procedure file:

```
SET PROCEDURE TO Rprtpro
```

After you have opened a procedure file with SET PROCEDURE, use DO to run any PROCEDURE in the procedure file. dBASE III PLUS first checks the PROCEDUREs in memory. If there are no PROCEDUREs with the filename, dBASE III PLUS then checks the disk for the file.

When it needs the PROCEDURE Reporta in the Rprtpro.prg file, the checkbook management system uses the command:

```
DO Reporta
```

You can have only one procedure file open at a time. The procedure file can have at most 32 PROCEDUREs. If you wish to open another procedure file, use SET PROCEDURE TO with the new filename. dBASE III PLUS automatically closes the previous procedure file. To close a procedure file without opening another, type either CLOSE PROCEDURE or SET PROCEDURE TO, without a filename.

NOTE
During debugging, if you attempt to edit an open procedure file, dBASE III PLUS will tell you that the file is currently open. Type CLOSE PROCEDURE first.

Using procedure files, you can effectively expand the power of dBASE III PLUS by providing your own routines. Because you can use procedures over and over, they are almost like new dBASE III PLUS functions.

Another excellent use of procedure files is for help screens. Most of the time, the program doesn't need these procedure files until the user wishes additional help. You can set up your help screens in a separate procedure file and SET the PROCEDURE file when necessary. Once the entire procedure file is in memory, the help screens appear very quickly on the screen.

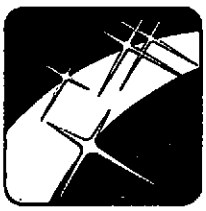
Hiding a Public Variable

A PUBLIC memory variable remains in memory while the program in which it is declared PUBLIC and all subprogram modules called by that program are in effect. However, you may at times wish to use a memory variable name in a subprogram but not to change its PUBLIC value in relation to the program as a whole. You can temporarily hide the PUBLIC variable.

A practical use of this feature is to make the names of memory variables consistent and your programs more understandable. For example, you may want to use the *balance* variable, which is PUBLIC, in a subroutine for another balance amount.

To hide a PUBLIC variable in a subprogram, write the command PRIVATE and the memory variable name at the beginning of the subprogram:

```
PRIVATE balance
```



The PRIVATE command hides the PUBLIC variable and declares a new PRIVATE variable with the same name. You can use the PRIVATE variable in this subprogram and all other programs that it calls. When the subprogram is finished and program flow RETURNs to the main program, the PUBLIC variable automatically loses its hidden status, and still has its previous value. Here is an illustration of how hiding a PUBLIC variable works:

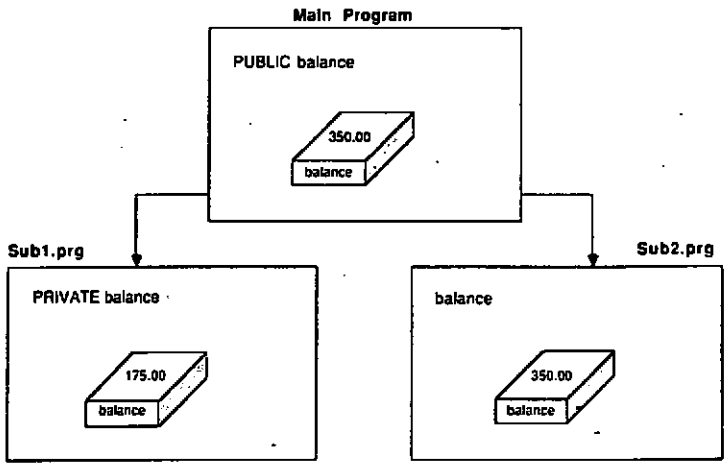


Figure 16-1 Hiding a PUBLIC variable

The *balance* variable, declared PUBLIC in the main program, contains the value 350.00 before program control branches to Sub1.prg. There, the PUBLIC version is hidden and cannot be accessed or changed; a new PRIVATE *balance* is declared. The subprogram considers *balance* as PRIVATE to it only. At the end of this module, *balance* contains the value 175.00. However, when program control RETURNs to the main program, the PUBLIC variable *balance* is again accessible and contains 350.00.

Parameter Passing

Module program design allows your program to use modules over and over, either as separate programs or combined in a procedure file. You can also reuse the same module, with a few changes for similar situations.

For example, you want to draw a box on the screen, but your program occasionally needs the same box at different locations, or with a different size. The program code for drawing the box would be the same, but the size of the box, its parameters, would be different. It wastes disk space to write another module to draw a different-sized box.

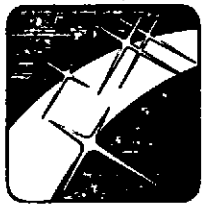
You can set up a generic box-drawing program that depends on the specific parameters it gets from the calling program. For example, if the calling program needs a box that is to be displayed at the very top of the screen and is five rows deep, it gives the correct coordinates to the box-drawing module. Later, when the calling program needs a box at the bottom of the screen and ten rows deep, it presents a different set of parameters to the same module program. This is known as parameter passing.

The beginning command line of the module must contain the names of the PARAMETERS that are being passed to it from the calling program. You must separate two or more PARAMETERS with commas. Here is a program module that uses PARAMETERS to draw any size double-lined box anywhere on the screen:

```
* BOX.PRG - draws a box given the passed parameters
PARAMETERS beginrow, begincol, endrow, endcol
CLEAR
@ beginrow,begincol TO endrow,endcol DOUBLE
RETURN
```

The four PARAMETERS are *beginrow*, *begincol*, *endrow*, and *endcol*.

WARNING
You'll get an error message if you add superfluous spaces after the list of memory variables in a PARAMETERS statement.



The calling program must initialize the passed variables and provide their values. It also must tell the module what PARAMETERS to use in the same order as they're listed in the PARAMETERS statement. It uses the DO...WITH command for this; that is, DO the file WITH the following PARAMETERS. So, to draw a box from row 10, column 5 down to row 23, column 75, use the following commands in the calling program:

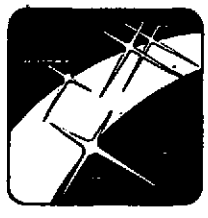
```
DO Box WITH 10,5,23,75
```

If a procedure file contains PARAMETERS, the PROCEDURE line comes before the PARAMETERS line. For example, when the user chooses **H** from the main menu, the program asks whether the report should be displayed on the screen or printed:

Do you want the output sent to the printer or the screen? (P/S)

The program code that handles display of this prompt is in a procedure file called Printer in the Rprtpro.prg file. The screen row of this prompt varies, so the Printer module uses PARAMETERS. Here is the entire Printer module:

```
PROCEDURE Printer
PARAMETERS row,pr
@ row,6 SAY "Do you want the output sent to the " +;
            "printer or the screen? (P/S)"
pr = " "
DO WHILE .NOT. pr$"PpSs"
   pr = " "
   @ row,70 GET pr
   READ
ENDDO
```



The two variables *row* and *pr* get their values from the calling program. The calling program passes the row and column coordinates as PARAMETERS to the module. Generally, the calling program supplies the same parameter names that are in the PARAMETERS line of the called module. You can use other variable names, but the order in which you pass them must correspond to their order in the PARAMETERS command. For example, the Report1 module in the procedure file Rprtpro.prg of the checkbook management system calls the Printer module like this:

```
* ask for output to printer or screen
DO Printer WITH newrow,pr
```

The Report1 module uses a different variable name, *newrow*, for the first parameter, but its value is still correctly passed to the *row* parameter in the Printer file.

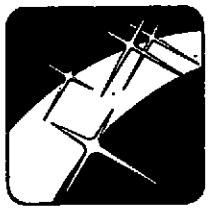
If the program using the PARAMETERS command changes the value of a parameter, this value is passed back to the calling program. Using parameter passing enables you to use your program modules in many different variations.

Potential Errors

Sometimes your program will encounter potential errors that can be corrected without ending the program prematurely. For example, the program can check to see if enough disk space is available before copying a file and, if there isn't, delete an unnecessary file.

The ERROR() function returns the integer error number for whatever dBASE III PLUS error occurred, and the MESSAGE() function returns the string error message for these messages. Your program can check these values and eliminate the error condition. A list of these error numbers and messages is in *Using dBASE III PLUS*.

Also be aware of the RETRY command, which is like RETURN except that it returns to the exact line of the calling program, instead of the next line. RETRY allows the program to reexecute the problem at the spot where an error originally occurred. Use RETRY in situations where you've made a change after an error and wish to RETRY the program run.



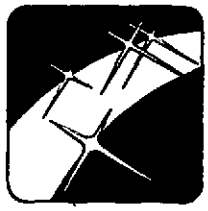
Using Assembly Language Routines

The dBASE III PLUS interpreter must interpret every command line before it can execute a program's command. This process is slower than if the commands were written in a language that the computer uses directly.

Because it works with the computer's microprocessor, assembly language is closer to the actual hardware of the computer than dBASE. Programs written in assembly language and assembled into binary code are executed much faster than interpreted dBASE commands. What is more, assembly language routines can control more of the hardware of your computer, such as aspects of the cursor that dBASE doesn't control.

> **WARNING**
> This is a very advanced technique. Be very familiar with assembly language before attempting these suggestions. If you are an experienced assembly language programmer, check the specifications under LOAD in the Commands and Functions section of *Using dBASE III PLUS* for dealing with addresses, memory, and segments.

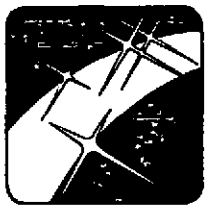
dBASE allows you to use separate modules written in assembly language within dBASE programs. These modules must first be assembled and linked, two processes that transform them into machine language, and finally converted into binary form. You must have an assembler and linker program to work with assembly language routines, as well as DOS's Exe2bin.com file.



The procedure for using assembly language subprograms is to LOAD the programs and then CALL them. The LOAD command places the assembly language routine in memory. It assumes that the assembled program has the file extension .bin, which is produced when you use the DOS EXE2BIN command. The routine is not treated as an external program on the disk but is ready at any time, like a procedure file. The CALL command executes the routine. You can have up to sixteen different assembly language routines in memory at one time, and each can be as large as 32,000 bytes. To remove a LOADed program from memory, use the command RELEASE MODULE with the correct filename. Check the commands and functions of *Using dBASE III PLUS* for more restrictions when working with assembly language programs and for an example of how to write, assemble, link, and convert to binary form an assembly language routine.

If you want to see an assembly language routine in action, run the Cbmenu2.prg. This is the same main program as Cbmenu.prg, with the addition of two assembly language routines to control the cursor. You have probably noticed that the INKEY() function makes the cursor jump around a bit on the screen. The two assembly routines included on the disk, Cursoff.bin and Curson.bin, control this. Cbmenu2.prg first loads the two assembly routines as part of the setup stage:

```
* Load two binary files to turn cursor on and off
LOAD Curson
LOAD Cursoff
```



It then CALLs these routines during the DO WHILE loop to accept the user's input:

```
DO WHILE .T.
   i=0
   DO WHILE i=0
      i=INKEY()
      * Turn off cursor
      CALL Cursoff
      @ 17,63 SAY TIME()
      @ 22,58 SAY ""
      * Turn on cursor
      CALL Curson
      IF UPPER(CHR(i))$"ABCDEFGHIJKLX"
         EXIT
      ENDIF
      i=0
   ENDDO
   * (...)
ENDDO WHILE .T.
```

The screen display looks much more regular and less distracting when you run this version of the main menu program. Note that when the user wishes to quit the program, the X choice, the program RELEASEs the two MODULE routines.

dBASE Programs in a Larger Context

In Chapter 13 you learned how to work with disk space and file management from within your programs. At times you'll need to relate your dBASE III PLUS applications to the operating system, MS-DOS™ or PC-DOS™, and to other applications programs, such as word processors.

The RUN Command

You can run other programs from within dBASE III PLUS. These programs can be resident operating system commands, such as COPY or DIR, transient operating system programs, such as FORMAT or CHKDSK, batch files with the .bat extension, or other application programs.

Use dBASE III PLUS's RUN command for running these external programs. You can use ! instead of RUN if you wish.

However, make sure that your computer has enough memory and that you have correctly set up the Config.db file. For more information about Config.db, see *Using dBASE III PLUS*.

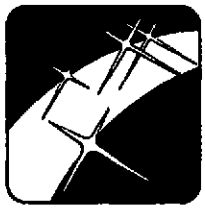
One of the common uses for the RUN command is when you want to reset the system date or time. dBASE III PLUS does not have commands to do this. You can STORE the new date or time in a memory variable and then substitute the variable for the date when RUNning the DOS DATE command. The variable must be a string, but you can first use a dBASE III PLUS date variable to test for correct input:

```
* Initialize date variable
today = CTOD('  /  /  ')
* Get new date with correct error checking
@ 10,10 SAY 'What is the correct date?' GET today
READ
* Change to string
today = DTOC(today)
* Change current date
RUN DATE &today
```

To change the system time, make sure that you ask for the time from the user in the correct form — hours:minutes. When RUNning batch files, for instance, the batch file Files.bat, use the filename without the extension.

```
! Files
```

NOTE
The operating system's command processor program, Command.com, must be in the root directory of the boot drive. If you're using a dual-floppy system, have these files on dBASE III PLUS System Disk #2. See the RUN command in the Commands and Functions section of *Using dBASE III PLUS* for more information.



The Operating System and dBASE Environment

When you write programs for commercial release, you may have to determine under what operating system dBASE III PLUS is running, certain specifications of the operating system environment, or what version of dBASE III PLUS is being used. There are three functions that give you this information.

If your program has to check the operating system under which it's running, for instance PC-DOS or UNIX™, use the OS() function. This function returns a string value of the operating system. It is useful if your program coding, such as memory variables, depends on which operating system is running:

```
STORE OS() TO opsys
* If the operating system is UNIX
IF SUBSTR(opsys,1,4) = 'UNIX'
   * Run special setup program for UNIX
   DO Setunix
ENDIF
```

If your program has to check one of the operating system's environments, such as the current PATH, use the GETENV() function. For example, your program needs the current setting for DOS's COMSPEC:

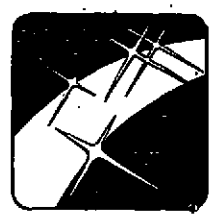
```
STORE GETENV("COMSPEC") TO environ
```

The environment name itself is a string enclosed in delimiters. This function works only for those commands issued at the operating system level, such as PATH or SET COMSPEC. When determining the current path, GETENV() does not work after you use the SET PATH command from within dBASE III PLUS.

Finally, the VERSION() function returns the string value of the current version of dBASE III PLUS. This function, like OS(), does not require any arguments.

Other Applications Programs

Your program may have to bring in data from other applications programs, or send out data to other programs. dBASE III PLUS now allows you to bring in and send out information from and to pfs:File with the IMPORT and EXPORT commands. These commands expand on the abilities of the APPEND FROM and COPY TO commands.



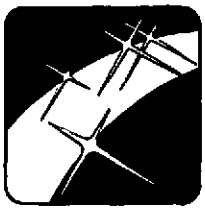
All microcomputer applications, including dBASE III PLUS, handle their files in different ways. Usually they supply information that is necessary to keep track of the data in the files. For example, dBASE III PLUS uses a header which contains specifics about the number of fields in a database file, their type and length, and other facts. This header would be of no use to another program such as WordStar. Similarly, WordStar's formatting specifications would be meaningless to dBASE III PLUS.

When you import or export data from files created by other programs, you must strip out this superfluous information and leave just the raw data. dBASE III PLUS calls these files system data files. Usually, the data is delimited with commas or spaces. When COPYing data to other programs, use the DELIMITED WITH and SDF options. When bringing in data from other programs, you must first use the other program to put the data into a format acceptable to dBASE III PLUS. Then use the APPEND FROM command with the DELIMITED WITH and SDF options. These are explained in *Using dBASE III PLUS*.

> **NOTE**
> APPEND FROM and COPY TO can now read and write Lotus 1-2-3™, VisiCalc™, and Multiplan™ files directly.

A Turnkey System

Learning how to use dBASE III PLUS in its larger context, especially with the operating system, can be useful when you wish to set up a turnkey environment. This is an application that literally runs itself as soon as the user inserts the dBASE III PLUS program disk and turns on the computer. It's like turning a key to start a car. A turnkey system makes use of a special DOS batch file and the ability of dBASE III PLUS to begin running a program when you start dBASE III PLUS.



As an example, you can set up the checkbook management system to begin as soon as you start the computer. First, set up the special automatically loading DOS batch file, Autoexec.bat. Use your word processor or dBASE III PLUS's MODIFY COMMAND to create this batch file. You can include the standard DATE and TIME commands and then the command to start dBASE III PLUS. Remember, each command must be on a separate line:

```
DATE
TIME
DBASE B:Cbmenu
```

In the Cbmenu.prg file, you also have to use the SET DEFAULT TO B command in the setup area if you have dual floppy disk drives. That way, the checkbook management system finds its database files. You can also SET the DEFAULT drive in the Config.db file by inserting the statement DEFAULT=B. See *Using dBASE III PLUS* for more information. If you delete Help.dbs from dBASE III PLUS System Disk #2, you may have enough room for your entire application program on the A drive. If you need more help with DOS batch files, look in the DOS manual under BATCH.

TIP
Many programmers change the name of the dBASE III PLUS program file from Dbase.com to Do.com. Then they can issue the command:

```
DO B:Cbmenu
```

from a batch file or the DOS prompt.

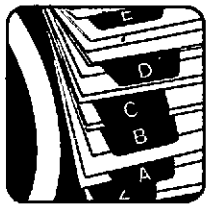
Where Do You Go From Here?

You have investigated the basics of dBASE programming using a typical dBASE applications program as an example of its power. You are now ready to begin writing your own applications in the dBASE language.

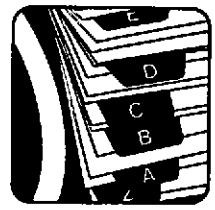
When you're more comfortable with dBASE, you may even want to develop applications that you can sell. You can encrypt your program code and protect against unlawful tampering or copying with RunTime+. It also condenses the size of your application by linking all the module programs in one unit, which makes your program run substantially faster. See the *RunTime+* section for more information.

There is much more to dBASE than what's covered in this book. You may wish to look at two other extremely helpful publications of Ashton-Tate: the *Advanced Programmer's Guide* and *TechNotes*, a journal of programming tips and useful sample programs. *TechNotes* is issued monthly by the Ashton-Tate Software Support Center. To purchase the *Advanced Programmer's Guide*, call Ashton Tate at (213) 329-8000. To subscribe to *TechNotes*, call the Ashton-Tate Subscription Service at (619) 747-1666.

Index

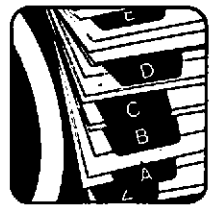


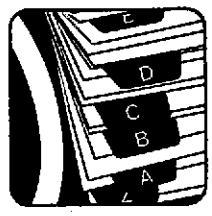
A

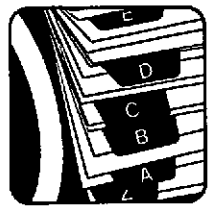


B

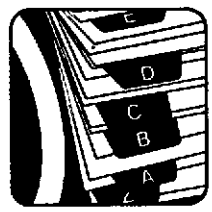
C

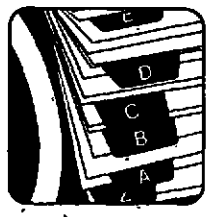






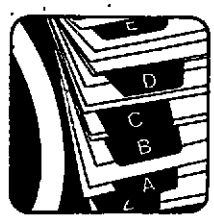
D

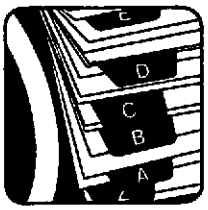




E

F

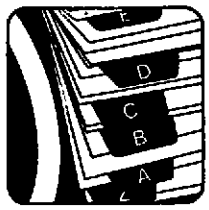




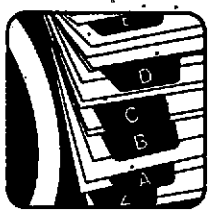
G

H

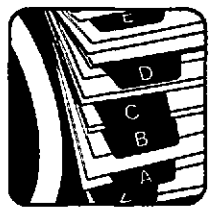
I



M

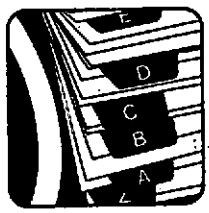


N



O

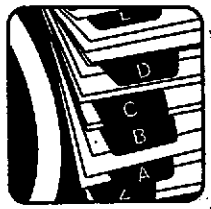
P



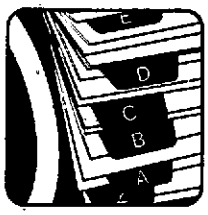
INDEX

Q

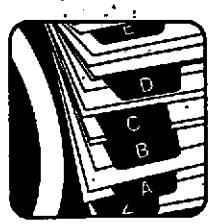
R



S



T



U