



REFERENCE MANUAL

CONTENTS

	page
Foreword	1
Introduction	
Of Languages and Compilers	1A-1
System Requirements	1B-1
Assumptions	1C-1
The Demonstration Disk	1D-1
Installation	
Installing for a 'Native' System	2A-1
Installing for a 'Target' System	2B-1
2.3 to 2.4	2C-1
Operations	
Quick Start	3A-1
Compiling an Application	3B-1
Native Linking	3C-1
Cross-Environment Linking	3D-1
Restrictions	3E-1
Programming Strategies	3F-1
Error Messages	3G-1
dBASE II Co-Products	3H-1
Version Differences	3I-1
Release 1.4	3J-1
Appendices	
Disk Files	4A-1
Semantic Differences	4B-1
Command Restrictions	4C-1
Error Messages	4D-1
dBASE/Compiler Defaults	4E-1
dBASE II Bugs (Fixed)	4F-1
List of Examples	4G-1
Index	5A-1

FOREWORD

Release 1.4 is designed to make it as easy possible for you to incorporate into your dBase/Compiler work procedures any necessary updates in dBase/Compiler documentation or code.

Each chapter of this manual is composed of modules. These modules are independently paginated to allow easy replacement.

A QUICK START section appears at the beginning of Chapter Three (Operations).

Please note that this is not a guide to dBase II. A working familiarity with dBase II is assumed throughout this manual.

Style Conventions

The style conventions adopted in this manual are as follows:

In text,, compiler commands are typed in uppercase letters.

Material contained in angle brackets ('< >') indicates something specified by the programmer, such as:

DO <filename>

Material printed in square brackets indicates it is optional, as in [optional].

INTRODUCTION

A: OF LANGUAGES AND COMPILERS

dBASE II as a language

dBASE II is a Data Base Management System (DBMS) with a data manipulating language. This language is similar in many respects to a 'system control language.' A dBASE II program is thus somewhat similar to a system batch file.

dBASE II is a versatile programming language that offers great freedom in choice of syntax. It also allows one to dynamically create data structures. However, 'there is no free lunch.' This flexibility requires a management overhead that limits the number of variables that a dBASE II program can use, the number of records a file may contain, and the number of files that can be open at any one time. It also determines the speed of execution of a dBASE II program under Ashton-Tate's dBASE II interpreter, and holds implications for anyone attempting to write a compiler for the language.

For example, dBASE II was developed to run on an 8-bit microcomputer. The maximum memory that an 8-bit machine can directly address is 64K. That a data base management system with many mainframe features can execute in such a small memory space is impressive. It is accomplished by using overlays to use memory effectively: only the routines that are currently executing, along with its data and subroutines, will occupy memory space. One challenge that a compiler for dBASE II had to overcome was thus also automatically managing a limited amount of memory to best effect.

If there is any single characteristic that distinguishes dBASE II as a true programming language it is probably the dBASE II control structure, including conditional branching, looping, and other features. Only the absence, until now, of a true dBASE II compiler allowed people to argue that it really isn't a language.

A Compiler for such a language

People do not often compile system control languages. But the advantages of dBASE II - minimal applications development time, ease of use - can be augmented by a compiler that allows an application to interact directly with an operating system. And dBASE II is not merely a system control language. It also has many features of programming languages that are normally compiled.

A Compiler for dBASE II

A language such as dBASE II can be easily implemented with an interpreter, where the runtime environment is determined directly by the source code, line by line. Since there is no variable declaration required; neither are any true sub-routine calls defined. This is effectively what Ashton-Tate sells to execute dBASE II command files.

Implementing a compiler for such a language is no trivial matter. There are considerations which a compiler must address that an interpreter does not, and many of the various strategies a compiler may adopt do not make sense for a language such as dBASE II. dB/Compiler, then, which we have designed as the optimal compiler for dBASE II, does not resemble other compilers in some ways.

For example, a conventional compiler approach is such that the product of compilation is assembler code for the target machine. This assembly code must be assembled using a relocatable or macro assembler for the target machine, and this code must then be linked to produce the executable module.

This conventional approach does not lend itself to data base applications, and hence dBASE II, because of both the complexity involved in the process, and the difficulty involved in managing limited memory. The code and data bases involved in a typical data base application are normally quite large. The actual speed of execution within RAM is greatly impaired by the continual disk access that is required to read and write needed data.

Authors of conventional compilers usually do not worry about the size of object code. And they usually force the programmer to choose the proper tools, such as a macro assembler and an overlay linker, to achieve memory management tasks. This approach would neglect two of the great advantages of dBASE II, which are its ease of use and the minimal development time required for most applications. A compiler for dBASE II should be flexible and powerful, but should also be as easy to use as possible.

An Unconventional Compiler

dB/Compiler is not a conventional compiler. dB/Compiler itself designates certain tasks as overlays, and the memory space these overlays use is already defined for any compilable program. The programmer can work without having to worry about managing memory.

Another unique feature is that dBASE II applications are first compiled into a pseudo-code (a machine level code for a virtual data machine) then linked in an easy

dB/Compiler Reference Manual

operation to produce executable code. This pseudo-code (named Dcode) endows the compiler with abilities it would not otherwise have.

Use of pseudo-code makes it very simple to use dB/Compiler to compile an application, for example. The compiler authors used Dcode to pre-design an overlay structure and a set of instructions which perform the data base and memory management functions so that the programmer need not worry about them.

Pseudo-code does impose a limited loss of efficiency in execution, but this is offset by better designed overlay structures and data access algorithms which render the introduced inefficiency insignificant. WordTech Systems is actively exploring hardware designs that would allow Dcode to directly execute.

dB/Compiler has some characteristics of an interpreter since a machine is not directly executing code from compilation. But there are levels of interpreters. The highest level of interpretation is at the programming source level, where each line of source program is interpreted and immediately executed, such as with dBASE II and BASIC. The lowest level of interpretation is interpreting a pseudo-code, such as a Pascal compiler that generates "Pcode." dB/Compiler operates at this level, and is thus analogous to a Pcode Pascal compiler.

What comprises dB/Compiler?

There are two distinct parts of dB/Compiler: a language processor, and a data base manager. Most conventional languages have only the first, and the second is the responsibility of the programmer; he must write it into his program using the language. These two components are integrated in dB/Compiler, and can be used in two simple steps to produce executable code.

The language processor of dB/Compiler is written in Pascal, and the data base management portion is written in 'C'. (The language processor will soon be rewritten in 'C' for portability.) The compiler object code is a pseudo-code we call Dcode. It is machine-level code for a 'virtual data machine,' or an ideal machine for processing data. Though this might be a bit hard to imagine, since no such machine currently exists, this intermediate level code maximizes the efficiency of a compiler for a language such as dBASE II.

Once produced, this object code is linked with the data base manager to produce an executable program. Several files combine to mimic a virtual data machine runtime environment that allows the dCode to execute. A directly executable module calls overlay files as necessary. The programmer is thus relieved of the burden of designing and testing the overlay scheme.

Advantages of Compilation

Compiled programs will execute without the presence of dBASE II. Yet they are compatible with dBASE II. They can use data files created by dBASE II, and they can also create data files that dBASE II can use. However, compiled programs are NOT callable by dBASE II command files, nor can a compiled program call a non-compiled routine.

There are many dBASE II features which the compiler improves upon. It does not address the basic file and record limitations, but it does fix bugs and other deficiencies. For example, dBASE II full screen editing features are fully supported, with much clearer definition of each possible editing condition. (There are several serious bugs and weaknesses in dBASE II's full screen editor, over which the programmer has very little control. These are corrected in the compiled version of a program, so that it will execute correctly and predictably.) A full list of the dBASE II errors rectified by

dB/Compiler is contained in Appendix F.

In addition to merely correcting errors, other amenities are added. You can SORT on multiple-field expressions in one step, for example.

dB/Compiler will let also you place linked code and data files on specific drives, so that you can manage an otherwise unwieldy, large application. (The Chapter 3 section on programming strategy discusses the drive switches one may use while doing this.)

Compiling a file may also introduce storage and execution time savings. Generally, the larger a data base, the greater will be the time savings when an application using that data base is compiled. Similarly, the space efficiency a compiled application offers varies with the exact nature of the requisite dBASE II command files, including how large they are, how and how frequently they are called, and the number of different dBASE II commands used. (The less the variety, the greater the space savings.)

Run time error detection provides protection against deleting opened files, and system read and write errors. This error detection is possibly better than that of dBASE II. A complete range of diagnostic compile time and run time error messages is included. These are listed in Appendix D.

D: SYSTEM REQUIREMENTS

'Target' System

A system running a compiled application requires minimum 54K (for CP/M 80) or 128K (MS-DOS, CP/M 86) RAM. This is 'clear' of whatever the operating system requires. For example, if an operating system itself occupies 10K of 64K of RAM, only 54K is available for a compiled application to execute in.

This 'target' system must include a computer terminal capable of supporting all dBASE II full screen capabilities.

A machine running a compiled application need have only one disk drive, as long as the application itself does not require more than one drive.

'Host' System

dB/Compiler requires 54K (CP/M 80) or 100K (MS-DOS, CP/M 86) 'clear' RAM to compile and link an application. This is in addition to whatever the operating system requires.

dB/Compiler supports dBASE II version 2.4. If you own version 2.3, you must run the dB/Compiler UPGRADE procedure described in Chapter 2, Installation. This allows the compiler to support your use of memory files to write memory variables.

dB/Compiler supports all dBASE II data access methods, including INDEXing and SORTing. Buffer areas are carefully managed to realize maximum efficiency during data access. The output of every command is completely compatible with dBASE II. The consultant can thus

C: ASSUMPTIONS

Although the compiler's error detection capabilities are thorough, possibly surpassing those of dBASE II itself, it is assumed that the compiler will be invoked on a dBASE II application program that is structurally and logically error-free. (That is, free of syntax, semantic, and logical errors.)

All compile time, link time, and run time errors will be represented by the appropriate error message. To avoid such errors the programmer should know which dBASE II commands are unacceptable to dB/Compiler (please see Appendix C) and which dBASE II commands will be evaluated differently under dB/Compiler execution (please see Appendix B). Finally, a compiled program should be thoroughly tested before sale or production use.

A compiled application must include all routines necessary for successful execution. Uncompiled command files cannot be called by a compiled command file. If the application is to allow routine maintenance of data, commands allowing data editing, deleting and entry must be included in the dBASE II source file before it is compiled.

This area, user entry and modification of data, is the single most likely area for errors to occur if care is not taken. This is why dB/Compiler does not currently support interactive commands that would allow a user to modify a data base without guidance, or somehow inadvertently cause run time execution to abort. (A list of these commands appears in Chapter 3, section E.) All of the necessary functions for data base creation and maintenance can be implemented by substituting dBASE II commands which the compiler supports. This approach ensures safety, flexibility, and the integrity of the data being input. (Chapter 3 also includes a section discussing programming strategies one might use in this regard.)

D: THE DEMONSTRATION DISK

A demonstration disk is provided with the compiler itself. This disk serves two functions. First, it allows a potential user to compile programs and evaluate dB/Compiler. If the purchaser decides that dB/Compiler is not what he or she wants, the package can be returned to WordTech for a refund, less a handling charge, provided the security seal on the compiler disk itself is intact and not tampered with. If WordTech deems the security seal to be broken in any way, WordTech will consider the purchase complete and the return option voided.

Second, the demonstration disk provides the programmer or consultant with a means for compiling limited-use programs to give to clients for trial and inspection. The sample compiler on the demonstration disk allows compilation of dBASE II programs, but it will only allow sixteen records to be entered into the data bases or processed by the command files in those programs. An application compiled by the sample compiler can effectively demonstrate a product while removing the temptation for illicit copying.

Installing the Demonstration Disk

The sample compiler and linker is installed and operated in the same way that the actual compiler is, in a few simple steps. It comes with it's own installation program. Please follow the instructions in Chapter 2.

Operating the Demonstration Disk

All the operations described in Chapter 3 can be performed with the sample compiler. You might use it to investigate the QUICK START section of that chapter.

WordTech recommends that all functions be investigated before the Compiler Service Disk is opened. WordTech will not accept packages for return where the Service Disk has been opened.

Please note the restrictions on semantic and command usage that the compiler imposes. While these are minimal, they can mean that without minor modifications a dBASE II application will not successfully compile, link, or execute.

You might check out some of the unique features dB/Compiler offers. For instance, it will allow you to SORT on many fields at once by compiling a command line such as

```
SORT ON FIELD1 + FIELD2 + FIELD3 TO COMB03
```

When using the sample compiler, there is an important point to remember. Compilation of a dBASE II application with dB/Compiler will not always appear to result in faster execution of that application, or in a module that occupies less memory than the original version of the application. The size of the program portion of a compiled application depends on the structure of command files, how they are called and how frequently, their size, and the commands they contain. The execution time of a compiled application depends both upon the above, and also upon the data bases involved: the number of records, their size, the number of fields, etc.

The sample disk will probably not demonstrate the execution efficiencies you will realize if you compile an actual application. It will probably be slower, since

generally the performance dB/Compiler offers increases geometrically with the number of bytes in a data base. The sample disk, as mentioned before, only allows limited data use; this can limit the demonstrable effectiveness of the compiler. Should you have questions about your programming needs and dB/Compiler, please do not hesitate to contact WordTech to let us advise you on the performance efficiencies you may expect to realize.

INSTALLATION

A: INSTALLING THE COMPILER

You should ensure that you have the right dB/Compiler version for your system: i.e. CP/M 80, CP/M 86, MS-DOS or PC-DOS. The PC-DOS version (and cross-linkers to PC-DOS) does not need to be installed.

Owners of dBASE II version 2.3 should run the dB/Compiler UPGRADE procedure to allow it to support use of memory files to read memory variables from. Please see Section C of this chapter, below.

A menu-driven process, contained in INSTALL.COM, installs dB/Compiler for a development system's terminal. This process installs the linker library for terminal attributes. The linker itself produces a file that allows an application to be installed.

dB/Compiler always assumes an 'intelligent' terminal capable of dBASE II full-screen operations. This holds true for both the 'host' and 'target' (client) systems.

Installing the Linker Library

dB/Compiler includes an installation program, which can be invoked by entering .INSTALL from the operating system prompt. This menu-driven program sets up the linker library file (with extension .LIB). The library file should be on the same drive as the installation program during execution. For CP/M 80 two library files are included; the file to be used should be present when the installation program is executed.

The install program offers many terminals as menu

choices. If a particular terminal is not to be found on the list, one can select a menu-driven process to specify the necessary terminal attributes.

Once a linker library is installed with a particular set of terminal attributes, all applications linked by the linker will contain those terminal attributes as a default. This saves the programmer from having to install a linked application before testing it.

It is not necessary to install the compiler itself; only the library for the linker need be installed.

The PC-DOS version, and cross-linkers to PC-DOS, need not have their libraries installed.

B: INSTALLING AN APPLICATION

dBase/Compiler linkers produce a file that allows a compiled application to be installed. The first letter of this file will be the first letter of the application's name; the rest of the name will be `INSTALL.COM`, as in `AINSTALL.COM` where the name of the application is `ACCOUNT`.

Please note that two different applications with different names, such as `ACCOUNT` and `AGEACONT`, can have installation programs with identical names, such as `AINSTALL.COM`. This could lead to problems if the program developer needs to keep both applications on the same disk. One of the installation programs could easily be renamed, using an operating system rename command, to avoid confusion.

The installation file for the compiled application is not necessary if the compiled application is to execute on a system with the exact terminal attributes of the development system. Also, this installation file is not necessary for execution of the application once the application is installed.

If the target system has terminal attributes other than those of the 'host' system, the end-user can run the application's installation program to install it for the target system. Most terminals are listed as menu choices, and the end-user need merely choose the correct entry. If a terminal is not listed, the end-user can choose a menu-driven process to specify the exact terminal attributes required. This procedure is very similar to the dBASE II installation procedure.

MS-DOS and PC-DOS Applications

A `CONFIG.SYS` file must be prepared to boot a system that will be running a compiled application under MS-DOS or PC-DOS. `CONFIG.SYS` specifies how a system is to be managed, and MS-DOS and PC-DOS look for it during a boot. If `CONFIG.SYS` is not found, MS-DOS and PC-DOS manage a system according to certain default parameters. These default parameters must be changed for a dBase/Compiler application to execute properly. The changes may be made simply by including the correct version of `CONFIG.SYS` on the default drive when a system that will be running the compiled application is booted.

That version of `CONFIG.SYS` should include a line specifying:

```
FILES = X
```

where X is some number greater than eleven. MS-DOS and PC-DOS reserve eight file spaces as a default, but twelve are required for execution of a compiled application.

For MS-DOS, if the run time system contains an ANSI terminal (such as the IBM PC does) the MS-DOS `CONFIG.SYS` should acknowledge this by containing the statement:

```
DEVICE = ANSI.SYS
```

If this line is not included and the system running MS-DOS includes an ANSI driver the compiled application's screen displays will not work properly. When this statement is included the MS-DOS file `ANSI.SYS` must also be included on the default drive of a system when it is booted to run a compiled application.

The preceding statement, `DEVICE=ANSI.SYS`, is not necessary in PC-DOS `CONFIG.SYS` files.

For both MS-DOS and PC-DOS, `CONFIG.SYS` may include an

optional line to maximize performance:

BUFFERS=16

The buffers default is two. Sixteen takes a little more memory, but speeds things up. More than sixteen may also be used.

The CONFIG.SYS file may be created in any one of a number of ways. A standard word processor or text editor may be used to create it. Or, the COPY command may be used, as in the following example:

From the operating system a programmer types:

```
A> COPY CON: CONFIG.SYS<CR>
FILES=12<CR>
DEVICE=ANSI.SYS<CR>
BUFFERS=16<CR>
^Z<CR>
```

(^Z ends edit. CONFIG.SYS is created from the console and the three lines are inserted. ^Z completes the edit session.)

Once the proper CONFIG.SYS file is created, it may be included on a compiled application's distribution disk. It's inclusion will then be 'transparent' at run time.

C: MEM FILE UPGRADE - 2.3 TO 2.4

dB/Compiler supports dBASE II version 2.4. Version 2.3 applications can be compiled if 2.3 memory files are upgraded to 2.4 version memory files. A program is provided with the operating copy of the compiler to allow this.

To upgrade a 2.3 version memory file to a 2.4 version:

From the system prompt, enter

```
UPGRADE filename1 [filename2]
```

Both files should be memory files. If you do not specify an extension UPGRADE assumes the filenames have a .MEM extension.

If only one file is named, that file is changed to a 2.4 version memory file. The old version of the file is put in a file whose name is the same, except it is prefixed with a \$.

Example 1C.1

A 2.3 version file, MEMFILE.MEM, is upgraded to 2.4 with the command:

```
UPGRADE MEMFILE.MEM.
```

MEMFILE.MEM is now a 2.4 version. The 2.3 version is stored in \$MEMFILE.MEM.

If two files are named, filename1 (2.3 version) remains unchanged, but a 2.4 version of it is created and named filename2.

Example 1C.2

MEMFILE.MEM is upgraded to 2.4 with the command:

UPGRADE MEMFILE MD124

The 2.3 version, MEMFILE.MEM, remains unchanged. It's 2.4 version offspring is stored in MD124.MEM.

OPERATIONS

A: QUICK START

1) dB/Compiler can be installed using the INSTALL.COM program provided to specify terminal attributes for the linker library. (The PC-DOS version does not need to be installed.) This is all the installation that is normally necessary. Once the linker library is installed all linked applications will have its terminal attributes as defaults. The linker also produces an installation program that allows these defaults to be changed.

2) Owners of dBASE II version 2.3 can use dB/Compiler with memory files after running the UPGRADE procedure described in Section C of Chapter 2 (Installation). This procedure converts .MEM files to a form usable by dBASE II version 2.4.

3) The MS-DOS and PC-DOS versions of dB/Compiler are copy protected. For those versions the instructions which follow assume that the dB/Compiler distribution disk is present in the default drive of the system in use. For more on copy protection, see page 3I-5.

4) The basic command to invoke the compiler is:

DBC [-A] [-Dx] [-P] [-V] [drive]filename[.extension]

The file specified as filename, and any .FRM files it uses, will be compiled. If no extension is specified, the compiler will assume the file has a .CMD extension. If no drive is specified, the compiler will assume <filename> is on the logged drive.

Compiled object code is written to a file with a name prefaced by a \$. MAILIST.COMD would be, in compiled form, \$MAILIST.COMD. (Object code from the demonstration compiler is marked with a # rather than a \$, as in #MAILIST.COMD.)

-A tells the compiler to automatically compile, in one pass, all other command, form, and format files called by <Filename>. These files must all reside on the same drive as filename. Drive prefixes of files with .CMD, .FORMAT TO, and .REPORT FORM command lines. (See Drive Specifications in Section B, below.)

-D specifies a drive to write object code files on, where the drive letter is not followed by a colon (:). If this option is not used the object code will be written to the disk <filename> resides on.

Compilation normally will stop if any error conditions are detected. A message line describing the error and its file line number will be displayed. Compilation will resume if anything other than ^C is entered. (^C always causes compilation to abort.) The -P option eliminates the need for keyboard input following an error message so that compilation may be left unattended. This is a useful option in conjunction with a printer.

Normally each line of a dBASE II command file is counted as it is compiled, and this count is displayed. The -V option turns off this line count. Any error messages will still be displayed along with the line number the error occurred on. -V is useful in conjunction with the -P option.

5) The native linker is invoked with the command:

```
DBL<vers.> [-Dq] [-Oj] [-Pr] [-Xtv] [-L]
              [drive]filename[.extension]
```

or

```
DBL<vers.> [-I] [-Dz] [drive]filename[.extension]
```

where 'vers.' is an abbreviation for the version of dBase/Compiler in use, as in DBLMS for the MS-DOS version, DBL80 for the CP/M version, etc. Do not include the \$ (or #) in the filename.

The first form above produces executable code in the form of up to five files (the number of overlays depends upon the version of dBase/Compiler being used):

Those files are:

	name	length
CP/M 80:	FILENAME.COM	about 14K
	FILENAME.OVL	variable
	FILENAME.OV1	variable
	FILENAME.OV2	variable
	FINSTALL.COM	about 14K
MS-DOS	FILENAME.EXE	about 90K
	FILENAME.OV1	variable
	FILENAME.OV2	variable
	FINSTALL.EXE	about 24K

The PC-DOS is similar to the MS-DOS version, but no installation program is produced.

The second form (with the -I option) of the linker command produces a file that is linked but not directly executable.

-D specifies the drive(s) to which to write, after linking, the .OV1 (first specifier - 'z' above) and .OV2 (second specifier - 'q' above) overlay files.

-B specifies the drive on which the linker library (with extension .LIB) will reside during linking. For CP/M 80 note that two libraries are supplied. The -L option

below specifies which library to use.

-O, under CP/M 80, specifies the drive to which to write, after linking, the .COM file, the installation program, and the third overlay (.OVL) that the linker produces. Under MS-DOS and PC-DOS -O specifies the drive to which to write the .EXE file and the installation file.

-X (CP/M 80 only) specifies the drives where the .OVL (first specifier - 't' above) and .OVL2 (second specifier - 'v' above) overlay files will reside at run time.

-L (CP/M 80 only) specifies a 56K-memory library (CPM 80) rather than default 54K-memory library. If the application will not be executing on a hard disk and/or RAM disk system, use of this option is strongly suggested.

-I (form 2, above) specifies that linked code will NOT be executable, but rather will be linked into one object code file. This file can then be linked again to produce executable code. This option allows many files to be combined for ease of handling.

6) A compiled and linked application can be run by entering, from the operating system prompt, the name of the executable file. The overlay files must all be present as specified (or default-specified) during the linking process. A specific version of CONFIGSYS must be present at run time under MS-DOS and PC-DOS, as described on page 2B-2.

7) dB/Compiler imposes some restrictions on dBASE II usage (Chapter 2, Section E; and Appendix C). There are also some semantic differences (Appendix B). Programmers with large and/or complex applications should probably expect to have to spend a reasonable

amount of time rewriting code so that it will compile and/or execute properly. Chapter 2, Section F, Programming Strategies, was designed to aid this process.

Example:

The command

DEC -A MAIL

causes the mail list program supplied on the dB/Compiler demonstration disk to be compiled. All the mailist command, format and report form files should be on the disk with the mail system. No colon is used in the drive specifier in the command line.

The CP/M 80 command

DBL80 -Db -Ob -L MAIL

causes the object code to be linked into executable form which is written to drive B:. The CP/M 80 larger linker library is used (because of the CP/M 80 -L option). Under CP/M 80 the files produced are:

MAIL.COM
MAIL.OVL
MAIL.OVL1
MAIL.OVL2
MINSTALL.COM

Other compiler versions produce fewer overlay files.

MINSTALL.COM is not necessary for execution, but can be used to install the compiled application. The application can execute after MAIL.COM is called from an operating system prompt.

Example:

The MS-DOS version command:

```
DBLMS -I ACCOUNT
```

causes the object code file \$ACCOUNT.PRG, and all the other .PRG, .FMT and .FRM files it invokes, to be linked into one object code file, named \$ACCOUNT.PRG. The previous contents of \$ACCOUNT.PRG are destroyed.

Notes for MS-DOS and PC-DOS

A CONFIG.SYS file should be prepared as described in Chapter 2, Section B.

If the overlays .OV1 and .OV2 will not reside on the same run time drive or directory as the .EXE file, their location can be specified at run time. At the operating system prompt enter the name of the application in the form:

```
FILENAME Z:\X\Y
```

where Z:\X\Y is the directory location of the overlays.

B: COMPILING AN APPLICATION

A dBASE II program is compiled into independently executable code by a simple, two step process. In the first step, the program is compiled using dB/Compiler's compiler. In the next step, compilation object code is linked using dB/Compiler's linker. The linker produces an executable file, overlay files, and an installation program for the application. A finished application program can be run merely by entering, from the operating system prompt, the name of the executable file.

The Basic Command

The compiler can be invoked, at the operating system prompt, with the command:

```
DBC [-A] [-Dx] [-P] [-V] [drive]filename[.extension]
```

Follow this command with a carriage return. The file specified as filename will then be compiled. Any form files invoked by the command file will also be compiled. No other command or format files will be compiled unless the automatic option (explained below) is used.

The MS-DOS and PC-DOS versions of dB/Compiler are copy protected. The dB/Compiler distribution disk containing the appropriate .EXE file must be in the default disk drive of a system when the compiler (or linker) is invoked. The instructions which follow are generally true for the copy protected versions, but they can be augmented by the information on page 3I-5.

If no file extension is specified the compiler will assume the file to be compiled has a .CMD extension (CP/M 80) or .PRG extension (MS-DOS, PC-DOS and CP/M

86). If no drive is specified the compiler will assume the file is on the logged drive.

Entry of ^C or <escape> at any time causes compilation to abort.

Relocatable object code files are constructed during compilation, one for each command or format file compiled. These files are recognizable by their file names, which begin with a \$ (or a # for the demonstration compiler) and are completed with up to seven letters of the name of the original file. Thus, the file MAILRPT.CMD would be compiled into \$MAILRPT.CMD.

Object code files are written to the same drive <filename> is on as a default. The -D option (explained below) can be used instead to specify which drive to write object code on.

Normally each line of a command file is counted as it is compiled, and this count is displayed on the screen. The -V option turns off this count, and the -P option eliminates the need for keyboard input following an error message.

Example 3B.1

A programmer working in MS-DOS wants to compile the following dBASE II command file AGEACCT.PRG, which resides on the logged disk in drive A:

```
**ACCOUNTS RECEIVABLE AGING PROCESSING
USE AGEREC
DO AGING
DO B:EXAMINE.111
REPORT FORM AGE1
REPORT FORM AGE2.TTT
SET FORMAT TO BLUE
READ
```

The dBase/Compiler command:

DBC AGEACCT

causes AGEACCT.PRG to be compiled to the object code file \$AGEACCT.PRG, which also resides on A. The report form files AGE1.FRM and AGE2.TTT are also processed and incorporated into \$AGEACCT.CMD, so they must reside on the same drive as AGEACCT.CMD. No other files (such as AGING and EXAMPLE.111) are compiled. Example 3.C.2

links this file.

Note that it was not necessary to specify the .CMD (or .PRG) extension; that is assumed as a default. Nor was it necessary to specify the .FRM extension of the file AGE1; that was assumed as a default.

If we were still logged on drive A, but AGEACCT.PRG, AGE1.FRM, and AGE2.TTT were all on the disk in drive B, the compiler command:

```
DBC B:AGEACCT
```

would cause AGEACCT.PRG to be compiled. The object code file \$AGEACCT.PRG would be written to drive B.

Automatic Compilation

Any separate command and format files that are called by the compiled program will NOT be compiled unless the -A (for automatic) option is used. If it is used, the compiler will automatically compile all other command, format, and form files called by the file specified in the command line. A maximum of 64 command files and 32 report forms may be compiled at any one time using this option. (See Section F of this chapter for strategies regarding applications with greater numbers of files.)

The compiler displays the name of each file being compiled. It displays the count of compiled lines if the count has not been turned off with the -V option.

The compiler makes certain assumptions as it compiles under this option. These assumptions concern the default file extensions of the files it looks for, and the drives it will look for these files on.

File Extensions

If a file is called by a DO FILE, SET FORMAT TO, or REPORT statement, and a filename extension is not specified, the compiler assumes the called file will have an appropriate extension. Thus, the compiler expects a file called by a DO statement to have a .CMD (or .PRG in MS-DOS) extension, a file called by a SET FORMAT TO statement to have an .FMT extension, and a file called by a REPORT FORM statement to have a .FRM extension. Other extensions may be used if desired, but they must be specified in the calling statement or the compiler will not find the file intended. If the compiler does not find a called file it does not abort, but instead skips it; the file can then be compiled later, prior to linking.

Drive Specifications

For the three commands mentioned above, DO FILE, SET FORMAT TO, and REPORT, the compiler ignores drive prefixes in the filename called. All these files MUST reside on the same disk as the file being compiled. This is true no matter what the intentions or flow of the original dBASE II program. (See Section F of this chapter for design strategies.) If the compiler cannot find a file that is actually on another disk, an error message appears, but compilation does not abort. The compiler merely skips that file, which can be compiled later.

For all other statements and commands, drive prefixes will be noted and interpreted as they would under dBASE II execution.

Example 3B.2

After using the PIP (or COPY) command to move

AGING.PRG to drive A from a disk in drive B, a programmer invokes the compiler by entering:

DEC -A AGEACCT

AGEACCT.PRG of Example 3B.1 is compiled to \$AGEACCT.PRG. The following files are also compiled since the automatic option was invoked:

file:	compiled to:
AGING.PRG	\$AGING.PRG
EXAMINE.111	\$EXAMINE.111
BLUE.FMT	\$BLUE.FMT

Note that in the original dBASE II command file the extensions .CMD (or .PRG), .FRM, and .FMT were not always specified but the compiler could infer the necessary information from context. Note that eccentric extensions (such as .111, .111) were specified. Had they not been, an error condition would have resulted as the compiler decided, from context, that it was to look for EXAMINE.PRG or AGE2.FRM which it would not be able to find. In this case the compiler would not have aborted, but would continue compiling other files; the missing file could be compiled later (without the -A option) prior to linking of the application.

Example 3B.3

Assume the programmer in the previous example did not COPY (or PIP) B:AGING.PRG to the drive AGEACCT.PRG is on. The compiler would generate an error message when it reached the command line DO B:AGING because it would ignore the drive prefix, search disk A: for AGING.PRG, not find it, and complain. This would not be true for many other commands, where the compiler does pay attention to the drive prefix.

Specifying Write Locations

-D lets you specify which drive to write compiled code to. Follow the -D with the drive letter, but do not include the colon which is standard for drive names. If you do not use this option to specify a drive compiled code will be written to the same drive that the file being compiled resides on.

Example 3B.4:

A programmer operating on drive A: enters the command:

DEC -DB AGEACCT

while AGEACCT resides on drive A. The compiled file \$AGEACCT.CMD is written to drive B:. Note that no colon was used in the drive specifier.

Continuous Compilation

Normally compilation stops when an error is detected. Compilation will resume when anything other than ^C or Kescape> is entered. The -P eliminates the need for keyboard input following each compile time error detected. The compiler can thus run unattended, and can reach the end of a file without further input from the programmer.

This is a useful option in conjunction with a printer and your operating system print commands. If many error messages are generated during one compilation the initial messages might be 'shoved' off the screen by later ones; using the printer with this option ensures that those early messages are not lost.

No Line Count

The -Y option turns off the line count mechanism. Error messages and the line numbers they refer to will still be displayed, but the count of successfully compiled lines will not.

Object Code

All of the above modes produce relocatable object code. This code is distinguishable by its file name, which is its dBASE II name prefixed with a \$ (\$ for the demonstration compiler). For example, the dBASE II command file TEST.CMD would be compiled into a file called \$TEST.CMD.

If a name is already 8 characters long (the maximum allowed by CP/M and MS-DOS) the last character of the name is truncated to accommodate the \$. For example, the file CONQUEST.CMD would be compiled into a file called \$CONQUES.CMD. This letter will be restored when the code is linked, so that execution of the finished, linked application could begin after typing CONQUEST from the CP/M or MS-DOS prompt.

Please note that while there are ten significant characters for a filename, and extension specifiers are retained intact by the compiler, the eighth character of a name is NOT significant, and might be temporarily dropped by the compiler. This can result in confusion if the programming practice of using filenames which are identical for seven characters, then distinguished by the eighth (such as ACCOUNTA.CMD, ACCOUNTB.CMD, ACCOUNTC.CMD... ACCOUNTX.CMD) is used. The authors of dB/Compiler recommend that the filename extension, which dB/Compiler fully supports, be used as much as

possible to distinguish files.

The object code (called dcode) the compiler produces is pseudo-code, a machine level code for a virtual data machine. (For a discussion of the relationship of this code to compilers and dBASE II, please see Chapter 1.) WordTech Systems, Inc. is currently exploring implementation of such a data base machine for the IBM-PC. This implementation would allow dcode to execute directly. dB/Compiler would then be a compiler in the classic sense.

C: THE NATIVE LINKER

Object code produced by dB/Compiler must be successfully linked to be machine-executable. A 'native' linker produces executable code for the operating environment in which compilation originally occurred. A 'cross-environment' linker works in one environment to link object code that was originally produced in a different environment.

Linking accomplishes tasks such as specifying the addresses of data management functions and producing an executable file. It can also be used as a development step, producing executable code for an environment other than that your compiler runs in.

Once an application is linked it can be run by calling a linker-produced executable file from the operating system. This executable file will call the overlays as necessary. These overlays must reside as specified (or default-specified) during the linkage process.

The dB/Compiler linkers have been designed to be as easy to use as possible. Unlike many other linkers, which require the programmer to specify exactly which files are to be linked, which overlays are to be used, etc., the dB/Compiler linker can operate with only the bare minimum of programmer specification (such as 'one filename.') The programmer has also been allowed flexibility for dealing with problems relating to the capacity of a system.

One way this was achieved was by designing the linker to only include those data management functions that an application actually requires for execution. For example, if no SORT command is used no sort capabilities are included in the finished application. Another was by designing the linker to allow programmer specifications when necessary.

Three types of files are produced by the linker: an executable file, with extension .COM, overlay files which it calls, and an installation program. The overlays allow the object code of the compiler to execute in real machine environments. When linking is complete the application can be run simply by typing the application's name.

Once linked, an application's name should not be changed using operating system commands such as REN (or PIP).

Invoking the Linker

The native environment linker is invoked by entering the name of the linker .COM file at the operating system prompt, along with the name of the file to be linked (including drive and extension, if necessary but omitting the \$ which distinguishes compiler object code). This file must have previously been compiled (or linked under the -I option explained below).

If a system of files is to be linked, only the file which coordinates and calls the other files need be specified in the link command. However, all other files required must be present on the same disk. If a required file cannot be found the linker will abort.

Up to two hundred fifty six (256) object code files can be linked in any one pass.

The MS-DOS and PC-DOS versions of the linker are copy protected. The instructions which follow can be augmented with the copy protection information found on page 3I-5.

The basic form of the command is:

```
DBL<vers.> [-Dzg] [-Oj] [-Br] [-L] [-Xtv]
              [drive]filename[.extension]
```

or

```
DBL<vers.> [-I] [-Dz] [drive]filename[.extension]
```

where <vers.> stands for the dB/Compiler version in use, as in DBLMS for the MS-DOS version, DBL86 for the CP/M 86 version, etc.

The command options -O and -D specify where code should be written to at link time. The -X option specifies where modules of finished code will reside at run time. -B specifies where the linker library file will reside during linking. -I initiates the first of a two-step linking process (an aid in integrating large applications). -L specifies that the CP/M 80 larger linker library be used rather than the default linker library.

Default Linking

If none of the linkage specification options are used, the file specified as <filename> is linked. All routines embodied in separate command files that are called by the specified file are automatically also linked and so must be previously compiled and present on the same disk as the file specified. If no drive is specified, the logged drive is assumed. If no destination is specified, linked code is written back to the drive <filename> resides on. If the -I option is not used the linker produces up to five files. If the -B option is not used, the linker library you will use (with extension .LIB) must be on the same drive as the file to be linked. Under CP/M 80 if the -L option is not used, the default library (for 54K free RAM at run time) is used rather than the larger library.

Example 3C.1

The following CP/M 80 dB/Compiler command:

```
DBL80 MAIL
```

causes the file \$MAILCMD to be linked into executable code. The five files produced are:

```
MAIL.COM
MAIL.OV1
MAIL.OV2
MAIL.OVL
MINSTALL.COM
```

MINSTALL.COM is a program that will install MAIL in its run time environment but is not necessary for execution of the application. MAIL.COM will initiate execution of the application in its run time environment.

16-bit versions of dB/Compiler might produce fewer than three overlays, depending upon the version.

File Sizes

The .COM (or .EXE) and installation files produced by the linker are always of a fixed length. The length of the overlays will vary according to the size and structure of the original application.

Large Memory Library

Two linker libraries are provided with the CP/M 80 model of dB/Compiler. The default is for use when the run time system will only have 54K clear RAM. Many run time systems will have more free RAM, and the programmer can use the -L option to specify that a larger library, for

56K RAM clear at run time, be used during linking. TO MAXIMIZE RUN TIME PERFORMANCE IT IS STRONGLY RECOMMENDED THAT THE LARGE MEMORY OPTION BE USED WHENEVER POSSIBLE.

This option only applies to the CP/M 80 version of dB/Compiler.

Library Location

The -B option specifies the drive on which the linker library (with extension .LIB) will reside during linking. As a default, it is assumed the library file will be on the same drive as the file(s) being linked.

There are two library files supplied with CP/M 80 versions of dB/Compiler: DBL80.LIB, and DBL80LLIB (for 56K or more RAM at run time). The library to be used is specified with the -L option above. Whichever is used, it should reside as specified with the -B option.

Non-Executable Output

The -I option gives the programmer considerable flexibility when planning a system. This version of the link command specifies that the file produced will be in a form that is linked (integrated), but not executable. It must be linked again before it will be executable. As with the previous form of the command, all other command files that are called by FILENAME will be linked automatically.

This form of the command lets one link many files into a single file. For example, 256 is the maximum number of object code files that may be linked in one pass of the linker. If more than 256 object code files must be linked, 255 can be linked under this option as a first step; they would then be counted as one file in the second step. This single linked file would also be easier to transport to various disks than 255 component files would be.

A file linked using this option retains its name and \$ prefix unchanged.

Example 3C.2

When AGEACCT.PRG of Example 3B.2 was compiled, the following files were produced:

```
$AGEACCT.PRG
$AGING.PRG
$EXAMINE.111
$BLUE.FMT
```

Assuming that the above files are all present on the logged disk, the MS-DOS linker command:

```
DLINKS -I AGEACCT.PRG
```

causes \$AGEACCT.PRG to be linked into non-executable form. The previous contents of \$AGEACCT.PRG are destroyed. The new \$AGEACCT.PRG includes all of \$AGING.PRG, \$EXAMINE.111, and \$BLUE.FMT. Although those three files are incorporated into \$AGEACCT.PRG, they themselves are left unchanged.

Drive Specification

The -X, -O and -D options let a programmer distribute compiled, linked code among available space in a system. There are up to five component modules to a finished, linked application (including the installation file which can be used to install a finished application). These modules are not further divisible, but they can be separated if space considerations dictate. The programmer then need only ensure that he has instructed the end-user on how the system was designed (i.e. disk containing .OV1 and .OV2 in drive B:, etc.).

The -D option specifies the drive on which to write two of the overlay files (those with .OV1 and .OV2 extensions) the linker produces. Please note that this does NOT specify how the files will be arranged at

run time.

If the -D option is not used, the default is the drive the file being linked resides on. If only one drive specifier is used, both overlays will be written to that drive. If two are used, .OV1 is placed on the first, and .OV2 is placed on the second. These two overlay files are of variable length, depending upon the length of the command files in the original application. Thus this option allows an executable module to be distributed among the available space of a system.

One may only specify one drive if the -I option is also used. One may specify two drives when not using the -I option. If one uses -I and -D and attempts to specify two drives, the second specification will be ignored and only the first will be used.

The -O option specifies where to write the .EXE (or .COM), .OVL and installation files produced by the linker. Please note that this does not specify where those files will reside at execution time. The default destination for these files is the drive where the file being linked resides.

The -X (CP/M 80 only) option specifies where two overlays (those with extensions .OV1 and .OV2) will reside during execution of the finished application. Use the option in the form -Xzq, where z specifies the drive for .OV1, and q is the drive prefix for .OV2. The .COM and .OVL files may reside anywhere as long as they are on the same drive. If the -X option is not used the default will be the logged drive, i.e. all four files must be on the same drive at execution.

Location of the .OV1 and .OV2 overlays can be specified under MS-DOS at run time in one of two ways. One way is to enter the name of the application at the operating system prompt in the form:

```
<FILENAME> Z:\A\B
```


where Z:\A\B is the directory location of the two overlays. Alternatively, the MS-DOS command:

```
SET DBC=Z:\A\B
```

may be used at run time to indicate where the .EXE file should look for the overlays.

Example 3C.2A

A programmer working under MS-DOS and logged onto drive C: invokes the linker with the command:

```
DBLMS -DA SALESUM
```

SALESUM.OV1 and SALESUM.OV2 are written to drive A. While still logged onto drive C she could immediately test the application by entering:

```
SALESUM A:\B\C
```

where A:\B\C was the last directory she had used while on drive A:.

Example 3C.3:

A programmer operating under CP/M 80 and logged on drive A enters the linker command:

```
DBL80 -DBA -BA -XC B:CONQUEST
```

The object-code file B:\$CONQUES.CMD is linked into an executable form. Executable file CONQUEST.COM, and CONQUEST.OVL and CINSTALL.COM are written to drive B, as is CONQUEST.OV1. CONQUEST.OV2 is written to drive A. When an end-user runs CONQUEST, CONQUEST.OV1 and CONQUEST.OV2 must reside on drive C as specified by the -X option. CONQUEST.COM and CONQUEST.OVL can reside anywhere at run time as long as they are on the same drive.

Example 3C.4:

A programmer enters the MS-DOS linker command:

```
DBLMS -I -DBA CONQUEST
```

The object-code file \$CONQUES.PRG and the files it calls are linked. The previous contents of \$CONQUES.PRG are destroyed, and the new \$CONQUEST.PRG is written to B. The second specifier of the -D option, A, is ignored.

Example 3C.5

A programmer has an application, AGEACCT.CMD, which is just large enough that space is tight on the system the application will be running on. During program development and testing the programmer established that for this particular application AGEACCT.OV1 is the largest file by far. AGEACT.OV2 is smaller but still rather large. Two of the other three files are of fixed length, about 14K. The programmer also knows that, by installing the application before he gives it to his client, he need not include the install program in the files he gives his client. The CP/M 80 linker command:

```
DBL80 -XBA AGEACCT
```

specifies that at run time AGEACCT.OV1 must reside on drive B, and AGEACCT.OV2 must reside on drive A. The

and AGEACCT.OVL can go anywhere as long as they are together on the same drive.

Example 3C.6

A programmer working in CP/M 80 wants to link an art investment tracking application that she has discovered, during development and testing, strains her system's disk capacities. She has also determined that the file with extension .OV1 is the largest file of the finished application. She places her compiled object code files on drive A, along with requisite linker files DBL80.COM and DBL80.LIB (she wants to use the larger memory model). The linker command:

```
DBL80 -CM -DBA -L ARTGOODS
```

links the application into executable form. ARTGOODS.COM, ARTGOODS.OVL, ARTGOODS.OV2 and AINSTALL.COM are all written back to drive A. ARTGOODS.OV1 is written to drive B.

Running a Finished Application

When an application is successfully compiled and linked, it consists of several modules: an executable .COM file three overlay files, and an installation program for the application. The application can be run merely by having the user call the .COM or .EXE file from the operating system prompt.

Example 3C.6

A consultant has written a program, called CONQUEST, to track a football team's season statistics. The compiled and linked version of CONQUEST for CP/M 80 consists of CONQUEST.COM, three associated CONQUEST overlays, and CINSTALL.COM, an installation file. After CINSTALL.COM has executed once, a linebacker can run the program on the team's computer by entering CONQUEST from the operating system prompt.

Notes on Execution

A SORT operation requires that a temporary file be opened at run time. This file is opened on the same drive that the file being sorted to resides on. The size of this file will vary with the size of the SORT key. The programmer should be aware of this when planning an application and allocating disk space.

For MS-DOS and PC-DOS, a specific version of the CONFIG.SYS file needs to be present during booting of a system that will be running a compiled application. This version of CONFIG.SYS is described on page 2B-2.

The copy-protected MS-DOS and PC-DOS versions of dB/Compiler will NOT automatically produce copy-protected applications.

B: THE CROSS-ENVIRONMENT LINKER

dB/Compiler's cross-environment linkers allow a programmer to use one compiler to generate executable code for many environments.

Compilation occurs in the original environment as it normally would, using the version of dB/Compiler appropriate for that environment. The relocatable object code of compilation, embodied in files with names beginning with a dollar sign ('\$'), is then moved to the new environment, and the cross-linker to that environment is used in much the same way that the 'native' linker for that environment would.

This procedure imposes two requirements upon the use of a cross-linker. First, the target environment must be accessible. A cross-linker to CP/M 80 will not work successfully in MS-DOS, and vice-versa. Second, an inter-environment file exchange capability must exist, i.e. modem, transfer program, etc.

The cross-environment linkers generate files with names identical to those of files generated by the standard linkers. Programmers should take care to not confuse these files, as they are not interchangeable.

Environment Differences

Cross-environment linking allows the potentials of other operating systems to be realized. That is, code originally written for one environment (such as 8-bit CP/M) will execute efficiently after cross-linking to another environment (such as 16-bit MS-DOS). Programmers should note, however, that the cross-linkers will not automatically make the syntax or semantic changes that are allowable or necessary when moving between environments.

For example, 16-bit versions of dB/Compiler impose fewer restrictions on dBASE II use than the 8-bit version does. Programmers working from 16-bit environments to CP/M 80 should be particularly careful to check their

dBASE II source files for semantic and syntactic features that are not allowed by the 8-bit dB/Compiler.

Along these lines, some commands and capabilities are unique to a dB/Compiler version or environment. Cross-linkers to CP/M 80 allow use of two linker libraries, and programmers using them should familiarize themselves with the CP/M 80 -L option (page 3C-4). Similarly, the PC-DOS version supports the dBASE II command SET COLOR TO . . . while other dB/Compiler versions do not.

Sections 2A, 2B, 3B, 3C, 3E, 3F, 3I, 4A, 4B, and 4C contain more information about the differences between dB/Compiler versions.

Code should be mature (well tested) before cross-linking. Execution problems can thus be attributed to semantic or other differences between environments, rather than logic or semantic problems in the application itself. In addition, when cross-linkers are used the cross-environment linker library should be installed for the target environment so that executable code will have the default terminal attributes of the target environment. Cross-linked code can then be tested with a minimum of effort, and without having to install the application before it is test run. (Cross-linkers to PC-DOS need no installation.)

Operating the Cross-Linkers

A cross-environment linker is invoked with the command:

```
DBLX<host version><target version> [-Dz] [-O]
      [-Br] [-L] [-Xtv] [drive]filename[.extension]
```

or

```
DBLX<host version><target version> [-I] [-Dz]
      [drive]filename[.extension]
```

The file specified is linked according to the options used. The options affect linking much as described in Chapter 3, Section C. Some options, however, are not

applicable for all cross-linker versions. For example, the -L option only pertains to cross-linkers to CP/M 80.

For a detailed discussion of linker options, please refer to Chapter 3, Section C. Briefly, they are:

- I The linker does not produce independently executable code, but rather links up to 264 separate object code files into one object code file.
- O Specifies where the .COM or .EXE, and .OVL (some versions only) files produced by the linker should be written to following linking.
- D Specifies where the overlays .OVL and .OV2 (some versions only) should be written following linking.
- L (linkers to CP/M 80 only)
Specifies that the large linker library for CP/M 80, DBLX80LIB, (for minimum 56K clear RAM at run time) be used during linking rather than the standard library for 54K clear at run time. This option only applies to linkers to CP/M 80. IT IS STRONGLY RECOMMENDED THAT THIS OPTION BE USED WHENEVER POSSIBLE TO MAXIMIZE RUN TIME EFFICIENCY.
- X Specifies where the overlay files .OVL and .OV2 produced by the linker will reside at run time.
- B Specifies where the linker library file (does not exist for all versions) will reside at run time.

Example 3D.1

The CP/M 80 to MS-DOS cross-linker consists of one file: DBLX80MS.COM. Note the 'X' in the name signifying that the file is for a cross environment linker. Note also that the order of the versions in the name file is important and relates to the actual cross-linker. DBLXMS80.COM would specify the MS-DOS to CP/M 80 cross-linker rather than the CP/M 80 to MS-DOS cross-linker.

A programmer using the CP/M 80 to MS-DOS cross-linker first uses the CP/M 80 dB/Compiler to compile an application named INVENTORY. Four INVENTORY object code files are produced, each with a 'S' as the leading character of its name. The programmer then uses a file transfer program to move the object code files to an MS-DOS machine.

Working in MS-DOS, the programmer first installs the cross-linker library so that it contains the terminal attributes of the system's terminal. He then invokes the cross-linker by entering:

```
DBLX80MS INVENTORY
```

from the MS-DOS prompt. The INVENTORY object code files are linked into an executable INVENTORY.EXE file, two overlays, and INSTALL.EXE. The programmer prepares a CONFIG.SYS file (necessary only for MS-DOS and PC-DOS at run time) and can then run and test INVENTORY.

E: RESTRICTIONS ON THE USE OF dBASE II

Interactive Operations

dBase/Compiler was designed to minimize the probability of an end-user accident causing execution of a compiled application to abort. dBase/Compiler thus purposely limits the scope of some interactive operations, as specified below. (Most of the interactive restrictions will be relaxed in a future release.) In addition, some dBase II usage is impossible to compile — it can only be interpreted with an interpreter such as dBase II.

All prohibited operations can be replaced using allowable commands. Section 10 of this chapter discusses some replacement program strategies.

Some allowable commands execute differently than under the dBase II interpreter. For the most part these semantic differences are barely noticeable. A list of these commands occurs in Appendix B.

Restricted Statements

APPEND*

Note: dBase/Compiler supports APPEND BLANK

(Interactive) BROWSE

CALL (Assembly)*

(Interactive) CHANGE

* will be implemented in future release

(Interactive) CREATE <FILENAME>*

NOTE: dBase/Compiler does support the command
CREATE <filename> FROM <filename>

DISPLAY STATUS

(Interactive) EDIT [n]*

HELP

(Interactive) INSERT

NOTE: dBase/Compiler does support INSERT BLANK

(Interactive) INSERT [BEFORE]

Note: dBase/Compiler does support
INSERT BEFORE BLANK

LIST STATUS

LOAD*

MODIFY COMMAND*

MODIFY STRUCTURE

PEEK*

POKE*

SET CALL TO <address>*

SET CARRY ON/OFF*

SET COLOR ON/OFF*

SET DEBUG ON/OFF*

SET ECHO ON/OFF

*will be implemented in future release

SET FX ON/OFF (IBM PC only)

SET SCREEN ON/OFF

(dB/Compiler always assumes full screen capability of the terminal that an application is being run on)

SET STEP ON/OFF

SET TALK ON/OFF*

(dB/Compiler always assumes SET TALK OFF)

Restricted Macro-Substitutions

Some forms of the macro-substitution (implemented in dBASE II with the `&`) are only possible using an interpreter such as dBASE II. Use of dB/Compiler thus precludes some macro-substitutions. These limits do not jeopardize the real power of either the macro-substitution or dBASE II, and all of the limits can easily be replaced using compilable commands (such as DO and IF structures). This chapter's Section F, on programming strategies, discusses macro substitutions with many examples of allowable and unallowable forms.

The limits are:

a) One may not macro substitute a keyword, such as STORE, USE, SORT, etc.

b) One may usually substitute filenames, variable names, etc. The only times when one may not macro substitute a filename or variable name are into a DO `<file>` command (as DO `&filename`), a SET FORMAT TO `<file>` command (as SET FORMAT TO `&fmtfile`), or into a REPORT FORM `<file>` command (as REPORT FORM `&formfile`).

c) One may not macro-substitute the `&`.

d) Due to space constraints the CP/M 80 version of

dB/Compiler does not support macro-substitution of an expression containing any operators such as +, -, (, <, OR(), etc. One can macro-substitute expressions containing identifiers, constants, etc. This limit is relaxed in 16-bit versions of dB/Compiler.

Variable Names

There are restrictions on variable and field names beyond those imposed by dBASE II. T, F, N, and Y are not allowable names in either their upper or lower case versions; the compiler would have no way of distinguishing them from the logical conditions T(rue), F(alse), N(o) and (Y)es.

INDEX and SORT keys

There are some limits to the keys that are allowed in an INDEX or SORT statement. Unlike dBASE II, dB/Compiler will let you SORT on concatenated keys, (as in SORT ON FIELD1 + FIELD2 TO DEXFILE). However, for both INDEX and SORT operations only character-type fields may be concatenated in the key. Only one numeric field may be used as an INDEX key at a time. If numeric fields are to be concatenated in a key, they must first be converted to character with the STR function. Alternatively, a data base can be modified to include an extra numeric field containing a sum of fields; this field can then be used as a key.

There are also limits to the functions that may be used in a key. The only allowable functions are \$ (substring), I, and STR(), and these functions must be used in certain ways. The STR function may not be nested within an expression, as in \$(STR()). The \$ function may only be nested with the I function, and the I must precede the \$. This restriction will be relaxed in a future release.

Nested Conditional Structures

No more than 20 levels of nested IF statements may be active at any one time.

No more than 20 levels of WHILE phrases may be active at any one time.

No more than 20 levels of nested DO CASE statements may be active at any one time. Please note that this is NOT a limit on the number of CASE phrases that may be active in one DO command.

Comments

Generally, comments on command lines, following the command, are not allowable. The only exceptions to this rule are the loop-end commands ENDDO, ENDIF, and ENDCASE, where a short comment is allowable following the command. Many programmers like to indicate there the condition the statement refers to.

The dBASE II comment operator, an asterisk in the first column of a command line, is fully supported by dB/Compiler except when the asterisk is immediately followed by a backslash ('\'). (Please see Appendix 'A').

Use of comments in source files, for program documentation, will not adversely affect performance as it would under the dBASE II interpreter.

Example:

The following command line would not be successfully compiled by dB/Compiler.

```
STORE 0 TO LCOUNT *Initialize line counter
```

The following command line would be successfully compiled by dB/Compiler:

```
ENDIF Sales > $10,000
```

Expressions

Users of the CP/M 80 version of dB/Compiler should not attempt to use expressions containing over one hundred (100) characters. Expression parsing is very complex, and there is not room in the 8-bit environment to allow more than one hundred characters per expression.

CP/M 80 users should try breaking large expressions into smaller components as an alternative.

Supported Commands

Please note that dB/Compiler does fully support:

- a) General macro substitutions (other than keywords and substitutions yielding expressions) using &.
- b) APPEND BLANK
- c) CREATE <NEW> FROM <OLD>
- d) TEST ()
- e) LOCATE

f) CONTINUE

g) SET CONSOLE ON/OFF

h) INDEX or SORT on multi-field expressions, as in:
INDEX ON FIELD1 + FIELD2 + FIELD3 TO COMDEX

i) all other dBASE II functions not specifically mentioned in the list of restrictions above

F: PROGRAMMING STRATEGIES

Topics in this section include:

- Macro-Substitution
- Design Strategies (for Compilation and Linking)
- User-Edited Data
- Speed
- Syntax

Macro-Substitution

(A summary table of allowable and unallowable macro-substitution forms follows this discussion.)

The macro-substitution function of dBASE II, implemented with an ampersand (&), gives dBASE II data processing power unequalled among many high level languages. Any dBASE II programmer will find macro-substitution not only a useful tool, but also a means of avoiding the limitations of dBASE II. It allows one to create arrays, dynamically create testing conditions, and so forth.

This capability also makes a language such as dBASE II very difficult to compile. A compilable language must have all the conditions and environments that are to be present during the running of a program resolved in advance. Some forms of macro-substitution are thus incompatible with a compiler. Some other macro-substitutions would make it possible for the user of an application to accidentally cause it to abort execution. Finally, some environments (especially the 8-bit environment) simply do not allow both support of some macros and acceptable performance.

The limitations use of dBase/Compiler imposes do not necessarily limit the programming power of a programmer. There are many versions of the macro-substitution which will compile. And all restrictions on the use of the

macro-substitution can be avoided fairly easily using allowable programming techniques. In fact, the rigidity this results in might limit unnecessary run time errors and so 'improve' a programming style.

Programming philosophy aside, the limitations dB/Compiler imposes do mean that if a programmer uses many 'fancy' macro-substitution forms, he or she may have to spend several hours or more replacing some of them with program structures that will compile.

A discussion and examples of how dB/Compiler supports the most important macro-substitutions follows.

Multi-Dimensional Arrays

dB/Compiler does support use of the macro-substitution to create arrays. However, it does not support macro-substitution of the & itself.

Thus, an unallowable use would be something like:

```
STORE '&B + &C' TO A
STORE 3 TO C
STORE 4 TO B
? C + &A
```

which, when compiled, would result in a run time error message rather than 7 as in dBASE II.

An allowable use might be:

```
STORE '7' TO TEMP12
STORE '1' TO DIM1
STORE '2' TO DIM2
? TEMP&DIM1&DIM2
```

which would result in a 7. Combination forms like this should be used carefully, because not all macro forms

will evaluate as they would under dBASE II.

For example, if variables were initiated using the following code:

```
STORE '1' TO A
STORE '2' TO B
STORE '3' TO C
.
.
.
STORE '100' TO TEMP1
STORE '200' TO TEMP2
STORE '300' TO TEMP3
```

using dBASE II one could then macro-substitute the content of one of the TEMP variables with a command line such as:

```
&temp&A
```

One could do this with the 16-bit versions of dB/Compiler but not with the 8-bit version, which would generate an error message at run time. (The compiler would have no way of detecting the error until run time.) For CP/M 80, one would have to first:

```
STORE TEMP&A TO X
```

then use &X to get the same result. (Please note that a form that looks vaguely similar:

```
STORE 'TEMP&A' TO X
```

would result in an error message if used prior to &A, because one would then be attempting to macro-substitute the ampersand (&) itself.)

The difference in the values dB/Compiler and dBASE II would return for the expression &TEMP&A lies in the evaluation algorithms used. dBASE II recursively evaluates each & until the expression is resolved. Because

of space constraints, however, the 8-bit version of dB/Compiler does not use the same evaluation algorithm. Instead, all macro-substitutions within a given level are concurrently evaluated, then the next outer level, etc. Macro-substitution, under CP/M 80, of &TEMP&A in the example above would not work because TEMP is not defined as a variable, while TEMP1, TEMP2, etc. are.

Along these lines, all versions of dB/Compiler accept simple 'layered' macro-substitutions, as in:

```
STORE 'B' TO A
STORE 'C' TO B
STORE 'I' TO C
? &&&A
```

Random File Access

dB/Compiler will support macro-substitutions of filenames in anything except DO <file>, SET FORMAT TO <formatfile>, and REPORT FORM <formfile> commands. Thus:

```
ACCEPT 'Enter Filename' TO FILENAME
```

```
.
.
.
```

```
USE &FILENAME
```

is allowable, as is:

```
IF UCHOICE = '3'
  STORE 'MAILIST.100' to RANFILE
ENDIF
```

```
.
.
.
```

```
USE &RANFILE
```

as is:

```
ACCEPT 'Enter File Group' TO FILEGRP
```

```
ACCEPT 'Enter File Number' to FNUM
```

```
.
.
.
```

```
USE &FILEGROUP&FNO
```

Data file names can be dynamically specified using either string concatenation or user input. The last example above demonstrates user input, which is the most error prone approach and is not recommended as good programming practice. That is, the end user should not bear the burden of knowing the exact names of files - an incorrect specification could cause an error. The second to last example above demonstrates an alternative strategy, one in which the user can choose a specific file without having to specify its name. This approach allows flexibility in selecting a particular data base for processing, yet still need not place undue burdens on the end-user and reduces the probability of a run time error occurring.

A simple mnemonic device can help an end-user choose a particular data file and eliminate the need to remember an exact name. For instance, in the mailing list program contained on the dB/Compiler demonstration disk, a data file name is composed of a fixed name and a concatenated extension number. The end-user need only remember the extensions numbers pertaining to each data base. The program performs error checking to see if that file actually exists. If it does not, the user is given a choice of reentering a number, or returning to the main menu. If it does exist, the file is opened by the following statement:

```
USE &MAILFN
```

where MAILFN has the value of, for example, mailist.350.

dB/Compiler does not support macro-substitution of command file names. When the end-user must define which

dB/Compiler Reference Manual

routine should be executed, the choices will have to be made through nested conditional structures. As far as format and form files are concerned, their management is seldom complex simply because usually there are not very many of them. A typical application might have less than ten forms or formats. A program can easily allow a user to choose one without using the macro substitution. For instance; the end user only needs to be shown the menu that contains the possible selections, rather than being prompted to enter the full name.

On Expressions

The CP/M 80 version of dB/Compiler supports only variables and identifiers in macro-substitutions on expressions. Operators, such as +, -, (, >, CHR(), are not acceptable. The program lines

```
ACCEPT "Enter Expression" TO EXP
```

```
.
.
.
```

```
? &EXP
```

would be an allowable form in both CP/M 80 and MS-DOS versions of the compiler as long as a user does not enter an expression with an operator into EXP. Thus, 'A' would be an acceptable entry in both versions, but 'A + B' would not be acceptable under CP/M 80. Similarly, the following program segment would also be unac-

ceptable under the CP/M 80 version:

```
STORE "A+B" TO X
```

```
.
.
.
```

```
STORE 3 TO A
```

```
STORE 4 TO B
```

```
.
.
.
```

```
? &X
```

as would:

```
STORE 'CHR(35)' TO B
```

```
.
.
.
```

```
? &B
```

Please note that generally attempts to use forms such as these, under CP/M 80, will not be accompanied by error messages at compile time. These errors can only be detected by dB/Compiler at run time. That is, these forms are in a sense compilable, but they are certainly not executable under CP/M 80. They would run under other versions.

A comparison chart follows, showing the macro-substitution forms various versions of the compiler support

EXAMPLES OF MAJOR USAGE OF MACRO-SUBSTITUTION

Keyword Substitution

? Supported in ?
 8-81T 16-81T
 (CP/M 80) (CP/M 86
 PC-005
 MS-005)

(1) ACCEPT "SET PRINTER ON OR OFF" TO STATUS
 SET PRINT STATUS

Alternative 1: (all compiler versions)

```
ACCEPT "SET ... " TO STATUS
IF STATUS = 'ON'
  SET PRINT ON
ELSE
  SET PRINT OFF
ENDIF
```

Keyword Substitution (cont.)

Alternative 2: (all compiler versions)
 (better programming practice, less prone
 to end-user error)

```
ACCEPT "DO YOU WANT TO TURN ON PRINTER? (Y/N)";
TO ANSWER
IF I(ANSWER) = 'Y'
  SET PRINT ON
ELSE
  SET PRINT OFF
ENDIF
```


Keyword Substitution (cont.)

(2) STORE 'SAY \$(SOMETHING,2,4)+CHR(7)+ TO INFO

0 4,4 &INFO

0 7,6 &INFO

(Most often a technique such as this is used solely for saving space; this is not necessary when dB/Compiler is used.)

? Supported in 7
8-bit 16-bit

NO

NO

Alternative (all compiler versions)

STORE \$(SOMETHING,2,4) TO TEMP1

STORE CHR(7) TO TEMP2

0 4,4 SAY TEMP1 + TEMP2

0 7,6 SAY TEMP1 + TEMP2

Alternative 2 (16-bit versions only):

STORE '\$(SOMETHING,2,4) + CHR(7)' TO TEMP

0 7,6 SAY &TEMP

Keyword Substitution (cont.)

(3) STORE "00 WHILE" TO LOOP

&LOOP A=B

END000

&LOOP T

END000

(Again, the primary purpose would probably be to save space. This is normally not necessary when dB/Compiler is used. Use such as this is very hard to read and decipher later.)

? Supported in 7
8-bit 16-bit

NO

NO

alternative (all compiler versions):

replace all &LOOPS with 00 WHILE

Multi-Dimensional Arrays

? Supported in ?
8-bit 16-bit

```
(1) STORE 'CA' TO MEM11
    STORE 'AZ' TO MEM12
    STORE 'NY' TO MEM13
    STORE 'NJ' TO MEM13
    STORE 'CA' TO MEM22
    STORE 'CA' TO MEM23
    STORE '100' TO CA
    STORE '200' TO AZ
    STORE '300' TO NY
    STORE '400' TO NJ
    ACCEPT 'COLUMN' TO C
    ACCEPT 'ROW' TO R
    ? MEMSC&R
```

(If user enters C as '1', R as '2', result will be AZ)

Multi-Dimensional Arrays (cont.)

? Supported in ?
8-bit 16-bit

(2) To access the contents of MEMSC&R:

? &MEMSC&R

no yes

(If the user entered 1 for C, and 2 for D, the result would be 200 in dBASE II and 16-bit versions of dB/Compiler. The 8-bit compiler resolves each variable only once. During substitution, since MEM is not defined, neither is &MEM, so &MEMSC&R becomes &MEM12, which is not further evaluated because of space limitation in the 8-bit environment.)

8-bit alternative:

STORE MEMSC&R TO TEMP
? &TEMP
RELEASE TEMP

Multi-Level Macro-Substitutions

```
(1) STORE 'B' TO A
    STORE 'C' TO B
    STORE 'D' TO C
    STORE 'I' TO D
    ? $$$A
```

would yield a result of: 1

? Supported in ?
8-bit 16-bit

yes yes

Dynamic File Access

```
(1) Command File
    Form File
    Format File
```

as in:

```
ACCEPT "Enter form file name: TO FORMFILE
REPORT FORM &FORMFILE TO PRINT
```

? Supported in ?
8-bit 16-bit

No No
No No
No No

Alternative (all versions):

Replace with menu-driven, programmer specified structures, such as DO CASE.

```
(2) Memory File
    Data File
    Index File
    Text File
```

as in:

```
ACCEPT "ENTER DATA BASE FILE" TO DBFILE
USE &DBFILE
```

Yes Yes
Yes Yes
Yes Yes
Yes Yes

? Supported in ?
8-bit 16-bit

(1) Containing only identifiers and constants,
as in:

STORE '3' TO INDC

.

? INDC

Yes

Yes

(2) Containing operators

STORE 'CHR(13)' TO CR

.

? SCR

No

Yes

8-bit alternative:

STORE 13 TO CR

? CHR(&CR)

(3) Containing function and function names:

STORE '1' TO A

STORE '2' TO B

STORE 'A+B' TO C

? &C

No

Yes

Macro on Macro

? Supported in ?
8-bit 16-bit

(1) STORE '3' TO A
STORE '4' TO B
STORE 'A + B' TO C
? &C

No

No

Dynamic Conditions

7 Supported in 7
8-bit 16-bit

No Yes

```
(1) STORE "A-B" TO CONDI
.
.
STORE "A>B.AND." + CONDI TO CONDI
.
.
STORE "C>D.AND." + CONDI TO CONDI
.
.
DO WHILE $CONDI
.
.
ENDDO
```

(For a discussion of Dynamic Condition alternatives for 8-bit systems, please see next section)

Macro-Substituting Dynamic Conditions (CP/M 80)

(Programmers using the 8-bit version of dB/Compiler can use this section to help them rewrite any portions of their code which use macro-substitutions of expressions containing operators.)

One form of macro substitution that dBASE II programmers have found to be both convenient and powerful is to create a testing condition during run time. The following example demonstrates this.

Example 3F.1 - A typical dBASE II Macro Substitution

Two routines follow which are from a small mailing list manager that performs basic data base management functions: creating a data base, adding data to it, searching for a particular piece of data, and reorganizing the data base. We will analyze macro-substitution in a particular function this mailing list program performs, the search function. The actual search routine, MAILSCH.COMD, is a separate module, and is the first program listed below. Another routine, MAILCOND.COMD, actually constructs the search condition and passes this value back to MAILSCH.

Two versions of both these routines are listed. As they first appear, these routines would not be compilable because of their use of the macro substitution. Following the listings is an example of how they might be recoded in a form that is compilable by dB/Compiler. The programs those segments are from are actually contained on the demonstration disk supplied with dB/Compiler.

Both versions of the mail list program use a data base with the following structure:

```

PRIMARY USE DATABASE
FLD  NAME  TYPE WIDTH  DEC
001  REF   C   020
002  NAME1  C   030
003  NAME2  C   030
004  ADDR1  C   030
005  ADDR2  C   030
006  CITY   C   030
007  STATE  C   002
008  ZIP    C   005
009  PHONE  C   013
010  REMARK C   020
011  TAG    L   001
** TOTAL **      00202

```

Uncompilable Version

There are nine fields of potential concern when searching for a particular record. One might want to find a record for reference number = ddd or Name1 = Joe, or Name2 = Smith, etc.

One might also want a search condition that is a combination of up to nine conditions involving these fields. If a traditional programming language is used, it is almost impossible to achieve such a task without using many lines of code. However, with dBASE II's macro-substitution the programmer can simply ask the user, at run time, to choose up to nine combined searching criterium. Thus the user can first choose a combination of fields to be searched, and then specify exactly what values are to be sought: the specific city,

name, zip, etc. Using the macro substitution, we then form this condition by concatenating one string after another.

A specific value for the memory variable COND might be:

```
NAME1=JOE.AND.CITY=BARMET BAY.AND.ZIP=08437.
```

However, given the above data structure, constructing a condition this way raises the possibility that a string could be longer than 256 characters, which is not allowed by dBASE II. For this reason the condition is broken up into component conditions COND1, COND2, etc.

Once a condition is passed back to MAILSCH, control enters a simple loop sequentially processing each record of the data file. This loop asks if the testing condition applies to the record. If it does not apply, we skip to the next one. The actual question is phrased, for example,:

```
IF.NOT. &COND1
SKIP
```

This verifies that a record yielding the condition COND is the record sought.

The program segments under consideration are as follows:

```

**MAILSCH.CMD FILE TO SEARCH FOR A SPECIFIC DATA RECORD
**THIS ROUTINE CALLED BY MAIL.CMD
STORE ' ' TO ext
@ 10,10 SAY 'Enter file number...' GET ext PICTURE '999'

```

```
@ 4,10 SAY 'Please select any combination from 1 up:
```



```

to 9'
  5,10 SAY 'of the following criteria to base your
search on:'
  7,20 SAY '(1) REFERENCE'
  8,20 SAY '(2) NAME 1'
  9,20 SAY '(3) NAME 2'
  10,20 SAY '(4) ADDRESS 1'
  11,20 SAY '(5) ADDRESS 2'
  12,20 SAY '(6) CITY'
  13,20 SAY '(7) STATE'
  14,20 SAY '(8) ZIP'
  15,20 SAY '(9) PHONE'
  16,20 SAY '(10) EXIT'
  18,10 SAY 'Which one(s) ? ' GET choice PICTURE:
'999999999'
  READ
  IF val(choice)=0 .OR. (('0',choice)>0
    STORE '0' TO choice
    LOOP
  ENDIF choice= "0"
  DO mailcond
  STORE 'Y' TO decision
  DO WHILE .NOT. eof .AND. ((decision)= 'Y'
    IF .not. (dcond1)
      SKIP
      LOOP
    ENDIF .not. (dcond1)
    IF .not. (dcond2)
      SKIP
      LOOP
    ENDIF .not. (dcond2)
    IF .NOT. (dcond3)
      SKIP
      LOOP
    ENDIF
    IF .NOT. (dcond4)
      SKIP
      LOOP
    ENDIF
  ERASE
  1,10 SAY 'File No. ' +text
  1,10 SAY 'Record No. ' +1(str(100000+1,6),2,5)

```

```

  6,10 SAY 'REFERENCE ... ' +ref
  7,10 SAY 'NAME 1 ..... ' +name1
  8,10 SAY 'NAME 2 ..... ' +name2
  9,10 SAY 'ADDRESS 1 ... ' +addr1
  10,10 SAY 'ADDRESS 2 ... ' +addr2
  11,10 SAY 'CITY ..... ' +city
  12,10 SAY 'STATE ..... ' +state
  13,10 SAY 'ZIP CODE .... ' +zip
  14,10 SAY 'PHONE ..... ' +phone
  16,10 SAY 'REMARKS ..... ' +REMARK
  STORE ' ' TO decision
  18,10 SAY 'Continue the search ? (Y/N)' GET:
decision
  READ
  SKIP
  ENDDO WHILE ((decision)= 'Y' .AND. .NOT. eof
  IF ((decision)= 'Y'
    12,0 SAY 'No match found. search ended ! Hit:
any key to continue.....'
    WAIT
  ENDIF
  ERASE
  ENDDO WHILE choice<> '0'
  RETURN

```

```

**MAILCOND.CMD (WITH MACROS)
**ROUTINE TO DEVELOP CONDITIONS FOR SEARCH OR NOTIFY
**CALLED BY VARIOUS COMMAND FILES
  STORE 'Y' TO cond1
  STORE 'Y' TO cond2
  STORE 'Y' TO cond3
  STORE 'Y' TO cond4
  IF choice= '
    RETURN
  ENDIF choice= '
  STORE 1 TO row
  STORE 1 TO pos
  ERASE
  DO WHILE pos<=len(choice)
    STORE 1(choice,pos,1) TO schoice

```

```

DO CASE
  CASE schoicer = '1'
    IF pos=len(choice)
      RETURN
    [ENDIF pos=len(choice)]
    STORE pos+1 TO pos
    LOOP
  CASE schoicer = 'I'
    STORE ' ' TO aref
    ! row,0 SAY 'Enter the REFERENCE ...' GET;
aref
    READ
    STORE cond1 + ' .AND. @TRIM(!AREF)),!(AREF
    F)))>0' TO cond1
    STORE row+1 TO row
    CASE schoicer = '2'
    STORE ' ' TO;
aname1
    ! row,0 SAY 'Enter the NAME 1 ...' GET;
aname1
    READ
    STORE cond1 + ' .AND. @TRIM(!ANAME1)),!;
    (ANAME1)))>0' TO cond1
    STORE row+1 TO row
    CASE schoicer = '3'
    STORE ' ' TO;
aname2
    ! row,0 SAY 'Enter the NAME 2 ...' GET;
aname2
    READ
    STORE cond2 + ' .AND. @TRIM(!ANAME2)),!;
    (ANAME2)))>0' TO cond2
    STORE row+1 TO row
    CASE schoicer = '4'
    STORE ' ' TO;
waddr1
    ! row,0 SAY 'Enter the ADDRESS 1.' GET;
waddr1
    READ
    STORE cond2 + ' .AND. @TRIM(!WADDR1)),!;
    (WADDR1)))>0' TO cond2
    STORE row+1 TO row

```

```

CASE schoicer = '5'
.
.
.
STORE row+1 TO row
ENECASE
STORE pos+1 TO pos
ENDDO WHILE pos(<=len(choice))
RETURN

```

Using macro substitutions as above gives a program maximum flexibility and simplicity. However, dB/Compiler, since it is a compiler, must resolve expressions at compile time. This means that the testing condition mentioned above must be defined at compile time. Not to require this would relegate dB/Compiler to the status of a mere interpreter.

dB/Compiler thus would not compile the code above as it is written. But this does not mean that the programming power endowed by the macro substitution function is lost when one chooses to use dB/Compiler. By substituting other programming techniques, a programmer can achieve the same effects.

Compilable Form

These are basically the same two routines as above, even to their names. However, macro substitutions that are not compilable have been replaced with program structures that are.

```

**KAILSACH.CND (NO-MACROS VERSION)
**ROUTINE TO SEARCH FOR A RECORD BASED ON CONDITION
**CONSTRUCTED IN KAILCOND.CND
STORE ' ' TO ext
P 20,10 SAY 'Enter file number..' GET ext PICTURE '999'

```

```

P 4,10 SAY 'Please select any combination from 1 up to 9'
P 5,10 SAY 'of the following criteria to base your search on:'
P 7,20 SAY '(1) REFERENCE'
P 8,20 SAY '(2) NAME 1'
P 9,20 SAY '(3) NAME 2'
P 10,20 SAY '(4) ADDRESS 1'
P 11,20 SAY '(5) ADDRESS 2'
P 12,20 SAY '(6) CITY'
P 13,20 SAY '(7) STATE'
P 14,20 SAY '(8) ZIP'
P 15,20 SAY '(9) PHONE'
P 16,20 SAY '(10) EXIT'
P 18,18 SAY 'Which one(s) ? ' GET choice PICTURE:
'999999999'
READ
IF val(choice) = 0 .OR. (('0',choice)>0
STORE '0' TO choice
LOOP
ENDIF choice = '0'
**CALL ROUTINE TO CONSTRUCT CONDITION
DB kailcond
STORE 'Y' TO decision
DO WHILE .NOT. eof .AND. !(decision) = 'Y'
IF (uref (>) chr(0))
IF .not. ((TRIM(!NREF)),!(REF))>0)
SKIP
LOOP
ENDIF
ENDIF
IF (aname1 (>) chr(0))
IF .not. ((TRIM(!NAME1)),!(NAME1))>0)
SKIP
LOOP
ENDIF
ENDIF
IF (aname2 (>) chr(0))
IF .not. ((TRIM(!NAME2)),!(NAME2))>0)
SKIP
LOOP
ENDIF
ENDIF

```


dBase/Compiler Reference Manual

```

ENDIF
IF (@addr1 <> chr(0))
  IF .not. (@(TRIM(!(@ADDR1)),!(ADDR1)))>0)
    SKIP
    LOOP
  ENDIF
ENDIF
IF (@addr2 <> chr(0))
  IF .not. (@(TRIM(!(@ADDR2)),!(ADDR2)))>0)
    SKIP
    LOOP
  ENDIF
ENDIF
IF (@city <> chr(0))
  IF .not. (@(TRIM(!(@CITY)),!(CITY)))>0)
    SKIP
    LOOP
  ENDIF
ENDIF
IF (@state <> chr(0))
  IF .not. (!(STATE)=!(HSTATE))
    SKIP
    LOOP
  ENDIF
ENDIF
IF (@zip <> chr(0))
  IF .not. (ZIP=trim(zip))
    SKIP
    LOOP
  ENDIF
ENDIF
IF (@phone <> chr(0))
  IF .not. (!(@PHONE,1,8))+!(PHONE,10,4)=trim(!
    (@PHONE,1,8))+!(PHONE,10,4)))
    SKIP
    LOOP
  ENDIF
ENDIF
ERASE
@ 3,10 SAY 'File No.      ' text
@ 4,10 SAY 'Record No. ' +!(xtr(100000+1,6),2,5)
@ 6,10 SAY 'REFERENCE ... ' *ref

```

```

@ 7,10 SAY 'NAME 1 ..... ' *name1
@ 8,10 SAY 'NAME 2 ..... ' *name2
@ 9,10 SAY 'ADDRESS 1 ... ' *addr1
@ 10,10 SAY 'ADDRESS 2 ... ' *addr2
@ 11,10 SAY 'CITY ..... ' *city
@ 12,10 SAY 'STATE ..... ' *state
@ 13,10 SAY 'ZIP CODE .... ' *zip
@ 14,10 SAY 'PHONE ..... ' *phone
@ 16,10 SAY 'REMARKS ..... ' *REMARK
STORE ' ' TO decision
@ 18,10 SAY 'Continue the search ? (Y/N)' GET decision
READ
SKIP
EODD WHILE !(decision)= 'Y' .AND. .NOT. eof
IF !(decision)= 'Y'
  @ 17,0 SAY 'No match found, search ended ! Hit;
  any key to continue.....'
  WAIT
ENDIF
ERASE
-- ENDD WHILE choice<> '0'
RETURN

```

```

**XALCOND.CMD (VERSION WITH NO MACROS)
**DEVELOPS CONDITIONS FOR SEARCH OR MODIFY
** CALLED BY VARIOUS COMMAND FILES
STORE CHR(0) TO uref

```

```

STORE 1 TO row
STORE 1 TO pos
ERASE
DO WHILE pos<=len(choice)
  STORE !(choice,pos,1) TO schoice
  DO CASE

```

```

CASE schoice= ' '
  IF pos=len(choice)
    RETURN
  ENDIF pos=len(choice)
  STORE pos+1 TO pos
  LOOP
CASE schoice= '1'
  STORE ' ' TO aref
  IF row,8 SAY 'Enter the REFERENCE ...' GET;
aref
  READ
  STORE row+1 TO row
CASE schoice= '2'
  STORE ' ' TO;
unaxel
  IF row,8 SAY 'Enter the NAME 1 ...' GET;
unaxel
  READ
  STORE row+1 TO row
CASE schoice= '3'
  STORE ' ' TO;
unaxel
  IF row,8 SAY 'Enter the NAME 1 ...' GET unaxel
  READ
  STORE row+1 TO row
CASE schoice= '4'
  STORE ' ' TO;
naddr1
  IF row,8 SAY 'Enter the ADDRESS 1.' GET;
naddr1
  READ
  STORE row+1 TO row
CASE schoice= '5'
  STORE ' ' TO naddr1
  IF row,8 SAY 'Enter the ADDRESS 2.' GET;
naddr1
  READ
  STORE row+1 TO row
CASE schoice= '6'
  STORE ' ' TO;
acity
  IF row,8 SAY 'Enter the CITY.....' GET acity

```

```

READ
STORE row+1 TO row
CASE schoice= '7'
  STORE ' ' TO nstate
  IF row,8 SAY 'Enter the STATE.....' GET;
nstate PICTURE 'AA'
  READ
  STORE row+1 TO row
CASE schoice= '8'
  STORE ' ' TO nzip
  IF row,8 SAY 'Enter the ZIP code...' GET nzip;
PICTURE '99999'
  READ
  STORE row+1 TO row
CASE schoice= '9'
  STORE ' ' TO nphone
  IF row,8 SAY 'Enter the PHONE.....' GET;
nphone PICTURE '(999)999-9999'
  READ
  STORE row+1 TO row
ENDCASE
STORE pos+1 TO pos
ENDDO WHILE pos<=len(choice)
RETURN

```

The macro substitution function was not even used above. The program simply put conditions inside loops, and used flags to decide whether or not to test for them.

Any programmer using a macro substitution as in the first example might have to rewrite some part of their code if it is to be successfully compiled by dB/Compiler. Hopefully with the second example as a guideline the job should not be too difficult.

Planning a System

There are two important, related aspects of planning a system of programs that will be compiled by dB/Compiler. Thought must be given to where code will be read from and written to during compiling, linking, and run time. Also, if the application is very intricate (involving more than 64 command and format files for CP/M 80, or 256 for 16-bit versions of the compiler) it might have to be compiled (and linked) in successive steps.

dB/Compiler ignores drive specifications in filenames in three dBASE II commands: DO <file>, SET FORMAT TO <file>, and REPORT FORM <file>. Thus if you are logged onto drive A, compiling a file in automatic mode, and that file has the command

```
DO B:DAYPROC
```

the compiler will ignore the drive specification and look for DAYPROC.CMD only on drive A. If you have not previously moved DAYPROC.CMD to drive A, an error condition will result and DAYPROC.CMD will not be compiled. Use of the SET DEFAULT TO <drive> command prior to a DO <file> command does not avoid this requirement. All files required for a successful compilation must always reside on the same disk.

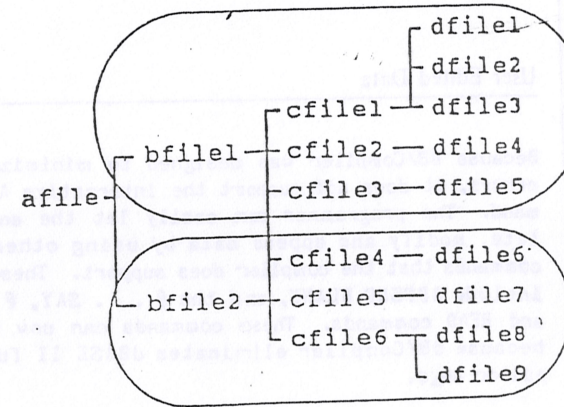
There are certain steps that must be taken when using dB/Compiler. These are:

1) If a command or program file is being compiled in automatic mode all command (or program), format, and form files that the file calls must be present on that file's disk for compilation to be successfully completed. If a command or program file is being compiled in non-automatic mode, any report form files it calls must also be present.

2) If there are too many files involved in the application, the programmer will have to restructure the

main routine, and/or compile and link the files in stages.

For example, consider the case where an application contains a command file that does nothing but invoke other command files and control the default drive specifications. This file could be compiled in non-automatic mode. Other files (which in this example are along conditional branches of a file flow scheme) could then be compiled in automatic mode.



The structure above does not contain 64 files, but it illustrates an approach to a structure that does. Each group of files circled could be compiled in one automatic pass. The coordinating file, afile, could then be compiled in non-automatic mode.

3) Up to 256 object code files may be linked in one pass of the linker. The only real requirement is that all files to be linked be present on the same disk. If more than 256 files need to be linked the -I linking

option can be used to convert up to 256 object code files into a single object code file.

4) By the nature of linking it is nonsense to talk of splitting linked code into component routines. Linked, executable code is composed of various files, however (an executable .COM file, and three overlay files) and it IS possible to specify where these files should reside during run time or written to during program development. This can be done with the -O, -X and -D options during the linking process.

User Edited Data

Because dBase/Compiler was designed to minimize run time errors, it does not support the interactive APPEND command. The programmer can easily let the end-user delete, modify and append data by using other dBASE II commands that the compiler does support. These commands include APPEND BLANK, and the @ . . . SAY, @ . . . GET, and READ commands. These commands can now be trusted because dBase/Compiler eliminates dBASE II full screen editor bugs.

Speed

Inclusion of comments in source code has absolutely no effect on execution time once that code is compiled.

The program segment that follows, when compiled and run on CP/M 80, is a strong candidate for the position of most time consuming routine discovered to date during

alpha and beta testing. It is not a problem in 16-bit environments. It is the REPLACE command that is so expensive.

```
SET FORMAT TO RUN
DO WHILE CONDISHN
READ
APPEND BLANK
REPLACE FIELD1 WITH EXP1, FIELD2 WITH EXP2
STORE FIELD3 TO CONDISHN
ENDDO
```

There are some steps one may take, beyond avoiding the preceding code, to maximize performance of a compiled program.

1) Anyone interested in maximizing CP/M 80 performance should use the CPM 80 large memory linker library (specified during linking, with the -L option) whenever possible.

2) If possible, consecutive REPLACE commands should be replaced with a single occurrence of REPLACE.

3) When extensive data editing is expected, index files should not be used, or NO UPDATE should be issued to avoid costly concurrent index updates. At the end of an editing session a REINDEX command could be issued to update index files.

4) Because dBase/Compiler supports 30 significant digits (dBASE II only supports 10), arithmetic operations execute relatively slowly in a compiled routine under CP/M 80. In general a programmer using CP/M 80 might try to avoid operations such as the following:

```
STORE 25 TO COUNT
DO WHILE COUNT
? 'Welcome to Stark Industries Invoice Entry'
STORE COUNT - 1 TO COUNT
ENDDO
```

dB/Compiler is faster than dBASE II in almost all areas under MS-DOS, and speed strategies are less important within that environment.

Syntax

dB/Compiler requires that dBASE II key words (such as SET, STORE, DO, WHILE, etc.) in command lines be separated by spaces. The compiler will otherwise not correctly parse the command line. Expressions do not need to contain spaces.

For example, the expression:

`?'`

would be acceptable to dBASE II, but would not be recognizable by dB/Compiler. Similarly, the command:

`DO WHILE.NOT.EOF`

would cause dB/Compiler to look for a command or program file. A correct way to phrase the command would be:

`DO WHILE .NOT.EOF`

Please note that the .NOT. and EOF are considered expression elements rather than key words, and do not need to be separated by spaces.

G: ERROR MESSAGES

All run time, compile time and link time errors detected will cause the display of appropriate error messages. Appendix D contains a complete list of these error messages.

Some error conditions cannot be detected at compile time, but will be detected at run time. For this reason an application should be fully run time tested before it is sold or otherwise offered for production use.

For people using dB/Compiler with CP/M 80 undoubtedly the most frequently encountered run time error will be due to use of an expression containing an operator (such as +, -, etc.) in a macro-substitution.

Some error conditions will result in a subsequent statement being declared erroneous, although the declared statement will appear to be syntactically and semantically correct. Probably the most common example of this is a failure to put spaces before and after key words in command lines.

For example, the command line:

`DO WHILE.NOT.EOF`

would cause the compiler to search for a command file called WHILE.NOT. When it subsequently encountered the ENDDO statement it would complain. This error could be fixed by surrounding the key words DO and WHILE with spaces, so that the line read:

`DO WHILE .NOT.EOF`

Note that expressions such as .NOT.EOF do not require spaces.

A discussion of some of the most commonly seen error

messages follows.

Compile Time Error Messages

Only xx command files allowed in one compiling process

The CP/M 80 version of dB/Compiler limits to 64 the number of separate command files (and the number of format files to 32) that may be compiled at any one time in automatic mode. The MS-DOS version limits are 256 command files and 64 format files. If an application is very complex, it may exceed these limits.

One way around this limit would be to separately compile branches of the system in automatic mode. The calling file could then be compiled alone in non-automatic mode. See the discussion of program design strategies in Chapter 3, Section F.

Disk Full

The compiler creates five temporary work files on the disk that the file being compiled is on. If this results in a disk full error, you should be able to tell by checking the disk directory for unfamiliar files. Such a condition can probably be rectified by rearranging the files to be compiled.

Jump Table Overflow

The most likely cause of this error is an attempt to compile a command file that in some respect is too large for the compiler to handle all at once. Try breaking large command files into smaller routines.

Link Time Error Messages

xxxxxx is not relocatable file

The most likely cause of this error is that the file named was not successfully closed during compilation. Try recompiling the file.

Only 256 files allowed in one linking process

Only 256 relocatable files can be linked in any one step. With the -I option, 255 relocatable files can be made into one relocatable file. See the discussion of the -I option in Chapter 3, Section C.

Run Time Error Messages

Either macro substitution for expression or syntax error or variable(s) undefined

95% of the run time errors that first time users of dB/Compiler encounter will be accompanied by this error message. Up to 90% of the occurrences of this error condition will be due to an attempt to macro-substitute an expression containing illegal operators. The balance will probably be due to use of upper or lower case Y, T, N or F as variable or field names.

Refer to Chapter 3, Section E for a discussion of the restrictions on the macro-substitution. Also, Chapter 3, Section F contains a detailed discussion of programming strategies to avoid unallowable macro-substitutions

Editor stack overflow

This condition usually results from an attempt to edit

dB/Compiler Reference Manual

too much of a screen at once. There is a limit on the number of field variables that may be edited at any one time, and the limit depends upon many conditions. If this error occurs, try breaking the edit up into more 'sessions.' don't put so many GET statements between READ statements. This error may rarely occur in 16-bit environments.

Report stack overflow

This message indicates that there are too many items in an attempted REPORT command. Avoid using too many fields at one time. This error may never occur in 16-bit environments.

Illegal key

This message refers to an illegal SORT or INDEX key. A discussions of the restrictions on SORT and INDEX keys occurs in Chapter 3 Section E.

H: dBASE II CO-PRODUCTS

dB/Compiler should be totally compatible with external products that manipulate dBASE II data files but do not alter them in any way. Thus, some product that SORTs or INDEXes a dBASE II file can be used to process that file before a compiled application processes the file. Care should be taken to ensure that these products do not alter the data base in some subtle way that would make it incompatible with compiled code.

dB/Compiler will not be able to compile and/or link applications generated by a program generator if that program contains any of the few program elements not supported by dB/Compiler. The most likely problem to be encountered here is an uncompileable use of the macro substitution. The second most likely problem is use of one of those interactive commands that the compiler does not support.

Programmers using graphics packages with dB/Compiler should note that dB/Compiler requires a special command to send a minus one (255 or hex FF) to an output device. This command, the "\, is detailed in Appendix B.

Appendix C lists all dBASE II command usage that is incompatible with dB/Compiler.

1: VERSION DIFFERENCES

CP/M 80 version - Unique Characteristics

CP/M 80 limitations:

Maximum of 100 characters per expression

No Macro-substitution of expressions containing operators

54K minimum clear RAM at run time

Macro-substitution parsing algorithm not equivalent to dBASE II parsing algorithm; multi-level macros do not evaluate the same as in dBASE II. (Please see page 3F-13.)

Files:

Compiler composed of five files.

Linker composed of two files and one optional file

Optional large-memory linker library (DBL80LLIB) included for use with minimum 56K clear RAM at run time

Linking produces one .COM file, three overlays (with extensions .OVL, .OV1, and .OV2) and one installation file

Special Options:

-X (CP/M 80 only) option allows specification of where overlays .OV1 and .OV2 will reside at run time.

-L (CP/M 80 only) option specifies that the large memory library (for 56K clear RAM at run time) be used during linking

MS-DOS version

Unique Characteristics

Special Preparations:

To run a compiled application an MS-DOS CONFIG.SYS file should be included on the disk in the default drive of the system when it is booted. This version of CONFIG.SYS should include the line:

FILES = X

where X is some number greater than 11. MS-DOS reserves eight file spaces as a default, but twelve are required for execution of a compiled application.

If the run time system contains an ANSI terminal, (such as an IBM PC does) CONFIG.SYS must contain the statement:

DEVICE = ANSI.SYS

If this line is included the MS-DOS file ANSI.SYS also must be included during the boot.

For more information on CONFIG.SYS see page 21-2, and documentation on MS-DOS.

Limitations:

SET COLOR not supported (but is supported in PC-DOS

version).

Minimum 128K clear RAM at run time.

The dBASE II QUIT TO <file> command can only be used to specify .COM and .EXE files. This eliminates use of commands such as: QUIT TO 'DIR.' This limitation will be relaxed in a future release.

Operations

256 files may be compiled (or linked) in any one pass.

MS-DOS linkers produce an installation program file, an .EXE file, and two overlays (.OV2 and .OV1).

The MS-DOS linkers, to be compatible with future multi-user provisions, do not perform selective library extraction. The net effect is that the .EXE file the linker produces is of a fixed size (about 86K) regardless of the size of the original application.

-O linker option specifies the drive the .EXE file should be written to following linking.

-D linker option specifies the drive that the .OV1 and .OV2 files should be written to following linking.

The CP/M 80 -X linker option is not supported in MS-DOS. The directory location of overlays .OV2 and .OV1 can be specified at run time by following the name of the .EXE file with the appropriate path. For example, if one were using the current directory C:\A\B, which contained MAIL.EXE, but MAIL.OV1 and MAIL.OV2 were under directory A:\X\Y, one could invoke MAIL by entering

MAIL A:\X\Y

from the prompt. As an alternative, instead of copying .OV1 and .OV2 to A:\A\B, one could use the MS-DOS command:

SET DEC=C:\A\B.

which flags the symbol DEC for the compiler. When MAIL is then invoked, MAIL.EXE would first try to find MAIL.OV1 and MAIL.OV2 on the current directory. Failing there, MAIL would immediately try C:\A\B.

If location is not specified as above, .OV1 and .OV2 are assumed to reside in the current file directory.

96 memory variables may be used with the MS-DOS version of dB/Compiler. THIS FEATURE SHOULD BE USED WITH CARE since programs adopting it will be incompatible with dBASE II, and it will be impossible to test execution of the application using dBASE II.

2500 bytes of memory have been reserved for memory variables (rather than 1536)

The MS-DOS macro-substitution parsing algorithm is similar to that of dBASE II.

Files

Only one compiler file (DBL.EXE) necessary in MS-DOS.

Only two linker files necessary in MS-DOS (DBLMS.EXE and CBLMS.LIB).

MS-DOS linkers produce only four files for an

application: an .EXE file, an installation file, and two overlays.

Copy Protection

Disks containing the operating version of the MS-DOS dB/Compiler are copy protected. Demonstration disks are not copy protected.

To support the copy protection scheme a few changes have been made in how a file is compiled and linked. The following information can be used to augment the instructions contained in Chapters 3A through 3D of this manual.

The distribution disk provided by WordTech Systems, with the .EXE file, must be in the default drive (usually drive A) of a system when either the compiler or linker is invoked. The compiler and linker may be copied to a hard disk or any other drive, and back-up copies may be made, but no copies will operate correctly unless the distribution disk is in the default drive when the dB/Compiler .EXE file is invoked.

The presence of the operating disk allows the copy protection software to check the signature on the operating disk. Once this check is complete the compiler (or linker) will pause and ask for keyboard input before continuing. At this point the operating disk may be removed and another disk may be inserted into that drive. (Programs to be processed thus do not have to be copied to the WordTech supplied distribution disk.) When a key is pressed, the file specified is processed as specified in the command line (including automatic compilation of a system of files as specified with the -A option).

To eliminate the need for repetitively putting the

Version Differences

operating disk into the default drive each time the compiler is invoked, dB/Compiler asks: "Do you want to compile another file?" when compilation is complete. If 'Y' is entered dB/Compiler will ask for the name of the next file or system of files to be compiled. Otherwise, control returns to the operating system.

Normally only the compiler or the linker (but not both) may be in RAM at any one time. The WordTech supplied operating disk thus must be inserted into the default drive to initiate a new type of operation, as when linking following compilation, etc.

Examples of using the copy protected compiler and linker appear at the end of this chapter, immediately after the section on PC-DOS characteristics.

PC-DOS Version

The PC-DOS version supports the IBM PC and 100% compatibles. It is generally very similar to the MS-DOS version of dB/Compiler, and is copy protected. Screen displays are optimized for the IBM PC. To take advantage of this optimization FORMAT files should be used whenever possible.

A list of unique attributes follows.

PC-DOS Unique Characteristics

The PC-DOS version supports use of the IBM PC cursor keys.

The PC-DOS version supports the dBASE II SET COLOR TO . . . command.

No installation program is necessary (or provided) for the PC-DOS version.

The dBASE II command SET FX TO . . . is NOT supported

A specific version of CONFIG.SYS must boot a system that will run a compiled application, as described for MS-DOS on page 2B-2. The DEVICE=ANSI.SYS line is not necessary, nor is the ANSI.SYS file.

Examples of Use of Copy Protected Versions:

Example 3I-1

A programmer wishes to compile a single dBASE II program file, called WEATHER.PRG. WEATHER.PRG is a large file and the version she wants to compile occupies most of a floppy disk. After booting up her system she places the dB/Compiler distribution disk in her default drive, and types

DBC WEATHER

to invoke the compiler on WEATHER. The compiler operating file DBC.EXE is loaded into RAM, and the copy protection system then checks the operating disk for the proper signature. When the signature is found and verified the message 'Press any key to continue' is displayed. The programmer removes the WordTech supplied operating disk, and inserts the disk containing WEATHER.PRG. She then presses any key and WEATHER.PRG is compiled into the relocatable object code file SWEATHER.PRG. WEATHER.PRG does not call any form (.FRM) files, and so SWEATHER.PRG is the only object code file produced.

Following compilation dB/Compiler asks: 'Do you want to compile another file?' The programmer does not want to compile another file, so she enters 'N'. Control returns to the operating system.

Example 3-I.2

The programmer now wishes to link WEATHER into executable code. From drive A she removes the disk she used for compilation, which now contains WEATHER.PRG, and SWEATHER.PRG. She inserts the dB/Compiler distribution disk containing the DBL.EXE into drive A and enters:

DBL WEATHER

to invoke the linker on WEATHER. The linker operating file DBLXEX is loaded into RAM, the disk signature is checked, and the message 'Press any key to continue' is displayed. The programmer removes the dB/Compiler operating disk and inserts the disk containing WEATHER.PRG and SWEATHER.PRG. The linker produces WEATHER.EXE, WEATHER.OV1, WEATHER.OV2 (which are all necessary for execution), and an installation program, WINSTALL.EXE.

Example 3I.3

Another programmer wishes to compile an application called BILLING. BILLING.PRG is a menu program that calls ten other .PRG files. The programmer wishes to compile the entire system in one pass. After booting his system he places the dB/Compiler distribution disk in his system's default drive, and enters:

```
DEC -A BILLING
```

to invoke the compiler. The -A option specifies that he wants to compile BILLING in automatic mode so that the entire system is compiled in one pass. DEC is loaded into RAM and the disk signature is checked. When the check is complete the compiler displays the prompt: 'Press any key to continue.' The programmer removes the dB/Compiler distribution disk and inserts the disk containing the BILLING system. He then presses any key. BILLING.PRG and all the program, format and form files it calls are each compiled into relocatable object code files. The compiler then asks 'Do you want to compile another file?' The programmer does not, so he enters 'N'.

Control returns to the operating system. The programmer runs BILLING and discovers a small error that necessitates minor changes in BILLING.PRG and three of the ten program files it calls. He uses his editor to implement the changes, swaps disks to insert the compiler operating disk, and then enters the compiler

command:

```
DOC BILLING
```

The programmer does not need to recompile the whole system, just the four edited files. DBLXEX is loaded into RAM and the disk signature is checked. At the prompt the programmer puts the application disk into the drive and presses a key. BILLING.PRG is compiled, but no other files are compiled since the programmer did not use the -A (automatic) option. When compilation is complete the compiler asks: 'Do you want to compile another file?' The programmer does so he presses 'Y.' The compiler asks him to enter the file name; the programmer enters 'AGING' and AGING.PRG is compiled. Following compilation the compiler asks if he wants to compile another file. He does, and presses 'Y.' The compiler asks for a file name, and he enters 'RCEIVBLS.' RCEIVBLS.PRG is compiled . . .

Example 3I.4

The programmer of Example 3 wants to link BILLING into executable code. He puts the dB/Compiler distribution disk into the system default drive and enters DBL BILLING. DBLXEX is loaded into RAM, and the electronic signature on the operating disk is checked. When the check is complete dB/Compiler displays the prompt: 'Press any key to continue.' The programmer removes the dB/Compiler operating disk and replaces it with the disk containing the BILLING object code files. He then presses any key. When the BILLING files are completely linked control returns to the operating system.

Example 3I.5

A programmer works on a computer system containing three

drives. he wants to compile an application named STOCK. After booting his system he places the dB/Compiler operating disk in the system default drive, which is drive A. STOCK.PRG is on drive B. The programmer can compile STOCK by logging drive A and entering:

DBC B:STOCK

DBC.EXE is loaded into RAM, and the signature of the disk in drive A is checked. When the check is complete dB/Compiler prompts the programmer to press any key to continue. He presses a key and STOCK is compiled.

Example 3I.6

A programmer has copied the dB/Compiler operating files onto his hard disk, which is drive C of his system. He wants to operate off drive C, and compile a file there called INVOICES.PRG. After booting his system he places the dB/Compiler operating disk in his default drive, drive A. From drive C he enters:

DBC INVOICES

DBC.EXE is loaded from drive C into RAM. The disk in drive A is checked for the security signature. When the check is complete the compiler prompts the programmer to press any key. He presses a key and INVOICES.PRG is compiled. The object code file is written to drive C:

dB/Compiler Release 1.4 incorporates many new features. Two new compiler and linker options have been added, as have the dBASE II assembler interface commands, along with some additional features.

New Options

-X Option (for 16-bit versions of dB/Compiler; -X option for the 8-bit version is entirely different and remains unchanged.)

The -X option may be used with both the compiler and linker. In each case it disables the message: 'Type any key to continue' prior to compiling or linking. When used with the compiler it also disables the message: 'Do you want to compile another file?'

This feature facilitates batch job processing. If you create a batch file to compile or link many files you can run it without having to resort to keyboard entry between files.

-F Option (for 8 and 16 bit versions)

Through Version 1.3, dB/Compiler truncated the eighth character of a file name to make room for the leading '\$' which distinguishes relocatable object code files.

When the -F option is used with the compiler and a filename consists of eight characters preceding the extension, the first character of the name will be eliminated to accommodate the '\$', rather than the eighth

character.

This option allows programmers to avoid the necessity of renaming command files if they have used the eighth character of a file name as a unique identifier (e.g. they have a system composed of files named MAILORD1.PRG, MAILORD2.PRG, MAILORD3.PRG, etc.).

For example,

DBC MAILORDE.PRG

would generate the object code file \$MAILORD.PRG, but the command:

DBC -F MAILORDE.PRG

would generate the object code file \$AILORDE.PRG.

For filenames with less than eight significant characters preceding the extension it makes no difference whether the -F option is used or not.

If the 'root' file (or menu or calling file) name contains eight characters and the -F option has been used with the compiler, you must also use the -F option with the linker to exactly specify the object code files to be linked. Please note that it is only necessary to specify the name of the 'root' or menu file. The object code version of that file will keep track of the names of the object code files it calls.

For example, given a file called MAIL.PRG, which calls both MAILCOND.PRG and MAILSORT.PRG, the command line:

DBC -A MAIL

would generate the object code files:

\$MAIL.PRG

\$MAILCOND.PRG

\$MAILSORT.PRG

Since the menu file's name has less than eight characters before the extension, both of the following linker command lines adequately specify the desired object code file:

DBLMS MAIL (or DBLPFC MAIL for the PC-DOS version)

or

DBLMS -F MAIL

If the compiler command:

DBC -F -A MAIL

were used instead the compiler would generate:

\$MAIL.PRG

\$AILCOND.PRG

\$AILSORT.PRG

Again, either of the two linker commands:

DBLMS -F MAIL

or

DBLMS MAIL

could be used to link MAIL.PRG and its called files.

Please note that when the name of a menu file contains eight significant characters, there must be consistency between linker and compiler command lines (and object files). For example, if there was a menu program MAILORDE.PRG that called two files, MAILCOND.PRG and MAILSORT.PRG, the compiler command:

DBC -A MAILORDE

would generate the object code files \$MAILORD.PRG, \$MAILCOND.PRG, and \$MAILSOR.PRG. The proper linker com-

mand to use would then be:

BLMS MAILORDE

The -F option would not work with the linker in this case, because it would cause the linker to search for \$AILORDE.PRG, which it would not find, rather than \$MAILORD.PRG.

Assembler Interface Commands

Release 1.4 also implements the dBASE II assembly language interface commands PEEK, POKE, SET CALL TO, CALL and LOAD.

The above commands are implementation dependent. dB/Compiler users hoping to compile these commands may have to modify their existing programs to ensure their proper execution. WordTech suggests you test each assembly routine separately before it is integrated into the complete application system.

The limited space available in the CP/M 80 environment has forced two separate implementations of the assembler interface commands, one for CP/M 80 and one for 16-bit operating systems. The 16-bit implementation is discussed first, below.

16-Bit Environments

Programmers using 8086/8088 versions of dB/Compiler for assembler interface commands should note the most important difference between dB/Compiler and dBASE II: a FAR RETURN (203, or CB Hex) must be used in place of the NEAR RETURN (195, or C3 Hex) that would be used under dBASE II. (Please see Page 3J-7).

In the 8086/8088 implementation dB/Compiler allocates another 64K segment of RAM on top of the Stack Pointer (SP; please refer to Figure 1) for anyone who uses PEEK, POKE, or LOAD. If less than 64K of RAM is left for use, whatever remains is allocated.

Under this scheme a programmer can POKE any address from 301 to 65536, and CALL from any address between 301 and 65536, without worrying about crashing the application. The addresses are completely under the programmer's control (unless QUIT TO ... is used; please see page 3J-8 below).

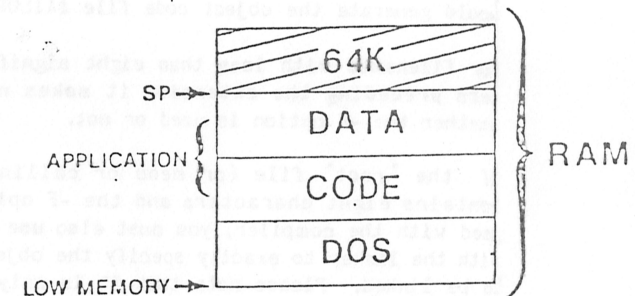


Figure 1

The following sample dBASE II program portion clears the 64K segment above the data segment of the application

Program:

```
STORE 301 TO COUNT
DO WHILE COUNT < 65536
  POKE COUNT,0
  STORE COUNT + 1 TO COUNT
ENDDO
```

Under dB/Compiler the program should still execute. This would not be the case under execution by the dBASE II interpreter; this program portion would crash the system.

Only this added 64K portion can be PEEKed or POKEd by your program. The same is true for LOAD, since LOAD is really nothing but another form of POKE.

Please note that in loading an INTEL HEX file, dB/Compiler ignores the first line and the last line. If your assembler does not generate a header line, you must add one line:

```
:* (:<CR>)
```

To prevent the first line of your code from being skipped. Similarly, if your assembler does not generate a trailer line, you should append the line:

```
:00 (:00<CR>)
```

to the end of the file.

Understanding the 1.4 protocol is very important. Please refer to figure 2. In Step 1, assume you POKE or LOAD something into the added 64K segment. In Step 2, assume you issue a CALL command, as in:

```
CALL MEMVAR1
```

before jumping to the calling address (previously set by the SET CALL TO <address> command; default is 300)

dB/Compiler reserves 255 bytes right above the SP to store the contents of the memory variable, and sets register BX to point to this area (ie. BX is set to 0).

The first character pointed by BX is the length of the memory variable, followed by its contents.

In Step 3 a 'FAR CALL' is issued, i.e. CS, DS and ES are set to point to the location 0 of the added 64K segment, then PC gets the value of the address set by the SET CALL TO . . . command. The codes are then executed.

To return from the assembly program, one must issue a FAR RETURN (OCBH or 203) rather than the NEAR RETURN (OC3H or 195) one would use with dBASE II. This is probably the most important modification of programs previously executed under dBASE II. A system will crash if this change is not made.

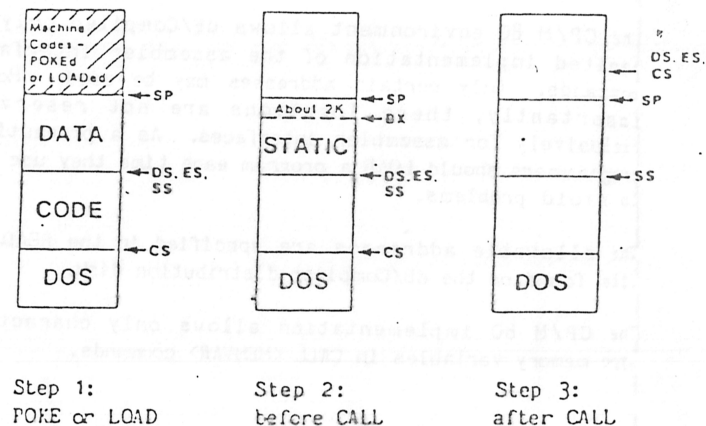


Figure 2

After returning from the assembly routine, the contents of MEMVAR1 of the application program (pointed to by register BX) will be restored back to MEMVAR1. That is, we can change MEMVAR1's value in the assembly routine, a technique sometimes known as "call by reference." One can even change MEMVAR1's length, by changing the byte pointed to by BX. (This would not be allowed by dBASE II.)

Please note that if you use the QUIT TO ... statement, you might have to POKE or LOAD your program again. QUIT TO ... also uses the memory portion above the Stack pointer.

CP/M 80 Implementation

The CP/M 80 environment allows dB/Compiler only a limited implementation of the assembler interface commands. Only certain addresses may be used. More importantly, these locations are not reserved exclusively for assembler interfaces. As a precaution programmers should LOAD a program each time they use it to avoid problems.

The allowable addresses are specified in the README file found on the dB/Compiler distribution disk.

The CP/M 80 implementation allows only character-type memory variables in CALL <MEMVAR> commands.

Debugging Aids

One fourth major feature of Release 1.4 is that if a compiled application aborts due to an error, "LIST

MEMORY" and "LIST STRUCTURE" (for each open data file) commands will be automatically issued after the error message.

This feature aids in debugging an application.

Graphics Characters

Another aspect of Release 1.4 will be transparent to most users. Previous releases of dB/Compiler turned off the high bit of the first character of a string if the first character was larger than 128. This is no longer true, and you can now easily use all the graphic characters supplied by your computer.

PC-DOS dB/Compiler

The final feature of Release 1.4 is a set of changes to the PC-DOS version of dB/Compiler. None of these changes are incorporated in any other versions.

In Version 1.3, when a compiled application finished executing the user had to press any key to return control to the operating system prompt.

With version 1.4 the screen is cleared and control returns to the operating system immediately after a program finishes executing. Programmers who need to display a message or who want to require keyboard entry for some other reason can use a WAIT command.

Release 1.4 also adds an installation program to the compiler for PC-DOS. This does not affect video attributes, but rather allows specification of the dBASE II date format. The linker for PC-DOS now produces an installation program that installs a compiled applica-

tion with a date format. This file is named with the first letter of the application's name, and INSTALL.COM, as in MINSTALL.COM for an application titled MAIL.

APPENDIX A

Disk Files

Operating Disk - CP/M 80

Compiler:	Linker:	Installation:
DEC.COM	DBL80.LIB	INSTALL.COM
DEC.001	DBL.LIB	UPGRADE.COM
DEC.002	DBL80.COM	
DEC.003		
DEC.004		

Files comprising a compiled application (named TEST):

TEST.COM
 TEST.OVL
 TEST.OV1
 TEST.OV2
 TINSTALL.COM (not necessary for execution)
 Requisite data files, .MEM files,
 etc.

Operating Disk - MS-DOS; PC-DOS

Compiler:	Linker:	Installation:
DBC.COM	DBLMS.EXE	INSTALL.EXE
	DBLMS.LIB	UPGRADE.EXE

A compiled application named TEST would contain the following files:

TEST.EXE
 TEST.OV1
 TEST.OV2
 TINSTALL.EXE
 Requisite data and .MEM files

Note: MS-DOS and PC-DOS versions need a specific CONFIG.SYS file present during at run time, as specified on page 2B-2.

Five temporary files are created during compilation (on object code disk). These are purged at the end of compilation.

The Demonstration disk contains a sample application, and a documentation file called README

One temporary file is opened during SORT operations, on the drive on which the file being sorted to resides.

Operating Disk - CP/M 80 to MS-DOS Cross-linker

XDBL80MS.EXE

XDBLMS.LIB

Operating Disk - MS-DOS to CP/M 80 Cross-linker

XDBLMS80.COM

XDBL80.LIB

All other cross-linker files will be named similarly.

APPENDIX B

dBASE II - dB/Compiler Semantic Differences

The following will generate semantic differences between dB/Compiler and dBASE II:

Calculations (involving more than ten significant digits)

INPUT

CHR(255)

ESCAPE

INT(expression)

LIST MEMORY

REPORT

RETURN

Semicolon (in a LIST or DISPLAY command)

SORT (different keys are allowed)

VAL(expression)

& (Macro-Substitution)

(Record Number)

2.41 Features

An in depth discussion of each follows.

Calculations (with more than ten significant digits)

dB/Compiler will allow up to thirty significant digits during calculations. dBASE II allows only ten. Since dB/Compiler data outputs are completely compatible with dBASE II, data will have only ten significant digits after being saved to a .MEM file by a compiled routine.

For example:

123456789.12 would be evaluated by dBASE II as 123456789.00 during processing and storage. It would be evaluated as 123456789.12 during processing by dB/Compiler, but as 123456789.00 after a save operation. 1234567892.35 would be evaluated as 1234567890.00 by dBASE II during processing and storage, but as 1234567892.35 by dB/Compiler during processing, and 1234567890.00 after saving.

For a discussion of how this affects execution speed please see Chapter 3 Section F, Programming Strategies.

CHR(255) (Hex FF)

Due to an internal representation scheme dB/Compiler will not normally allow a hex FF (= 255) to be sent to an output device. That is, dB/Compiler will not support an expression such as CHR(255). It would send a null character. A special feature has been included in dB/Compiler for graphics capabilities. A leading asterisk followed immediately by one or more backslash ('\') characters will be evaluated as a comment by dBASE II, and so ignored in processing, but when compiled will return as many 255's as there are backslashes. For example:

*\\

will be regarded as a comment line by both dBASE II and dB/Compiler. (Please note that there is a space between the asterisk and the first backslash)

*a\

will be regarded as a comment line by both dBASE II and dB/Compiler.

*\

will be regarded as a comment line by dBASE II, but will send one 255 when compiled.

*\\

will be regarded as a comment line by dBASE II, but will send three 255's when compiled.

Escape

When ESCAPE is SET ON and the Escape key is pressed, dBASE II returns to the dBASE II dot prompt.

When ESCAPE is SET ON in a dB/Compiled routine and the escape key is pressed, dB/Compiler asks 'ARE YOU SURE (Y/N)?'. If anything other than a 'y', 'Y', 't' or 'T' is entered, execution immediately resumes. If a 'y' or 'Y' or 't' or 'T' is entered, control returns to the operating system. The programmer can always SET ESCAPE OFF.

Input

dB/Compiler will support entry of only constants and identifiers via the INPUT command. No expressions containing operators (such as +, -, etc.) may be used.

In 16-bit environments the alternative:

ACCEPT <expression> TO A
? &A

may be used.

INT

Due to features inherent in a compilable language, INT function operands will be evaluated differently by dB/Compiler than dBASE II. This might necessitate a change in programming practice if the INT function is used to determine whether a number is evenly divisible. The algorithm

$$\text{INT}(a/b)*b=a$$

should be used rather than

$$\text{INT}(a/b)=a/b,$$

since the latter will be evaluated differently by dB/Compiler than by the dBASE II interpreter. For example, dBASE II evaluation of

? INT(1979/4)=1979/4

would yield the logical operator FALSE, but dB/Compiler evaluation of the same expression would yield a TRUE. They both would evaluate the expression

? INT(1979/4)*4=1979

as false.

List Memory

dB/Compiler stores memory variables differently than dBASE II does. To remain compatible with dBASE II the LIST MEMORY command results in a description of memory as it would be mapped under dBASE II, rather than as it is actually mapped. This minor difference should have no practical effect on operations.

Report

The REPORT command executes under dB/Compiler with minor differences in space adjustments and line counts.

Return

The RETURN command in dBASE II returns control to either the calling routine, or to the dBASE II interactive mode (the dot prompt) if there is no calling routine.

The RETURN command under compiled execution returns control to the calling routine if there is one, and to the operating system if there is not.

Semicolon

dB/Compiler fully supports use of the semicolon to extend a command line beyond the column width of a computer terminal during command file execution. That is, if a dBASE II program contains very long command lines, and semicolons have been used to continue some of those lines, those semicolons will not prevent successful compilation of the file.

However, dB/Compiler does not support use of the semicolon as a continuation operator to printer or terminal. Semicolons in these situations will be treated as a mere character.

Example:

The dBASE II command line

? 'I;like;dB/Compiler'

would cause screen output, upon dBASE II execution, such as:

```

I
like
dB/Compiler

```

while the same command line, when compiled by dB/Compiler, would cause screen output during execution in the form:

I;like;dB/Compiler

Sort

dB/Compiler will let you specify an expression containing concatenated character fields as the SORT key,

rather than just a single field.

For example, a compiled

`SORT ON CONTROL + DOC TO COMBO`
might produce a data ranking in COMBO such as:

	control	doc
record one:	a	1
record two:	a	2
record three:	b	1
	.	.
	.	.
	.	.

This operation would require, in uncompiled dBASE II, the steps:

```

SORT ON CONTROL TO TEMP
USE TEMP
SORT ON DOC TO COMBO

```

Please note that the concatenated fields must be of character-type. Up to thirty fields may be concatenated in one key, depending upon the length of the field names involved. The key expression may not exceed 254 bytes.

Val

A compiled VAL operand will be evaluated slightly differently than under dBASE II execution when decimal places or non-numeric characters are contained. Ending non-numeric characters evaluate as zero, but leading numeric decimal places are represented in both the

length and value of the resulting string. Examples:

expression	dBASE II evaluation	dB/Compiler evaluation
VAL('3.45abc')	3	3.45
VAL('3.33abc')	0	0.33
? VAL('3.45aa')	3	3.45
? VAL('3.45aa') + 4.000	3.45	3.45

&(Macro-Substitution)

There are two ways in which the macro-substitution function evaluates differently under dB/Compiler than dBASE II.

First, if the logical operators (such as .NOT., .OR., .AND., etc.) are used immediately following a macro-substitution, a space should be inserted between the last character of the macro-substitution and the first period ('.') of the logical operator. Otherwise, the compiler will interpret the logical operator as part of the name of the variable being substituted. Thus:

IF &TEST1.&AND.&TEST2

would result in an error, while:

IF &TEST1 .&AND.&TEST2 would not. (In general the compiler does not require spaces between elements of an expression.

Second, the CP/M 80 version of dB/Compiler does not evaluate nested macro-substitutions exactly as dBASE II does. Each level of substituted variables is evaluated simultaneously, then the next level, rather than each

variable recursively, as under dBASE II. For an analysis, please see page 3F-13.

#(Record Number)

dB/Compiler normally allows a record number to equal zero only following a FIND operation in which the search criterion was unfulfilled. In this respect dB/Compiler is much closer to dBASE II version 2.41 than it is to version 2.4. dBASE II version 2.4 allows a record number to equal zero much more frequently. For example, the command:

SKIP -#

would result in:

? #
0

under dBASE II version 2.4, but would result in:

? #
1

during execution under dB/Compiler. Similarly, following a SORT under dBASE II the record number function is set equal to zero. Following a SORT under dB/Compiler the record number function is set equal to the number of the last record contained in the data file.

2.4) Elements Supported

While dB/Compiler supports dBASE II version 2.4, a few elements of dBASE II version 2.41 are also included.

Please note that these do not include any of the multi-user provisions. The supported elements are:

A)
CLEAR

When compiled this command has the same effect as SET FORMAT TO SCREEN, in addition to closing all data and index files, and clearing memory variables.

B)
PICTURE 'A'

dB/Compiler supports the 'A' element of the PICTURE clause, to allow entry of blanks.

C)
RESTORE FROM <memory file> ADDITIVE

If the memory file contains a variable with the same name as an active variable, the active variable is destroyed by dB/Compiler and the restored variable is made active. dBASE II version 2.4 would generate an error condition given these circumstances.

D)
SAVE TO <file> [ALL EXCEPT]

The parameter 'ALL EXCEPT' is fully supported.

APPENDIX C

Restrictions on dBASE II Use

All of the following restricted dBASE II commands can be replaced with compilable alternatives. The restrictions are discussed in greater detail in Operations, Section E. Alternative programming strategies are discussed in Operations, Section F.

Restricted Statements

APPEND*

(Interactive) BROWSE

CALL (Assembly)*

(Interactive) CHANGE

(Interactive) CREATE <FILENAME>*
NOTE: dB/Compiler does support the command
CREATE <filename> FROM <filename>

DISPLAY STATUS.

(Interactive) EDIT [n]*

HELP

(Interactive) APPEND
Note: dB/Compiler does support APPEND BLANK

(Interactive) INSERT
NOTE: dB/Compiler does support INSERT BLANK

* will be implemented in future release

(Interactive) INSERT [BEFORE]
Note: INSERT BEFORE BLANK is supported

LIST STATUS

LOAD*

MODIFY COMMAND*

MODIFY STRUCTURE

PEEK*

POKE*

SET CALL TO <address>*

SET CARRY ON/OFF*

SET COLOR ON/OFF

SET DEBUG ON/OFF

SET ECHO ON/OFF

SET FX TO

SET SCREEN ON/OFF

(dB/Compiler always assumes full screen capability of the terminal an application is being run on)

SET STEP ON/OFF

SET TALK ON/OFF*

(dB/Compiler always assumes SET TALK OFF)

* will be implemented in future release

Restricted Macro-Substitutions

a) One may not macro-substitute a keyword, such as STORE, USE, SORT, ON, etc.

b) One may usually substitute filenames, variable names, etc. The only times when you may not macro-substitute a filename or variable name are into a DO <file> command (as DO &filename), a SET FORMAT TO <file> command (as SET FORMAT TO &fmtfile), or into a REPORT FORM <file> command (as REPORT FORM &formfile).

c) One may not macro-substitute the &.

d) In CP/M 80 versions of dB/Compiler one may not macro-substitute an expression containing any operators such as +, -, (, <, etc. One can macro-substitute expressions containing identifiers, constants, etc. This restriction is relaxed in MS-DOS versions of dB/Compiler.

Variable Names

T, F, N, and Y are not allowable names in either their upper or lower case versions.

INDEX and SORT keys

Only character fields may be concatenated within an INDEX key or SORT key. Only \$, ! and STR() functions are in a key, and then only in certain uses. The STR() function may not be nested within an expression, as in \$(STR()). The \$ function may only be nested with the ! function, and the ! must precede the \$.

Nested Conditional Structures

No more than 20 levels of nested IF statements may be active at any one time.

No more than 20 levels of WHILE phrases may be active at any one time.

No more than 20 levels of nested DO CASE statements may be active at any one time.

Comments

The only comments on command lines allowable are in the loop-end commands ENDDO, ENDIF, and ENDCASE.

The dBASE II comment operator, an asterisk in the first column of a command line, is fully supported by dB/Compiler except when the asterisk is immediately followed by a backslash ('\'). (Please see Appendix B.)

Expressions

The CP/M 80 version of dB/Compiler only supports expressions of less than one hundred (100) characters.

APPENDIX D

Error Messages

(Some error messages are discussed in Chapter 2, Section G.)

Compile Time Error Messages

Syntax error

Macro substitution is not supported at this point

The number of index files must be between one and seven

"To" phrase expected

"Get" phrase expected

"From" phrase expected

"Extended" expected

"On" phrase expected

"blank" expected

"Houpdate" expected

"Additive" expected

"Secondary" or "primary" expected

The number of fields must be between one and five

"For" phrase expected

"With" phrase expected

file name or key expected

"Structure" expected

Null string not allowed

Boolean operator expected

Unknown command

Number expected

Expression expected

Identifier expected

Key expected

This parameter is always assumed OFF

Type ^C to abort or any other character to continue

If stack overflow

While stack overflow

Case stack overflow

Jump table overflow

Expression stack overflow

Only xx command files allowed in one compiling process

Only xx report files allowed in one compiled command file

Unbalanced endif

Unbalanced enddo

Unbalanced endcase

Can't open xxxxxx

Unbalanced if statement

Unbalanced case statement

Unbalanced while statement

Unbalanced text statement

Disk Full

Scope - record is not supported in locate command, please use go to

While expression is not supported in locate command

store error

The maximum number of index files is seven

The number of function parameters is wrong

Report form expression array overflow

Link Time Error Messages

xxxxxx is not a relocatable file, ABORT!

Usage: dbl [-i] [-l] [-dxy] [-oz] [-xab] filename

Disk Full

Only 256 files allowed in one linking process

read from xxxxxxxx failed

dBase/Compiler Reference Manual

Run Time Error Messages

Character string expected

File: xxxxxxx is currently open

No more variable space available

Only 64 variables allowed, the last one used is:
xxx

File: xxxxxxx already exists

File: xxxxxxx does not exist

Data item: xxxx does not exist

Macro must be a character string

Either macro substitution for expression or syntax
error or variable(s) undefined

Must operate on data type: xxxxx

Empty file name not allowed

Null skeleton

Editor stack overflow

Too many digits

Report stack overflow

Null drive name

You must use character string for file names, field

names or variable names

Error creating file: xxxxxx

Error opening file: xxxxxx

Error closing file: xxxxxxx

Error reading file: xxxxxxx

Error writing file: xxxxxxx

Error positioning in file: xxxxxxxx

Bad field name: xxxxx

Bad field type: xxxxx

Not a dBASE II file: xxxxxx

No such record in index: xxxxxx

Illegal key: xxxxxxx

No PRIMARY database in use

No SECONDARY database in use

No INDEX file in use

Creating from: xxxxxx would end up with more than 32
fields

Creating from: xxxx would end up with more than 1000
bytes per record

Index SYNC error:

Maximum allowed number of records reached:

FIELD not found: xxxxxxx

APPENDIX E

dB/Compiler Defaults

Compilation Defaults

Command Form: DBC <file>

Only <file> is compiled, along with any invoked .FRM files.

Default drive specifier is logged drive.

Default filename extension is .CMD (CP/M 80) or .PRG (MS-DOS).

Default object code destination is drive <file> is on.

Compilation pauses if an error condition is displayed.

Line count is displayed.

Command Form: DBC -A <file>

As above, but:

<file> and any .CMD (or .PRG), .FMT or .FRM files it calls are compiled. (CP/M 80: up to 64 .CMD, and up to 32 .FRM files. MS-DOS: up to 256 .PRG, and up to 32 format files.)

File extensions not specified in file names in DO, SET FORMAT TO, and REPORT FORM statements are assumed to be, respectively, .CMD (or .PRG), .FMT, and .FRM.

Linking Defaults

Command Form: DBL<version> <file>

Extension of <file> is assumed to be .CMD or .PRG.

Drive of <file> is assumed to be logged drive.

Default destination of linker product files is drive <file> resides on.

At run time files can reside together on any one drive.

CP/M 80: 54K RAM (run time) linker library, DBL80.LIB, used (56 RAM (run time) option is provided via DBL80L.LIB.

Linker library (with extension .LIB) must reside on drive <file> is on during linking.

Up to 256 object code files can be linked in any one pass of linker.

Default terminal attributes of CP/M 80 linkers are Tele-Video 950; attributes are changeable with INSTALL.COM included on dB/Compiler operating and sample disks.

Default terminal attributes of a linked application are the current attributes of the linker itself; attributes of the application are changeable with install program produced by linker.

Run Time Defaults

Application may be on any drive as long as the files required for execution are together.

APPENDIX F

dBASE II Bugs Fixed

Full Screen Editor Bugs

1) When a ^B is entered in dBASE II while in edit mode after a READ statement, strange and unpredictable things happen to the cursor and data. This is corrected.

2) Sometimes following a second READ statement, the screen's cursor location is extremely unpredictable. This has been fixed.

3) When a memory variable containing a number longer than ten digits is saved and restored, that variable can be evaluated incorrectly by dBASE II. For example:

```
.STORE 1234567891.2334 TO A
1234567891.0000
.SAVE TO B:JJ
.RESTORE FROM B:JJ
.LIST MEMORY
A          .(N)    1234567891.0000
** TOTAL **      01 VARIABLES USED  00007 BYTES USED
.? A
*****67891
```

This contradiction is resolved in compiled programs.

4) Sometimes the @ . . . GET combination does not work correctly in dBASE II if it is used on a memory variable recently RESTORED from storage. This seems to be a problem in the RESTORE or SAVE command, and is corrected in a compiled program.

5) A dBASE II SORT of a file where every record has been marked for deletion (but not yet PACKed) can yield an output file full of garbage. The equivalent dB/Compiler output file would be empty.

6) dBASE II does not produce consistent results when it is told to SUM a field in a file containing no records. Sometimes it will return a value other than zero. Results seem to depend upon the commands previous to the SUM. dB/Compiler will always return a value of zero in this situation.

7) dBASE II does not handle INDEX updates correctly. Consider the following excerpt of dBASE II code:

```
USE XM INDEX YY.
COUNT TO CNT
DO WHILE CNT>0
GO BOTTOM
REPLACE ZZ WITH a
STORE CNT-1 TO CNT
END0
```

where XX is any data file name, YY is its index file, constructed using a field named ZZ as an index key. We assume the programmer knows in advance the smallest value ZZ can hold, and that is a, above.

This program segment tests the index updates. The correct result should be ZZ in XX containing a, since all ZZ's are equal. The sequence in INDEX YY should be based on record numbers. A consecutive LIST command should list the file in record number (1) order.

dBASE II sometimes crashes when a data-base becomes larger than about 10K and an operation such as this is attempted. Even when it does not crash, the resultant data sequencing is incorrect. This is also true with a

mirror image program segment that goes to the top of the file and replaces the index key field with the largest possible value.

In other words, index updating using dBASE II is unreliable because of this serious bug. This situation is corrected in dB/Compiler.

8) The record number operator # does not always evaluate correctly in dBASE II. For example, if you are at the third record in a data file with one hundred records, and issue the command

SKIP 3000

dBASE II will generate the appropriate error message (RECORD OUT OF RANGE), however, the # function returns the wrong result (in this case, 3003). This bug is fixed in dB/Compiler. If an attempt is made to access a record out of range, in all cases the dB/Compiler # will return the record number of the last record in the file.

In addition, the current record is sometimes undefined under dBASE II execution (following a COPY command, for example). This can be observed by entering a DISPLAY command immediately following the COPY command. The record number usually indicates the last record in the file, however what is displayed is not necessarily the last record. This condition is also rectified by dB/Compiler.

INDEX

\$ 3A-1, 3B-2

& 3F-1

@... GET 4F-1

!, dBase II 4F-3

^B 4F-1

*\ 3A-1, 4B-2

-A 3A-1, 3A-2, 3B-3

Abort, prevention of 3E-1

Amperand 3F-1

see also, macro-substitution

Amperand, macro-substitution of 3F-2, 3F-3

ANSI terminals, 2B-2

Application installation 2B-1

Arrays, creation of 3F-1

Array, multi-dimensional 3F-2, 3F-12

Assumptions

made by compiler 3B-3

on application source code 1C-1

Automatic mode, compilation using, 3A-1, 3A-2, 3B-3

Bugs, dBase II 4F-1

Calculations 4B-1, 4B-2

CHR(255) 4B-2

Commands, dBase II: see individual listings

Comments 3E-5

Compiled applications

size 3C-4

advantages 1A-1

Compiler installation 2A-1

Compiler options 3A-2, 3A-3, 3B-3 — 3B-7

automatic compiling (-A) 3A-1, 3B-3

aborting compilation (^C) 3A-2, 3B-1

error condition override (-P) 3A-2, 3B-2, 3B-6

line count off (-V) 3A-2, 3B-7

object code destination (-O) 3B-6

Compiler system requirements 1B-1

Compiling an application 3A-1, 3B-1, 3F-33

CONFIG.SYS, 2B-2

Concatenated fields 3E-4, 4C-3

Concatenating strings 3F-21

- Conditional structures, nested 3E-5, 3F-6
- Copy protection, 3A-1, 3I-5
- Cross-environment linking 3C-1, 3D-1
- Cross-linking, 3C-1, 3D-1
- dbase II bugs
 - fixed 4F-1
 - full screen editing 4F-1, 4F-2
- dbase II co-products 3H-1
- dbase II, restrictions on 3E-1 — 3E-7, 4C-1 — 4C-4
- DCC 3A-1, 3B-1,
- db/Compiler
 - assumptions on use with 1C-1, 3B-3
 - options
 - see compiler options
 - see linker options
 - philosophy behind 1A-1 — 1A-8
 - differences from dbase II: see Differences
- DCL 3A-2, 3C-2
- dCode 3B-8
- Defaults
 - compiler 4E-1
 - run time 4E-2
 - linker 4E-2
- Demonstration disk 1D-1
- Design Strategies 3F-1 to 3F-36
 - for speed 3F-34
 - for macro-substitution 3F-1
 - for compiling and linking 3F-32 — 3F-33
- Differences, db/Compiler and dbase II
 - macro-substitution evaluation 3F-3, 3F-4
 - semantic 4B-1 — 4B-7
 - syntax 3F-36
- Disk full 3G-2, 3C-10
- Disk files, list 4A-1
- CO FILE 3B-4, 3E-4
- Drive allocation 3C-6
- Drive specifiers, during compilation 3B-4
- Dynamic file access, macro-substitutions in 3F-4
- Error condition 3B-5, 3G-1
- Error detection 3G-1

- Error messages
 - discussion of
 - compiler 3G-1

- discussion of
 - linker 3G-2
 - run time 3G-3
- listed
 - compiler 4D-1 — 4D-3
 - linker 4D-3
 - run time 4D-4

- ESCAPE 4B-3
- Executable code 1A-5, 3C-9, 3C-10

- Fields, concatenated 4B-6, 4C-3, 3E-4
- Files

- library files (.LIB) 3D-2
- maximum number allowed for compiling 3D-3
- sizes 3C-4
- random access 3F-4

- File extensions 3A-1, 3A-4, 3A-4, 3B-7

- File names
 - version differences 3C-2, 3D-2, 4A-1,
 - truncated 3B-7
 - extensions 3B-4, 3B-7
 - macro-substitution of 3F-4, 3F-15, 3E-3, 4C-3

- GET 3G-3

- Host system 1B-1, 2B-1

- I 3C-2
- INDEX 3G-4
- INDEX keys 3E-4, 4C-3, 3G-4
- INDEX updates, dbase II 4F-2, 4F-3
- Input, user 3F-5, 3F-34

- Installation
 - db/Compiler 2A-1
 - linker library 2A-1
 - finished application 2B-1
 - UPGRADE procedure (dbase II 2.3) 2C-1
 - CONFIG (CONFIG.SYS) 2A-2

INSTALL.COM 2A-1, 3A-1
 INT function 4B-4

Jump table overflow 3C-2

Keywords
 macro-substitution of 3E-3, 4C-3
 syntax when using 3E-36

Linker
 capacity 3C-2
 installation 2A-1
 invoking 3A-3, 3C-1 — 3C-10
 large library (CP/M 80) 3C-4
 library 3A-1, 3A-4
 product of 3C-2

Linker options
 drive specification; .OV1, .OV2 (-D) 3A-3, 3B-6
 drive specification; .COM, INSTALL, .OV1 (-D) 3A-3
 runtime drive specification; .OV1, .OV2 (-X) 3A-3
 56K memory option (-L) 3A-3, 3C-4
 non-executable code option (-I) 3A-3, 3C-5
 -Db 3A-4, 3C-7
 -Ob 3A-4, 3C-7
 -L Large memory 3A-4, 3C-4

Macro-substitution 3E-1 — 3E-20
 arrays 3E-2 — 3E-3, 3E-12 — 3E-13
 command file names 3E-5, 3E-15
 expressions 3E-6, 3E-16 — 3E-18, 3E-19 — 3E-31
 keyword 3E-8, 3E-9, 3E-10, 3E-11
 layered 3E-4, 3E-14
 restrictions on 3E-3 — 3E-4, 4C-3
 numeric device 3E-5

Native linker 3C-1

Object code, relocatable 3B-7
 options,
 linker, see linker options
 compiler, see compiler options
 Overlay files, purpose 3C-2

-P 3B-6
 PICK 4E-2
 Performance maximization, 3E-34
 Parsing 3I-3
 PIP 3B-4, 3B-5
 Printer usage 3B-6
 Programming strategies 3E-1
 Pseudo-code 3B-8

REND 3G-3
 Relocatable object code 3B-7
 REPORT 3B-4, 3G-4
 Restrictions on dBASE II 3E-1 — 3E-7, 4C-1 — 4C-4
 RETURN 4B-5
 Requirements: see System requirements

Semantic differences, appendix 4B-1 — 4B-18
 Semicolon 4B-6
 SET FORMAT TO 3B-4
 Space allocation 3C-6
 see also: linker options, compiler options
 SORT 3C-1, 3C-10, 3C-4, 4B-6
 SORT keys 3E-4, 4C-3
 Speed 3E-1, 3E-34
 Strategies, see Design strategies
 SUM 4E-2
 Syntax differences,
 spaces in command lines, 3C-1
 System requirements
 Host 1B-1
 Target 1B-1

Target modules 3C-1, 3D-1
 Target system 1B-1

UPGRADE procedure 3A-2
 User input 3E-5, 3E-34

VAL 4B-6, 4B-7
 Variable names, illegal 3E-4
 Virtual data machine 3B-8