Section IV:

By now you should be writing command files that can perform useful work for you.

To help you further, in this section we will introduce more functions, a few more commands and go into quite a bit of detail on how you can print out your data in exactly the format you want it.

Expanding your control with functions

Functions are special purpose operations that may be used in expressions to perform things that are difficult or impossible using regular arithmetic, logical and string operations. dBASE II functions fall into the same three categories, based on the results they generate.

Functions are called up by typing in ^?^ then a space and the function. They can be called from the terminal or within command files.

NOTE: the parentheses shown below must be used.

(Remember that "strings" are simply a collection of characters (including spaces, digits and symbols) handled, manipulate and otherwise used as data. A "substring" is a portion of any specific string.)

Don't worry about memorizing them now, but do scan the descriptions so that you know where to look when you need one of them in a command file.

!(<variable/string>)

is the lower- to uppercase function. It changes all the characters from 'a'..'z' in a string or string variable to uppercase. Any other characters in the string are unaffected. You'll see this used frequently in the accounting system (Section VI) to convert inputs from the keyboard into a standard form in the files. This makes it simpler when searching for data later, since you will know that all of the data is stored in uppercase, regardless of how it was entered.

TYPE(<expression>)

is the data type function and yields a C, N or L, depending on whether the expression data type is Character, Numeric or Logical.

INT(<variable/expression>)

is the integer function. It "rounds off" a number with a decimal, but does it by throwing away everything to the right of the decimal. The term inside the parentheses (you must use the parentheses) can be a number, the name of a variable or a complex expression. In the latter case, the expression is first evaluated, then an integer is formed from the results.

Note that INT(123.86) yields 123, while INT(-123.86) yields -123. A call to a variable yields a truncated integer formed from the current value of that variable. If we were on record 7 of MoneyOut.DBF, a call to INT(Amount) would produce 2333, the integer part of $2,333.75.

To rounded to the nearest whole number (rather than chop), use this form: INT(value + 0.5). The value within parentheses is first determined, then the integer function of that is taken.

The integer function can also be used to round a value to any number of decimal places. INT(value*10 + 0.5)/10 rounds a value to the nearest decimal place because of the order of precedence of operations (parentheses, then integer, then divide). To round to two places, use "100" in place of the "10"s. For 3 places, use "1000", etc.

VAL(<variable/string/substring>)

is the string to integer function. It converts a character string or substring made up of digits, a sign and up to one decimal point into the equivalent numeric quantity. VAL('123') yields the number 123. VAL(Job:Nmbr) yields the numeric value of the contents of the job number field in our MoneyOut database, since we stored all Job Numbers as characters. You can also use it with the substring operator: VAL($(<string>)).

STR(<expression/variable/number>, <length>, <decimals>)

is the integer to string function. It converts a number or the contents of a numeric variable into a string with the specified length and the specified number of digits to the right of the decimal point. The specified length must be large enough to encompass at least all the digits plus the decimal point. If the numeric value is shorter than the specified field, the remaining portion is filled with blanks. If the decimal precision is not specified, "0" is assumed.

This function is used quite often in the accounting system to simplify displays. Numbers are converted to strings then concatenated with (joined to) other strings of characters for displays.

LEN(<variable/string>)

is the string length function. It tells you how many characters there are in the string you name. This can be useful when the program has to decide how much storage to allocate for information with no operator intervention. However, if a character field variable name is used, this function returns the size of the field, not the length of the contents (since any unused positions are filled with blanks by dBASE II).

$(<expression/variable/string>, <start>, <length>)
is the **substring function**. It selects characters from a string or character variable, starting at the specified position and continuing for the specified length.

As an example, if we had a variable called <Date> whose value was '810823', the function ^$(Date,5,2)^ would give us '23'. To convert these numerals to a number, we could use ^VAL($(Date,5,2)).

You'll find an example of this in the <DateTest.CMD> file in Section VI, where groups of two characters are taken from a 6-character date field, converted to integers (using the VAL(...) function), then evaluated to see if they are in the correct range.

Don't confuse this with the substring logical operator described in Section II.

@(<variable1/string1>, <variable2/string2>)
is the **substring search function**. You might think of this as "Where is string1 AT in string2?" When you use this function, it produces the character position at which the first string or character variable starts in the second string or character variable. If the first string does not occur, a value of "0" is returned.

One use for this is to find out where a specific string starts so that you can use the preceding substring function. Another use is to find out if a specific string occurs at all.

(If you only need to know whether one string is in another one, you can use the relational string operator: String1$String2, Section II.)

You'll find these useful in a command file when the computer is searching without operator intervention, and you can't simply step in and look to see where the data is.

CHR(<number>)
yields the **ASCII character equivalent** of the number. Depending on how your terminal uses the standard ASCII code, ? CHR(12) may clear your screen, CHR(14) might produce reverse video while ? CHR(15) would cancel it. Other functions can be used to control hardware devices, such as a printer. Check your manual--you'll probably find a few interesting features.

To get underlining on your printer, try joining a character string, the carriage return and the underline like this: ? 'string' + CHR(13) + ' '. You could even set up a command file that uses the LEN function to find out how long the string is, then produces that many underline strokes.

&

is the **macro substitution** function. When the symbol is used in front of a memory variable name, dBASE II replaces the name with the value of the memory variable (must be character data). This can be used when a complex expression must be used frequently, to pass parameters between command files, or in a command file when the value of the parameter will be supplied when the program is run.

In the <Reportmenu.CMD> file in Section VI, it is used to get the name of the required database:

```
? 'Which file do you want to review?'
ACCEPT TO Database
USE &Database
```

It could also be used as an abbreviation of a command: ^STORE 'Delete Record' TO D^. The command: ^&D 5^ would then delete record 5 when the program runs.

If the macro command is not followed by a valid string variable, it is skipped. This means that you can use the symbol itself as part of a string without getting an error indication.

FILE(<"filename"/variable/expression>)

yields a True value if the file exists on the disk, False if it does not. If you use a specific file name, use the quote marks. The name of a string variable does not require the quote marks. You can also use any valid string expression: ^FILE("B:"+Database)^ would tell you whether the file name stored in the memory variable <Database> is on drive B (see ReportMenu.CMD in Section VI).

TRIM

eliminates the trailing blanks in the contents of a string variable. This is done by typing:
^STORE TRIM (<variable>) TO <newvariable>

Changing dBASE II characteristics and defaults

dBASE II has a number of commands that control how it interacts with your system setup. You can change these parameters back and forth "on the fly", or set them up once at the beginning of your command file and leave them. In many applications, the defaults will be just what you need.

Parameters are changed in your command files (or interactively) by using the SET command. In the list below, normal default values are underlined.

Once again, there's no need to memorize these. As you work with the established defaults, you can decide if you want to change any of the parameters on the list.

SET TALK ON Displays results from commands on console.
OFF No display.

SET PRINT ON Echoes all output to your 'list' device.
OFF No listing.

SET CONSOLE ON Echoes all output to your console.
OFF Console off.

SET SCREEN ON Turns on full screen operation for APPEND, EDIT, INSERT and CREATE commands.
OFF Turns full screen operation off.

SET FORMAT TO SCREEN sends output of @ commands to the screen

SET FORMAT TO PRINT sends output of @ commands to the printer

SET MARGIN TO <nnn> sets the left-hand margin on your printer ("nnn"<=254)

SET RAW ON DISPLAYs and LISTs records without spaces between the fields
OFF DISPLAYs and LISTs records with an extra space between fields

SET HEADING TO <string> changes the heading in the **REPORT** command

SET ECHO ON All commands in a command file are displayed on your console as they are executed.
OFF No echo

SET EJECT ON enables page feed with **REPORT** command
OFF disables page feed

SET STEP ON Halts after completing each command, for debugging command files.
OFF Normal continuous operation.

SET DEBUG ON sends output from the ECHO and STEP commands to the printer only
OFF sends ECHO and STEP output to the screen

SET BELL ON enables bell when field is full
OFF disables bell

SET COLON ON uses colons to delimit input variables on the screen
OFF disables the colons

SET CONFIRM ON waits for <enter> before leaving a variable during full screen editing
OFF leaves the variable when the field is full

SET CARRY ON carries data from the previous record forward to the new record when in **APPEND**
OFF shows a blank record in **APPEND** mode

SET INTENSITY ON enables dual intensity for full screen operations
OFF disables dual intensity

SET LINKAGE ON permits databases to be linked for display with up to 64 fields and up to 2000 bytes per displayed record. The P. or S. prefix must be used when field names are similar in both databases
OFF disables linkage

SET EXACT ON requires that all characters in a comparison between two strings match exactly
OFF allows different length strings. E.g., 'ABCD'='AB' would be True (Also affects **FIND** command)

SET ESCAPE ON allows the <escape> key to abort command file execution
OFF disables the <escape> key

^**SET ALTERNATE TO <filename>**^ creates a file with a .TXT extension for saving everything that goes to your CRT screen. To start saving, type ^**SET ALTERNATE ON**^.

You can change the file that you are saving to by typing ^**SET ALTERNATE TO <newfile>**^.

To stop, type ^**SET ALTERNATE OFF**^. This also terminates when your QUIT dBASE II.

Merging records from two databases (UPDATE)

Data can be transferred from one database file to another with the following command:

UPDATE FROM <database> ON <key> [ADD <field list>]
[REPLACE <field list>]

Note: Both databases must be presorted on the "key" field before this command is used.

Both files are "rewound" to the beginning, then key fields are compared. If they are identical, then data from the FROM data base is either added numerically to data in the USE file, or is used to replace data in the use file for the fields specified in the field list. When fields do not match, those records are skipped. This command can be used to keep inventory updated, for example.

In the accounting system in Section VI, it is used in <Payroll.CMD> and <CheckStub.CMD>. It's useful and worth experimenting with.

JOINing entire databases

JOIN is one of the most powerful commands in dBASE II. It can combine two databases (the USE files in the PRIMARY and SECONDARY work areas) to create a third database. The form of the command is:

JOIN TO <newfile> ON <expression> [FIELD <list>]

In operation, the command positions dBASE II on the first record of the primary USE file and evaluates each of the records in the secondary USE file. Each time the "expression" yields a true result, a record is added to "newfile". If you are in the primary area when you issue the JOIN command, prefix variable names from the secondary USE file with S.. If you are in the secondary area, prefix variables from the primary USE file with P.. (See example below.)

When each record in the secondary USE file has been evaluated against the first record of the primary USE file, dBASE II advances to the second record of the primary USE file, then evaluates all of the records from the secondary USE file again. This is repeated until all records from the files have been compared against each other.

Note: This can take a great deal of time to complete if the two databases are very large. It may also not be possible to complete at all if the constraints are too loose. Two files with 1,000 records each would create a JOIN database with 1,000,000 records if the JOIN expression was always true, while dBASE II is limited to 65,535 records in any single database.

To use the command, use this sequence of instructions:

```
USE Inventory
SELECT SECONDARY
USE Orders
JOIN TO NewFile FOR P.Part:Number=Part:Number;
        FIELD Customer,Item,Amount,Cost
```

This creates a new database called <NewFile.DBF> with four fields: Customer, Item, Amount and Cost. The structure of these fields (data type, size) are the same as in the two joined databases. (Notice that the "P." prefix is used to call a variable from the work area not in USE.)

Full screen editing and formatting (@..SAY..GET..PICTURE)

dBASE II has a powerful series of formatting commands that allow you to position information precisely where you want it. You saw this in action in our <Sample.CMD> program, where we used:

@ <coordinates> SAY ['prompt'] GET <variable>

This command was able to position prompts and variables (and their values) at any location we specified on the screen. When we listed a series of commands, then followed them with READ, we were able to control the format of the entire screen. You might want to create and run the following command file fragment to refresh your memory:

```
STORE "  "  TO MDate
STORE "   " TO MBalance
STORE "   " TO MDraw
@ 5,5 SAY "Set date MM/DD/YY " GET MDate
@ 10,5 SAY "What is the balance? " GET MBalance
@ 15,5 SAY "How much is requested" GET MDraw
READ
ERASE
@ 5,5 SAY "Should we do an evaluation?" GET MEvaluate
READ
```

The command can also be used without the SAY phrase as **@ <coordinates> GET <variable>** (with a later READ in the command file). This displays only the colons delimiting the field length for the variable.

Tip: In the SCREEN mode the line numbers do not have to be in order, but it's good practice to write them this way since they must be in order for PRINT formatting.

This command can also be expanded for special formatting like this:

@ <coordinates> SAY [expression] GET <variable> [PICTURE <format>]

The optional **PICTURE phrase** is filled in using the format symbols listed below. The command:

@ 5,1 SAY "Today's date is" GET Date PICTURE '99/99/99'

would display:

Today's date is: / / :

assuming that the Date variable was blank. In this example, only digits can be entered.

The GET function symbols are:

9 or # accepts only digits as entries.
A accepts only alphabetic characters.
! converts character input to uppercase.
X accepts any characters.
$ shows '$' on screen.
* shows '*' on screen.

With this command, you can format your menu and input screens any way you want them, quickly and easily.

Tip: The Osborne series of accounting books, besides describing some fairly sophisticated systems, also includes CRT Mask Layouts for menus and entry formats with coordinates clearly marked. Well worth their price for this alone.

Formatting the printed page (SET FORMAT TO PRINT, @..SAY..USING)

When you **SET FORMAT TO PRINT**, the **@**-command sends its information to the printer instead of the screen.

The GET and PICTURE phrases are ignored, and the READ command cannot be used.

Data to be printed on checks, purchase orders, invoices or other standard forms can first be organized on the screen with this command, then printed exactly as you see it:

@ coordinates SAY variable/expression/'string' [USING format]

For printing, the coordinates must be in order. The lines must be in increasing order (print line 7 before line 9, etc.). On any given, line the columns must be in order (print column 15 before column 63, etc.).

This command can output the current value of a variable that you name, the result of an expression, or a literal string prompt message.

If the **USING phrase** is included, this command specifies which characters are printed as well as where they appear on the page. the symbols used are:

9 or # prints a digit only.
A prints alphabetic characters only.
X prints any printable character.
**$** prints a digit or a '$' in place of a leading zero.
***** prints a digit or a '*' in place of a leading zero.

The command **10,50 SAY Hours*Rate USING '$$$$$$$.99'** could be used in both the screen and the printer modes since it has no GET phrase. For Hours = 8 and Rate =12.73, it would print or display **$$$$101.84**, useful for printing checks that are more difficult to alter.

Setting up and printing a form

To set up a form, use measurements based on your printer spacing (ours prints 10 characters per inch horizontally, with 6 lines per inch vertically).

The "Outgoing Cash Menu" that we used in our earlier command file could very well have had another selection item called "4 = Write checks", so we're going to do part of the WriteCheck command file.

To start with, we'll have to input the date. The following command lines accept the date to a variable called MDate, and checks to see whether it is (probably) right:

```
ERASE
SET TALK OFF
STORE "  " TO MDate
STORE T TO NoDate
DO WHILE NoDate
   @ 5,5 SAY "Set date MM/DD/YY" GET MDate PICTURE "99/99/99"
   READ
   IF VAL($(MDate,1,2)) < 1;
        .OR.  VAL($(MDate,1,2)) > 12;
        .OR.  VAL($(MDate,4,2)) < 1;
        .OR.  VAL($(MDate,4,2)) > 31;
        .OR.  VAL($(MDate,7,2)) <> 81
      STORE "  " TO MDate
      @ 7,5 SAY "***** BAD DATE, PLEASE RE-ENTER.  *****"
      STORE T TO NoDate
   ELSE
      STORE F TO NoDate
   ENDIF
ENDDO because we now have a valid date
ERASE
```

In English, the above first sets the value of MDate to 8 blanks, then the @..SAY command displays:

```
Set date MM/DD/YY: / / :
```

When the date is entered, it is checked by the IF to see whether the month is in the range 1-12, day is in the range 1-31, and year=81. This is done in three steps:

- the substring function $ takes the two characters representing the month, day or year (e.g., for month it starts in the 4th position and takes 2 characters)
- the VAL function converts this to an integer
- this integer is then compared against the allowed values

If the value is out of range, MDate is set to blanks again and an error message comes up. When a date within the allowed range is entered, the program continues.

The printout for the check itself could be the next portion of the program. Using the measurements of our

checks, this is the list of commands:

```
@ 8,3 SAY Script * A character variable that prints the
                 * amount in script.  This is filled in
                 * by another procedure called Chng2Scrpt.
                 * We stubbed this for now like this:
                 *  STORE 'Script Stub' TO Script
                 *  RETURN
@ 11,38 SAY Vendr:Nmbr
@ 11,50 SAY MDate
@ 11,65 SAY Amount
@ 13,10 SAY Vendor
@ 14,10 SAY Address
@ 15,10 SAY City:State
@ 15,35 SAY ZIP
@ 17,10 SAY Who
```

You can check this out on your screen before you print it, then switch from SCREEN to PRINT modes with the SET command. The values for the variables are provided elsewhere in your command file.

Longer forms are no problem: a printer page can be up to 255 lines long. To reset the line counter, issue and 'EJECT^ command with the printer selected.

Time to regroup

Because dBASE II is such a powerful system, it has a large number of commands and techniques for dealing with your database needs and allowing you to get more information more easily than any other database system or file handler currently running on micros.

The easiest way to learn the techniques is to go through the examples and key them into your computer, changing names as you go to reflect your needs rather than our examples.

You start by using the CREATE command to create your databases. Besides MoneyOut.DBF, we've created a number of other database structures that you might find useful. They're listed at the beginning of Section VI.

The <Costbase.DBF> file started life as <MoneyOut.DBF>. We've modified it a great deal, changing field names and sizes (with and without data in the database).

We've also added our own spacing between the fields (see fields 5, 7 and 10), rather than using the dBASE II "RAW" default of a space between each field. **NOTE:** you can NOT enter the hyphen as a field name when you CREATE a file. To enter it, you have to MODIFY STRUCTURE, use ctl-N to insert a blank line, then type in the hyphen and the field size. Ctl-W saves the change (ctl-O with Superbrain).

You may want to check some of the other database structures, then see how they are used in the programs. We tried to keep the field names and their individual structures the same for all our databases to allow for file merges and other uses. Data from one database will fit into corresponding fields in another, and with common names the transfer is straightforward.

You might want to check through the command files in Section VI now. Most of the dBASE II commands have been used, and the files work the way they are set up.

The first command file is the main menu for the system, with sub-files selected by pressing a number. Some of the files get a bit complicated (<Payroll> for example), so you might go to some of the utility programs at the end of Section VI before you try to unravel the rest of the programs.

Writing these command files, we used exactly the procedures that we recommended earlier: first define the problem in a general sense. Gradually keep dropping down in levels of detail, using ordinary English at first, then pseudocode, putting terms that dBASE II would understand in capitals when we finally got to that level.

When we came up with something that had to be done, but we weren't sure how to do it, we simply made up a procedure name for it, then went back to it later.

The indentation and mixture of upper- and lowercase

letters was not done just for this manual: it's the way we work all the time. It makes writing the command files a lot easier because you can see groupings of the structures that you are using.

The identifiers were pulled out of our semi-English pseudocode, modified a bit to fit within the 10 characters allowed, but not enough to destroy the meaning.

Comments are sprinkled throughout the files for documentation, although in many cases the programs are almost self-documenting because so many of the dBASE II commands are similar to English equivalents.