Section III:

If you understand how to write expressions, you are very close to being able to write programs.

There are four basic programming structures that you can use to get a computer to do what you want to do:

- **Sequence**
- **Choice/Decision**
- **Repetition**
- **Procedures**

You've already seen that dBASE II processes your commands sequentially in the order in which you give them. In this section we'll explain how you make choices (IF...ELSE), how you can make the computer repeat a sequence of commands (DO WHILE..), and how to use sub-files of commands (procedures).

Then we'll show you how to use these simple tools to write command files (programs) that will solve your applications problems.

Setting up a command file (writing your first program)

The commands we've introduced so far are powerful and can accomplish a great deal, yet only scratch the surface of the capabilities of dBASE II. The full power comes into play when you set up command files so that the commands you enter once can be repeated over and over.

When you create a command file you are programming the computer, but since dBASE II uses English-like commands, it's a lot simpler than it sounds. Also, because dBASE II is a relational database management system, you work with increments of data and information, rather than bits and bytes.

To set up a command file, you list the commands you want performed in a CP/M file with a <.CMD> extension to its name, using a text editor or word processor.

dBASE II starts at the top of the list and processes the commands one at a time until it is done with the list.

Other computer languages operate exactly the same way. In BASIC the sequence is very visible because each program line is numbered. In other languages (dBASE II among them), the sequence is implied and the computer will process the first line on the page, then the second line, etc. Some languages use separators (such as colons) between command statements; dBASE II simply uses the carriage return to terminate the command line.

The only time the sequence is not followed is when the computer is specifically told to go and do something else. Usually, this is based on some other conditions and the computer must make a decision based on expressions or conditions that you have set up in the command file. We'll tell you more about this later.

For now, let's create a command file called <Test>.

You can do this using a text editor or wordprocessor, but there's an easier way with dBASE II. Type:

^MODIFY COMMAND Test^

dBASE II now presents you with a blank screen that you can write into using the full screen editing features described earlier. Use them now to enter the short program at the top of the next page (do not type the "^" symbols).

The end of a line indicates the end of a command (unless you use a semicolon), so keep the list of commands as shown on the next page.

```
^USE Names^
^COPY Structure TO Temp FIELDS Name, ZipCode^
^USE Temp^
^APPEND FROM Names^
^COUNT FOR Name = 'G' TO G
^DISPLAY MEMORY^
^? 'We have just successfully completed our first command file.'
```

When you're finished, use clt-W (ctl-O with Superbrain) to get back to the dBASE II prompt. Now type:

```
A> ^dBASE Test^
```

If you typed the program in exactly the way it was printed, it crashed. Now type ^MODIFY COMMAND Test^ again and insert a semi-colon to correct the <Zip:Code> field name.

Once you get to writing larger command files of your own, you'll find that this built-in editor is one of the most convenient features of dBASE II, since you can write, correct and change programs without ever having to go back to the system level of the computer. Currently, this built-in editor can back up only about 5,000 lines, so editing should be planned in one direction for larger files.

The command file itself is trivial but does show you how you can perform a sequence of commands from a file with a single system command. This is similar to the way you use .COM files in your operating system.

If you are already in dBASE II (with the dot prompt), you type:

```
. ^DO <filename>^
```

where <filename> has the <.CMD> extension.

TIP: You may want to rename the main dBASE file to <DO.COM>, so that you can type ^DO <filename>^ whether you're in your system or in dBASE II. To do this with CP/M, type: A> ^REN DO.COM=dBASE.COM^

Making choices and decisions (IF..ELSE)

Choices and decisions are made in dBASE II with ^IF..ELSE..ENDIF^. This is used much as it is used in ordinary English: IF I'm hungry, I'll eat, (OR) ELSE I won't. With a computer, you use the identical construction, but do have to use exactly the words that it understands.

Simple decision: If only a single decision is to be made, you can drop the ELSE and use this form:

```
IF condition [.AND. cond2 .OR.  cond3 ....]
    do this command
    [cmd2]
    [....]
ENDIF
```

The "condition" can be a series of expressions (up to a maximum of 254 characters) that can be logically evaluated to being true or false. Use the logical operators to tie them together. Using our <MoneyOut> file, we might set up the following decision:

```
IF Job:Nmbr = '730' .AND.  Amount.  > 99.99;
       .OR.  Supplier = 'MAGIC TOUCH';
       .OR.  Bill:Date > '791231'
   do this command
   [cmd 2]
   [ ... ]
ENDIF
```

If all the conditions are met, the computer will perform the commands listed between the IF and the ENDIF (in sequence), then go on to the next statement following the ENDIF. If the conditions are not met, the computer skips to the first command following the ENDIF.

Two choices: If there are two alternate courses of action that depend on the condition(s), use the IF..ELSE statement this way:

```
IF condition(s)
   do command(s) 1
ELSE
   do command(s) 2
ENDIF
```

The computer does either the first set of commands or the second set of commands, then skips to the command following the ENDIF.

Multiple choice: Frequently, you have to make a choice from a list of alternatives. An example might be a the use of a screen menu to select one of several different procedures that you want to perform. In that case, you use the IF..ELSE..IF construction.

This is the same IF..ELSE that we've described, but you use it in several levels (called "nesting"), as shown below.

```
IF conditions 1
   do commands 1
ELSE
   IF conditions 2
      do commands 2
   ELSE
      IF conditions 3
         do commands 3
      ELSE
         .
         .
         .
      ENDIF 3
   ENDIF 2
ENDIF 1
```

This structure can be nested as shown as far as it has to be to choose the one set of commands required from the list of alternatives. It is used frequently in the working accounting system at the end of Part I.

Notice that each IF must have a corresponding ENDIF or your program will bomb.

TIP: dBASE II does not read the rest of the line after an ENDIF, so you can add in any identification you want to, as we did above. It helps keep things straight.

Repeating a process (DO WHILE..)

Repetition is one of the major advantages of a computer. It can continue with the same task over and over without getting bored or making mistakes because of the monotony. This is handled in most computer languages with the **DO WHILE** construction:

```
DO WHILE conditions
   do command(s)
ENDDO
```

While the conditions you specify are logically true, the commands listed will be performed.

Tip: Remember that these commands must change the conditions eventually, or the loop will continue forever.

When you know how many times you want the process repeated, you use the structure like this:

```
STORE 1 TO Index                 * Start counter at 1
DO WHILE Index < 11              * Process 10 records
   IF Item = ' '                 * If there is no data,
      SKIP                       * skip the record and
      LOOP                       * go back to the DO WHILE,
   ENDIF blank                   * without doing ProcessA
   DO ProcessA                   * Do file ProcessA.CMD
   STORE Index+1 TO Index        * Increase counter by 1
ENDDO ten times
```

In this example, if there is data in the <Item> field, the computer performs whatever instructions are in another command file called ProcessA.CMD, then returns to where it was in this command file. It increases the value of the variable Index by 1, then tests to see if this value is less than 11. If it is, the computer proceeds through the DO WHILE instructions again. When the counter passes 10, the computer skips the loop and performs the next instruction after the **ENDDO**.

The **LOOP** instruction is used to stop a sequence and cause the computer to go back to the start of a **DO WHILE** that contains the instruction.

In this case, if the Item field is blank, the record is not processed because the **LOOP** command moves the computer back to the **DO WHILE Index < 11**. The record with the blank is not counted, since we bypass the command line where we add 1 to the counter.

The problem with **LOOP** is that it short-circuits program flow, so that it's extremely difficult to follow program logic. The best solution is to avoid the **LOOP** instruction entirely.

Procedures (subsidiary command files)

The ability to create standard procedures in a language greatly simplifies programming of computers.

In BASIC, these procedures are called sub-routines. In Pascal and PL/I, they are called procedures. In dBASE II they are command files that can be called by a program that you write.

In our previous example, we called for a procedure when we said DO ProcessA. "ProcessA" is another command file (with a .CMD extension to its name). The contents of this command file might be:

```
IF Status = M
   DO PayMar
ELSE
   IF Status = S
      DO PaySingle
   ELSE
      IF Status = H
         DO PayHouse
      ENDIF
   ENDIF
ENDIF
RETURN
```

Once again, we can call out further procedures which can themselves call other files. Up to 16 command files may be open at a time, so if a file is in USE, up to 15 other files can be open. Some commands use additional files (REPORT, INSERT, COPY, SAVE, RESTORE and PACK use one additional file; SORT uses two additional files).

This is seldom a limitation, however, since any number of files can be used if they are closed and no more than 16 are open at any time.

A file is closed when the end of the file is reached, or when the ^RETURN^ command is issued by a command file. The RETURN command returns control to the command file that called it (or to the keyboard if the file was run directly).

The RETURN command is not always strictly necessary, as control returns to the calling file when the end of a file is encountered, but it is good programming practice to insert it at the end of all your command files.

Big tip: Notice that the command lines are indented in our examples. This is not necessary, but it increases command file clarity tremendously, especially when you have nested structures within other structures. Using all uppercase for the commands, and both upper- and lowercase for the variables helps, too.

Entering data interactively during a run (WAIT, INPUT, ACCEPT)

For many applications, the command files will have to get additional data from the operator, rather than just using what is in the databases.

You command files can be set up so that they prompt the operator with messages that indicate the kind of information that is needed. One good example is a menu of functions from which one is selected. Another use might be to help ensure that accounting data is entered correctly. The following commands can do this.

^WAIT [TO memory variable]^

halts command file processing and waits for a single character input from the keyboard with a WAITING prompt. Processing continues after any key is pressed (as with the dBASE II DISPLAY command).

If a variable is also specified, the input character is stored in it. If the input is a non-printable character (<enter>, control character, etc.), a blank is entered into the variable.

^INPUT ['prompt'] TO memory variable^

accepts any data type from the CRT terminal to a named memory variable, creating that variable if it did not exist.

If the optional prompting message (in single or double quotes, but both delimiters the same) is used, it appears on the user terminal followed by a colon showing where the data is to be typed in. The data type of the variable (character, numeric or logical) is determined by the type of data that is entered. Character strings must be entered in quotes or square brackets.

^ACCEPT ['prompt'] TO memory variable^

accepts character data without the need for delimiters. Very useful for long input strings.

Tips on which to use when:

WAIT can be used for rapid entry (reacts instantly to an input), but should not be used when a wrong entry can do serious damage to your database.

ACCEPT is useful for long strings of characters as it does not require quote marks. It should also be used for single character entry when the need to hit <enter> can improve data integrity.

INPUT accepts numeric and logical data as well as characters, can be used like ACCEPT.

Placing data and prompts exactly where you want them (@..SAY..GET)

The ^?^, ^ACCEPT^ and ^INPUT^ commands can all be used to place prompts to the operator on the screen.

Their common drawback for this purpose is that the prompts will appear just below the last line already on the screen. This works, but there's a better way.

If your terminal supports X-Y cursor positioning, another dBASE II command lets you position your prompts and get your data from any position you select on the screen:

```
^@ <coordinates> [SAY <'prompt'>]^
```

This will position the prompt (entered in quotes or square brackets) at the screen coordinates you specify. The coordinates are the row and column on the CRT, with 0,0 being the upper left-hand "home" position. If we specified "9,34" as the coordinates, our prompt would start on the 10th row in the 35th column.

Note: If you installed half intensity or reverse video, the prompt will be at half intensity or in reverse video. To disable this, re-do the installation procedure and use the "Modify/Change" option.

The SAY.. is optional because this command can also be used to erase any line (or portion of a line) on the screen. Bring dBASE II up and type:

```
^ERASE^
^@ 20,30 SAY 'What?'^
^@  5,67 SAY 'Here...'^
^@ 11,11 SAY "That's all."^
^@ 20, 0^
^@  5, 0^
^@ 11,16^
```

Instead of just showing a prompt, the command can be used to show the value of an expression with one or more variables. The form is:

```
^@ <coordinates>[SAY <expression>]
```

Type the following in dBASE II:

```
^USE Names^
^@ 13,9 SAY Zip:Code
^@ 13,6 SAY State
^SKIP 3^
^@ 23, 5 SAY Name + ', ' + Address
```

The command can be expanded further to show you the values of variables being used (memory variables or field names in a database) at whatever screen position you specify. (The variables used by both **GET** and **SAY** must exist before you call them out or you will get an error.)

^@ <coordinates>[SAY <expression>][GET <variable>]

To see how this works, type the following (do NOT QUIT dBASE when you're done--there's more to come):

```
^ERASE^
^USE Names^
^@ 15, 5 SAY 'State' GET State
^@ 10,17 GET Zip:Code
^@  5, 0 SAY 'Name' GET Name
     (Stay in dBASE)
```

This sequence has positioned the values of the variables (with and without prompts) at various places on the screen. With this facility, you can design your own input forms so that the screens that your operator sees will look just like the old paper forms that were used before.

To get data into the variables on the screen at your chosen locations, type:

^READ^

The cursor positions itself on the first field you entered. You can now type in new data, or leave it the way it was by hitting <enter>. When you leave this field, it goes to the second variable you entered.

Change the data in the remaining two fields. When you finish with the last one, you are back in dBASE II. Now type **^DISPLAY^**. The record now has the new data you entered.

As you can see, **GET** works somewhat like the **INPUT** and **ACCEPT** commands. It is much more powerful than either because it allows you to enter many variables.

A database may have a dozen or two fields (up to 32), but for any given data entry procedure, you may be entering data in only half a dozen of those. Rather than using **APPEND**, which would list all the fields in the database on the screen, you can use **^APPEND BLANK^** to create a record with empty fields, then **GET** only the data you want.

Our <Names> file is not a good example (the accounting system at the end of this section is better), but we can us it to show how to selectively get data into a database with a large structure.

To give you more practice with command files, create a file called <Trial.CMD> with the following commands in it:

```
^ERASE^
^? 'This procedure allows you to add new records to
the'^
^? 'NAMES.DBF database selectively.  We will be
adding'^
^? 'only the Name and the Zip:Code now.'^
^?^
^? 'Type S to stop the procedure,'^
^? '<enter> to continue.'^
^WAIT TO Continue^

^USE Names^
^DO WHILE Continue <> 'S' .AND. Continue <> 's'
^   APPEND BLANK^
^   ERASE^
^   @ 10, 0 SAY "NAME" GET Name^
^   @ 10,30 SAY "ZIP CODE" GET Zip:Code^
^   READ^

^   ? '  S to stop the procedure,'^
^   ? '<enter> to continue.'^
^   WAIT TO Continue^
^ENDDO^
^RETURN^
```

When you're back to CP/M, type ^dBASE Trial^ (or ^DO Trial^ if you renamed the dBASE.COM file as we suggested). Now enter data into several records. After you've finished, LIST the file to see what you've added.

As you can see, data entry is simple and uncluttered.

The screen can be customized by placing prompts and variable input fields wherever you want them.

NOTE: You must use the ^ERASE^ or ^CLEAR GETS^ command after every 64 ^GET's^. Use the latter command if you do not want to change the screen.

A command file that summarizes what we've learned

Before you read on, you can run the following file to see what it does. Type ^dBASE Sample^ if you're in CP/M or ^DO Sample^ if you're in dBASE II. Respond to the prompts. After you've run it, you can come back and go through the documentation. It summarizes most of what we've covered so far and includes copious commentary.

```
*********************** SAMPLE.CMD ***********************
* This command file prompts the user with screen
* messages and accepts data into a memory variable, then
* performs the procedure selected by the user. This is only
* a program fragment, but it does work.
*      We haven't written the procedures that are called
* by the menu yet, so instead, we can have the computer
* perform some actions that show us what it does
* and which paths it takes (stubbing).
*     Normally, dBASE II shows the results of the commands
* on the CRT. This can be confusing, so we SET TALK OFF.

SET TALK OFF
USE MoneyOut
ERASE

* It's good housekeeping to erase the screen before you
* display any new data on it.
* Our substitute display function can be used to put
* information on the CRT screen like this:

?
?
?
?
? '       OUTGOING CASH MENU'
?
?
? '       0 = Exit'
? '       1 = Accounts Payable Summary'
? '       2 = Enter New Invoices'
? '       3 = Enter Payments Made'
?
? '       Your Choice is Number'
WAIT TO Choice
ERASE

* Since we haven't developed the procedures to do these
* three items yet we'll have the computer display
* different comments, depending on which alternative is
* selected from the menu.
```

```
IF Choice = '1'
   @ 0,20 SAY 'One'
ELSE
   IF Choice = '2'
      @ 1,20 SAY 'Two'
   ELSE
      IF Choice = '3'
         @ 2,20 SAY 'Three'
      ELSE
         @  7,20
         @  8,20 SAY ' ANY OTHER CHARACTER INPUT EXCEPT 1, 2, OR 3 '
         @  9,20 SAY ' CAUSES THIS COMMAND FILE TO TERMINATE AFTER '
         @ 10,20 SAY ' PRINTING OUT THIS MESSAGE.  NOTICE THAT THE '
         @ 11,20 SAY ' DIGITS HAD TO BE IN QUOTE MARKS IN THE "IF" '
         @ 12,20 SAY ' STATEMENTS ABOVE BECAUSE THE WAIT COMMAND   '
         @ 13,20 SAY ' ACCEPTS ONLY CHARACTER INPUTS               '
         @ 14,20 SAY '                                             '
      ENDIF 3
   ENDIF 2
ENDIF 1

* Each IF must have a corresponding ENDIF.  We've also
* put a label after the ENDIF to indicate with IF it
* belongs with, to make certain that we have closed all
* the loops.

?
?
?
?
INPUT 'Do you want to continue (Yes or No)?' TO Decision
ERASE
IF Decision
   INPUT "Okay, let's have a number, quickly." TO Number
ELSE
   @ 10,20 SAY "  WHY NOT?  "
   WAIT
ENDIF
ERASE
@ 10,20 SAY " I'M NOT READY FOR THAT.  GOOD-BYE.  "

* This next DO WHILE loop provides a delay of a few seconds
* to keep the last message on the CRT long enough to be read
* before the program terminates.  You may find this useful
* in command files that you write.  To change the delay time,
* either change the limit (100) or the step (+ 1).
STORE 1 TO X
DO WHILE X < 100
   STORE X + 1 TO X
ENDDO
ERASE
RETURN
```

You may want to run the program again. Try all the alternatives, then try entering inputs that are definitely wrong. You'll see how the program works and how dBASE II handles errors.

While it's only a program fragment and doesn't do any useful work, <Sample.CMD> does point up quite a few things:

1. Using **ERASE** frequently is good housekeeping that's easy to do.
2. Using indentation helps make the operation of the program clearer. That's also why we used upper- and lowercase letters. The computer sees them all as uppercase, but this way is much easier for us humans.
3. The "?" can be used to space lines on the display and to show character strings (in quotes or brackets).
4. The **WAIT** command waits for a single character before letting the program move on. The input then must be treated as a character, the way we did in the nested IF's by putting quotes around the values we were looking for.
5. The **INPUT** command waits for and accepts any data type, but characters and strings must be in single or double quotes or square brackets. When you have an apostrophe in your message, use the double quote or square brackets to define the string or the computer gets confused.
6. You don't have to predefine variables. Just make up another name whenever you need one (up to a maximum of 64 active at any one time).
7. Logical values can be treated in shorthand. "IF Decision" in the program worked as if we had said: "IF Decision = T".
8. The **RETURN** at the end of the program isn't necessary, but was tacked on because you would need it if this were a sub-procedure in another command file. That's how the computer knows that it should go back where it came from, rather than just quitting.

Working with multiple databases (PRIMARY, SECONDARY, SELECT)

As we've seen, when you first start working with DBASE II, you type ^USE <filename>^ to tell dBASE II which file you're interested in, then proceed to enter data, edit, etc.

To work on a different database, you type ^USE <NewFile>^. dBASE II closes the first file and opens the second one, with no concern on your part. You can use any number of files this way, both from your terminal and in command files. You can also close a file without opening a new one by typing ^USE^.

When you USE a file, dBASE II "rewinds" it to the beginning and positions you on the first record in the file. In most cases, this is exactly what you want. In some applications, however, you will want to access another file or files without "losing your place" in the first file.

dBASE II has an exceptionally advanced feature that permits you to work in two separate active areas at the same time: PRIMARY and SECONDARY. You switch between them with the ^SELECT^ command

You are automatically placed in the PRIMARY area when you first start. To work on another database without losing your position in the first one, type in ^SELECT SECONDARY^, then ^USE <newfile>^. To get back to the original work area, type ^SELECT PRIMARY^, then continue with that database.

The two work areas can be used independently. Any commands that move data and records operate only in the area in USE.

Information, however, can be transferred from one area to the other using P. and S. as prefixes for variables. If you are in the PRIMARY area, use the S. prefix for variables you need from the SECONDARY area; if you are in the SECONDARY area, use the P. prefix for variables you need from the PRIMARY area.

As an example, this command is used in the <NameTest.CMD> file in the accounting system at the end of this Part of the manual. Individual records in a file in the PRIMARY area are checked against all the records in another file in the SECONDARY area.

The same command is also used in the <TimeCalc.CMD>, <DepTrans.CMD> and <Payroll.CMD> files.

While you may not think of an application now, keep the command in mind: you'll find it useful.

Generally useful system commands and functions

MODIFY COMMAND <filename> lets you modify the named command file directly from dBASE II using the normal full screen editing features.

BROWSE displays up to 19 records and as many fields as will fit on the screen. To see fields off the right edge of the screen, use ctl-B to scroll right. Use ctl-z to scroll left.

CLEAR resets dBASE II, clearing all variables and closing all files.

RESET is used after a disk swap to reset the operating system bit map. Please read the detailed description in the command dictionary (Part II) before using it.

* allows comments in a command file, but the comments are not displayed when the command file is executed. This allows notes to the programmer without confusing the operator. There must be at least one space between the word or symbol and the comment, and the note cannot be on the same line as a command. REPEAT: commands and comments must be on separate lines.

REMARK allows comments to be stored in a command file, then displayed as prompts to the operator when the file is used. There must be at least one space between the word and the remark, and the remark cannot be on a command line.

RENAME <oldfile> TO <newfile> changes file names in the CP/M directory. Do NOT try to rename files in USE.

QUIT TO '<system .COM file list>' allows you to terminate dBASE II and automatically start execution of CP/M and other .COM files. Each .COM file named must be in single quotes, and separated from other file names (in single quotes) by commas.

You can also use the ^?^ command to call out the following functions:

is the record number function. When called, it provides the value of the current record number.

* is the deleted record function, and returns a True value if the record is deleted, False if not deleted.

EOF is the end of file function. It is True if the end of the file in USE has been reached, False if it has not been reached.

A few words about programming and planning your command files

The first thing to do when you want to set up a command file is to turn the computer off.

That's right: that's where many programmers go wrong. They immediately start "coding" a solution, before they even have a clear idea of what they're trying to do.

A much better approach is covered in a number of texts on structured programming and some of the structured languages. One reference you might check is Chapter 2 in "An Introduction to Programming and Problem Solving in Pascal" by Schneider, Weingart and Perlman. Another is Chapters 1, 4 and the first few pages of 7 in "Pascal Programming Structures" by Cherry. Then if you really want to get into programming, there's an excellent text on PL/I called "PL/I Structured Programming" by Joan Hughes.

Briefly, here's the approach:

Start by defining the problem in ordinary English. Make it a general statement.

Now define it further. What inputs will you have? What form do you want the outputs and reports in?

Next, take a look at the exceptions. What are the starting conditions? What happens if a record is missing?

Once you've defined what you want to do, describe the details in modified English. The texts call it "pseudocode". All this means is that you use English terms that are somewhat similar to the instructions that the computer understands.

You might write your program outline like this:

```
Use the cost database
Print out last month's unpaid invoices
Write a check for each unpaid invoice.
```

Adding a bit more detail, it looks like this:

```
USE CostBase
Print out last month's unpaid invoices using
   the SUMMARY.FRM file
Start at the beginning of the database
And go through to the end:
If the invoice has not been paid
   Pay the invoice
   And enter it in the database
Do this for every record
```

In perhaps two more steps, this could be translated into a command file like this:

```
USE CostBase
* Print a hardcopy summary for December, 1980.
REPORT FORM Summary FOR Bill:Date >= '801201' .AND.
      Bill:Date <= '801231' TO PRINT
GOTO TOP                  * Go to the first record
DO WHILE .NOT. EOF        * Repeat for the entire file
   IF Check:Nmbr=' '      * If the invoice isn't paid,
      DO WriteCheck       * write a check, then
      DO Update           * update the records
   ENDIF
SKIP                      *Go to the next record
ENDDO
```

The term <u>top-down, step-wise refinement</u> can be applied to this procedure, but that's forty-three dollars worth of words to say: "Start at the top, then divide and conquer".

Actually, it's just a sensible approach to solving most kinds of problems. First state the overall problem, trying to define what it is and what it isn't. Then gradually get into more and more detail, solving the details that are easy to solve, putting the more complicated details aside for later solution (again, perhaps in parts).

At this stage in our example, we haven't done the <Summary.FRM> file nor the <WriteCheck.CMD> and <Update.CMD> files, but it doesn't matter.

And in fact, we're probably better off not worrying about these details because we can concentrate on the overall problem solution. We can come back after we've tested our overall solution and clean up these procedures then.

Tip: You can still test a partial program like this by using what programmers call stubs. Set up the command files that you've named in the program and enter three items: a message that let's you know the program reached it, WAIT and RETURN. dBASE II will go to these procedure files, give you the message, then return and continue with the rest of the program after you hit any key.