Data General

Customer Documentation

Learning Business BASIC

Learning Business BASIC

093-000684-00

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000684

Printed in the United States of America
Rev. 00, December 1989

Notice

Learning Business BASIC
093-000684-00

Revision History:

Original Release - December, 1989

Effective with:

Business BASIC for AViiON™ Systems, Rev. 1.00
Business BASIC for 386/ix™ Systems, Rev. 1.00
AOS Business BASIC, Rev. 4.20
AOS/VS Business BASIC, Rev. 5.10
RDOS Business BASIC, Rev. 8.20
DG/RDOS Business BASIC, Rev. 8.21

Preface

Scope

Data General's Business BASIC runs on the following operating systems: mapped ECLIPSE® RDOS, DG/RDOS, AOS, AOS/VS, AOS/VS II, the DG/UX™ operating system, and INTERACTIVE Systems Corporation's 386/ix™ operating system. In most instances, this guide uses the term *DG/RDOS* to refer to RDOS and DG/RDOS, and the term *AOS/VS* to refer to AOS, AOS/VS, and AOS/VS II. Of course, when there are differences between related operating systems that will affect your programming, the guide will point out those differences.

UNIX® is a registered trademark of AT&T. However, since the 386/ix software product and the DG/UX software product have been derived from, or relate to, or work in conjunction with UNIX software, these two software products are sometimes referred to herein as UNIX products, solely for the purpose of improved readability. This liberty is taken in this manual only where 386/ix and UNIX, or DG/UX and UNIX, have areas of commonality or overlap, and not where the differences between them are significant.

This manual is for experienced programmers who have not used Business BASIC. The purpose of the manual is to acquaint these programmers with Business BASIC operations and programming procedures. The manual provides an overview of what commands, statements, functions, subroutines, and utilities are available to help programmers. It is not intended to provide detailed instructions on how these features work. Explanations of these Business BASIC features are contained in the Business BASIC reference manuals. This manual also discusses the file and database structures that Business BASIC supports.

Organization of This Manual

This manual comprises six chapters, one appendix, and a glossary.

Chapter 1 presents an overview of the Business BASIC software package and information on the Business BASIC modes of operation.

Chapter 2 discusses briefly how to write, execute, debug, and document a Business BASIC program.

Chapter 3 contains information on Business BASIC variables, expressions, and arithmetic operations.

Chapter 4 discusses Business BASIC subroutines (both those supplied with the system and those you write yourself), assembly-language subroutines, and Business BASIC utilities.

Chapter 5 presents an overview of the Business BASIC file structures and the language's input and output operations.

Chapter 6 contains information about the two Business BASIC database structures.

Appendix A contains sample programs that demonstrate how to set up and use logical and PARAM database structures.

The glossary contains definitions of key terms that you may come across in your work in Business BASIC.

Document Set

Learning Business BASIC is part of a four-manual set that describes the language, its subroutines and utilities, and how the system is set up. The other manuals in this set are

- *Commands, Statements, and Functions in Business BASIC* (093-000351)
- *Subroutines, Utilities, and the Business BASIC CLI* (093-000389)
- A manual that explains how to use Business BASIC on your operating system. If you are working on a UNIX system, you need the book *Using Business BASIC on DG/UX and 386/ix Systems* (093-000685)

Reader, Please Note

Data General manuals use certain symbols and styles of type to help clarify the information they contain. The Data General symbol and typeface conventions used in this manual are defined below. You should familiarize yourself with these conventions before reading the manual.

This manual also presumes the following meanings for the terms "command line," "format line," and "syntax line." A command line is an example of a command string that you should type verbatim; it is preceded by a system prompt. A format line shows how to structure a command; it shows the variables that must be supplied and the available options. A syntax line is a fragment of program code that shows how to use a particular routine; some syntax lines contain variables.

Convention	Meaning
boldface	In command lines and format lines: Indicates text (including punctuation) that you type verbatim from your keyboard. The names of all commands, statements, functions, subroutines, utilities, files, and directories are also printed in boldface.
`constant width font`	Represents a system response on your screen. Syntax lines also use this font.

italic	In format lines: Represents variables for which you supply values, for example, the names of your directories and files, your username and password, and possible arguments to commands.
[ ]	In format lines: These brackets surround an optional argument. Don't type the brackets; they only set off what is optional.
...	In format lines and syntax lines: Means you can repeat the preceding argument as many times as desired.

Contacting Data General

- If you have comments on this manual, please use the prepaid Comment Form that appears at the back. We want to know what you like and dislike about this manual.
- If you require additional manuals, please use the enclosed TIPS order form (USA only) or contact your local Data General sales representative.

Telephone Assistance

If you are unable to solve a problem using any manual you received with your software, and you are within the United States or Canada, contact the Data General Service Center by calling 1-800-DG-HELPS for toll-free telephone support. The center will put you in touch with a member of Data General's telephone assistance staff who can answer your questions.

Free telephone assistance is available with your warranty and with most Data General service options. Lines are open from 8:30 A.M. to 8:30 P.M., Eastern Time, Monday through Friday.

For telephone assistance outside the United States or Canada, ask your Data General sales representative for the appropriate phone number.

End of Preface

Contents

Chapter 1—Introduction to Business BASIC

Chapter 2—Program Development

Chapter 3—Numeric and String Variables

Chapter 4—Subroutines and Utilities

Chapter 5—Business BASIC Files

Chapter 6—Database Structures in Business BASIC

Appendix A—Sample Programs

Glossary

Index

Tables

Table

Figures

Figure

Chapter 1
Introduction to Business BASIC

Business BASIC is a version of the BASIC programming language that contains most standard BASIC commands, statements, and functions plus specialized statements and functions for handling file access, controlling the format of data, and performing system tasks. Business BASIC supports simple file structures and database file structures. These file structures include operating-system files, linked-available-record files, ISAM files, logical files, and a subfile-master file system. On AOS/VS systems, Business BASIC also includes an interface to Data General's INFOS® II file system.

This chapter provides an overview of the structure and composition of the Business BASIC software package. It also presents some general information on Business BASIC.

The major components of the Business BASIC package are shown in Figure 1-1

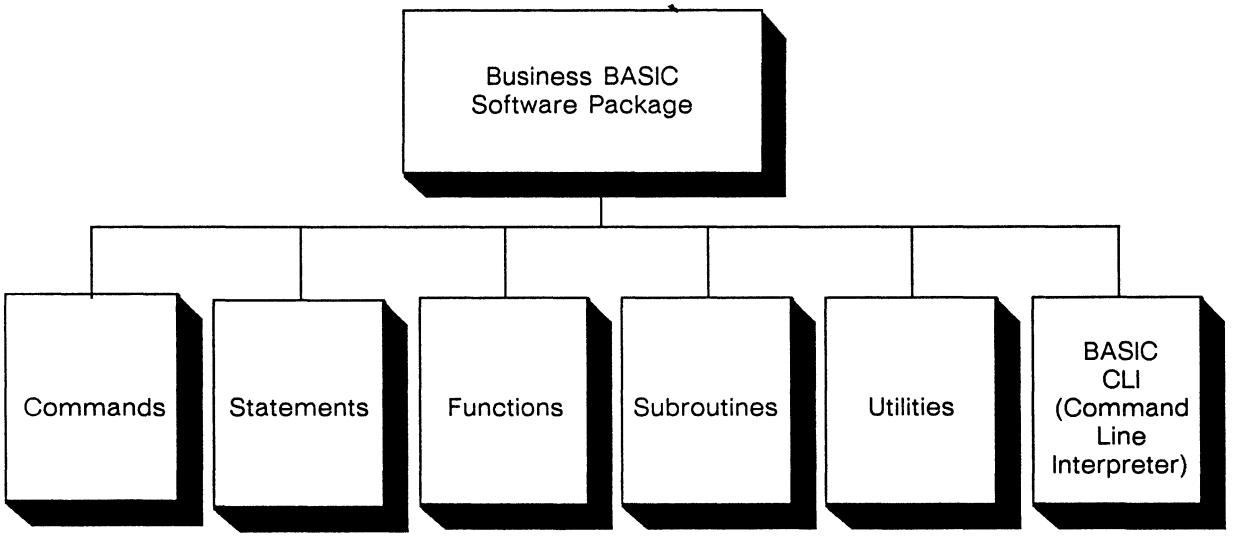

Figure 1-1 Components of the Business BASIC Software Package

The boxes in Figure 1-1 represent tools that you can use when you program with Business BASIC. These tools can be summarized as follows:

1. The three boxes at the left (Commands, Statements, Functions) make up the Business BASIC language.
2. The next two boxes represent the subroutines and utility programs provided by Business BASIC. The subroutines and utilities perform a variety of tasks to help you use Business BASIC.

3. The BASIC Command Line Interpreter (CLI) gives you capabilities similar to those offered by the DG/RDOS CLI, the AOS/VS CLI, and the UNIX® shell. It lets you perform operating system functions without leaving Business BASIC.

Commands, Statements, and Functions

The commands, statements, and functions that make up the Business BASIC language are the fundamental tools you use in developing application programs. By definition, a command is an instruction that is entered without a line number and is executed immediately. A statement is preceded by a line number and is not executed until you enter a command such as **RUN**. A function evaluates to either a numeric or string value. It is often used as part of a statement or command.

In most cases, statements and functions use arguments. These can include variables, numeric and string assignments, messages to print, and subroutine destinations.

Business BASIC supports two types of variables, numeric and string, and also arrays of numbers and strings. (String arrays are available only on UNIX systems.) Numeric variables must contain integers; you cannot assign a floating-point number to a variable or an element in a numeric array. String variables contain character data. You must declare the length (or dimension) of a string variable before you assign a value to it. More information on variables appears in Chapter 3.

For a summary of the commands, statements, and functions that are included in your Business BASIC software package, see the manual *Commands, Statements, and Functions in Business BASIC*.

Business BASIC Subroutines

The Business BASIC software package also contains prewritten subroutines. The subroutines are specialized portions of Business BASIC code whose modular design makes them easy to incorporate in application programs. The Business BASIC subroutines help you meet the processing needs of your programs and reduce the amount of coding you need to do.

The subroutines included in the software package reside in the Business BASIC system library directory. Their filenames have .SL extensions to distinguish them from utility programs and other files. You execute a subroutine from within a Business BASIC program with a **GOSUB** *line-number* statement. The line number is the entry point to the subroutine.

You can also write your own subroutines and place them in the library directory so that you can reuse them. In addition, on AOS/VS and DG/RDOS systems, Business BASIC provides an interface that allows you to use assembly-language subroutines.

Subroutines are discussed further in Chapter 4. Explanations of individual subroutines are contained in the manual *Subroutines, Utilities, and the Business BASIC CLI*.

Business BASIC Utility Programs

Utilities are Business BASIC programs that perform data processing functions, such as formatting screens and maintaining your database.

Most utilities are stand-alone programs. You can execute them by using the **RUN**, **CHAIN**, or **SWAP** command, or you can start them through the BASIC CLI by entering the utility name preceded by an exclamation point (!). Some utilities, however, have restricted execution modes.

The utilities are discussed further in Chapter 4. For explanations of how individual utilities work, see the manual *Subroutines, Utilities, and the Business BASIC CLI.*

BASIC CLI

Business BASIC also has its own Command Line Interpreter (CLI). The BASIC CLI gives you capabilities similar to those provided by the DG/RDOS CLI, the AOS/VS CLI, and the UNIX shell without forcing you to leave Business BASIC. This means that you do not need to exit Business BASIC to create and delete files, to move files between directories, or to print files.

You start the BASIC CLI by entering **RUN "CLI**, **CHAIN "CLI**, **SWAP "CLI** or **!CLI**. An exclamation point (!) prompt indicates that you are in the BASIC CLI.

There are three ways to execute a BASIC CLI command:

- Start the BASIC CLI (which puts you in BASIC CLI mode), and then enter the command.
- At the asterisk prompt, enter the command preceded by an exclamation point: !*command*.
- Have your program store a BASIC CLI command in a string, move that string to an area of memory reserved for passing information between programs (the common area), and then swap to the BASIC CLI.

You can also execute programs and utilities through the BASIC CLI by using one of these three methods. The BASIC CLI swaps to the file you specify when it does not recognize a command.

To leave the BASIC CLI, use the **POP** or the **QUIT** command, both of which return you to your previous level, or enter **BYE**, which logs you out of Business BASIC.

For more information on the BASIC CLI, see the manual *Subroutines, Utilities, and the Business BASIC CLI.*

Business BASIC Operational Modes

To execute Business BASIC programs, you enter a **RUN**, **SWAP**, or **CHAIN** command followed by the appropriate argument. You can execute programs from keyboard mode, BASIC CLI mode, or program mode.

Keyboard mode is indicated by an asterisk (*) prompt while BASIC CLI mode is indicated by an exclamation point (!) prompt. You can execute a program using the BASIC CLI while you are in keyboard mode by entering the program name preceded by an exclamation point (that is, !*program-name*), or you can execute the BASIC CLI and then type in the program name. Program mode begins when you execute a program and then use it to perform tasks, such as executing another program. You can enter program mode by executing a program from keyboard mode or from BASIC CLI mode. The primary operational mode when you are working in Business BASIC is keyboard mode. You automatically enter keyboard mode when you execute the interpreter or, on DG/RDOS systems, when you log in to Business BASIC. While in keyboard mode, you can execute commands, or you can create and run Business BASIC programs (unless you are working on a run-only Business BASIC system).

Everything you type while in keyboard mode goes into an assigned buffer area called working storage. This is the area of the computer's memory that holds your program and data. Anytime you use the **ENTER** command to bring the text of a program in from a disk file or the **LOAD** command to load the metacode version of a program, that program is stored in working storage in binary format.

When a program is in working storage, you can execute it by entering the **RUN** command with no arguments or with a line number. You can also execute a program by entering **RUN** "*program-name*. In addition, you can use the command **SWAP** "*program-name*, **CHAIN** "*program-name*, or **CON** (continue).

File Structures

Business BASIC supports the following categories of files:

- Operating system files.
- Business BASIC files, that is, operating system files containing an embedded Business BASIC structure.
- INFOS II files, that is, operating system files containing an embedded INFOS II structure (AOS/VS only).

With Business BASIC running, you can access any operating-system data files, except DG/RDOS sequential files, by using direct random access or sequential access. You must use sequential access with DG/RDOS sequential files. For information on the structure of the files your operating system creates, see the documentation for your operating system.

Business BASIC supports two additional file structures to increase your flexibility in working with data. These files are operating system files containing an embedded Business BASIC structure that tells the BASIC interpreter how the files are organized. The two Business BASIC file types are

- Linked-available-record files. These files allow dynamic record allocation and allow you to access the records directly.
- Index files. Business BASIC uses the indexed sequential access method (ISAM) of working with index files.

On AOS/VS systems, Business BASIC also provides an interface to the INFOS II file management system. INFOS II files are operating system files that contain an embedded INFOS II structure. The INFOS II file system provides data handling capabilities that let you create, maintain, and use many types of databases in batch and multiterminal environments.

In addition to the file structures mentioned above, Business BASIC supports two database structures for working with files: the logical file database structure and the PARAM file database structure. These structures increase the number of files you can open within a program by allowing you to use subsections of a physical file. These subsections are called logical files in the logical structure and subfiles in the PARAM structure. Since the operating system does not recognize subsections of files, the database structures catalog the subsections so that the Business BASIC system can use them.

End of Chapter

Chapter 2
Program Development

This chapter uses examples to show you the Business BASIC program development phases. The examples are simple so that you can begin working in Business BASIC quickly and at the same time extrapolate the information you need to develop application programs. To perform these examples, you need to be running Business BASIC and, unless otherwise specified, be in keyboard mode (indicated by the asterisk prompt).

Programming with Business BASIC consists of four steps:

1. Writing the program.
2. Executing the program.
3. Interrupting and debugging the program.
4. Documenting and storing the program.

These steps are discussed in this chapter.

Writing Business BASIC Programs

Because Business BASIC is an interpreter, not a compiler, executing a single program generally requires two main steps:

1. Getting the program into working storage.
2. Entering the command, such as **RUN**, that begins program execution.

The program can be an existing program that is stored elsewhere (including one written in an editor) or one that you created by typing in program statements while in keyboard mode. Each line you type in while in keyboard mode is stored in working storage and added to the lines already there.

Before you write a program in keyboard mode, enter the **NEW** command. This clears working storage.

To create program statements, type in both the statement line number and the statement contents; then press the New Line or Enter key. Each time you type in a program statement, the Business BASIC interpreter checks the syntax of the line and reports any errors. If there are none, the system adds the statement to the others in working storage. For information about how large a Business BASIC program can be, see the manual that explains how to use the language on your operating system.

The following is a four-line Business BASIC program that multiplies 3 by 300. Type in the program and execute it by entering the command **RUN**. Business BASIC displays the result on your screen.

```
* NEW
* 10 LET A = 300
* 20 LET B = 3
* 30 PRINT A*B
* 40 END
* RUN
 900

*
```

You do not have to use an **END** statement in Business BASIC programs. Normal program execution halts when Business BASIC encounters either the last line of code or an **END/STOP** statement. However, since Business BASIC may execute all the program statements in working storage, using an **END** statement prevents the system from executing lines left in working storage by an earlier program. (You can also avoid this problem by clearing working storage with a **NEW** command before placing your program in it.)

The program remains in working storage until you replace it with another program, clear working storage with the **NEW** command, or log out of Business BASIC.

Figure 2–1 illustrates the general command format you use for the statements in your program. This syntax is the same for almost all lines of Business BASIC code.

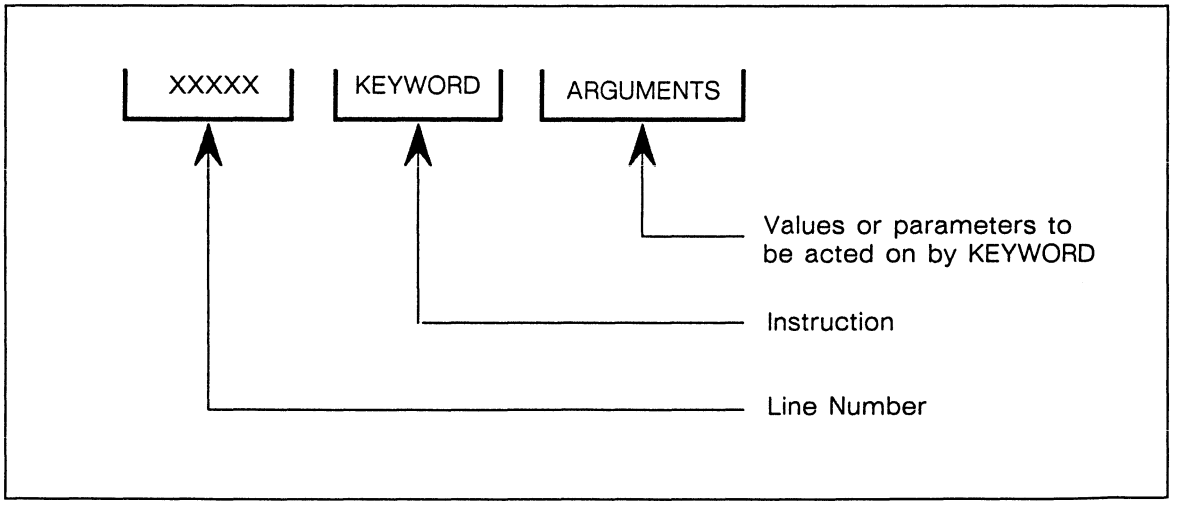

Figure 2–1 Business BASIC Program Statement Format

1. XXXXX (LINE NUMBER). Start each line of code with a line number in the range 1 to the maximum number of lines that your operating system allows. To find this number, consult the manual that explains how to use Business BASIC on your operating system. Business BASIC left-pads all line numbers containing fewer than the maximum number of digits with zeros when you list the program.

2. KEYWORD. Keywords are the instructions that tell the program to perform actions. The program you just executed contains four keywords: two **LET** statements, one **PRINT** statement, and an **END** statement.

3. ARGUMENTS. Arguments (also known as parameters) often follow statements. Arguments can include variables, numeric or string assignments, messages to print, and subroutine destinations. For example, line 10 of the program you just executed contained the argument A=300.

Working storage holds the values you assign to variables as well as the program. At this point the variable A has a value of 300, which was assigned when the program was executed. To check this, enter **PRINT A**; the value 300 appears next to the A:

```
* PRINT A 300

*
```

To see the program contents of working storage, use the **LIST** command:

```
* LIST
00010 LET A=300
00020 LET B=3
00030 PRINT A*B
00040 END

*
```

You could have entered program statements 10 and 20 as

```
* 10 A = 300
* 20 B = 3
```

because the Business BASIC interpreter inserts optional keywords, such as **LET**, and removes extra spaces (A = 300). Business BASIC also left-pads any line numbers that are part of your program statements and, in some cases, inserts spaces. Any omitted keywords or zeros are displayed when you list the program. These characters are also included when Business BASIC calculates the length of a line. The maximum number of characters the **LIST** command permits is dependent on your operating system, so to find this maximum, refer to the manual that explains how to use Business BASIC on your operating system.

CAUTION: *It is possible that Business BASIC will expand a line so that it contains more than the maximum number of characters allowed on a line, but does not produce an error. In such a case, you can run your program; however, if you use the LIST command to display your program or write your program to a listing file, the line will be truncated.*

Similarly, when you use the **ENTER** command to load a Business BASIC program, the number of characters in a line cannot exceed the maximum referred to above. If you enter a program with longer lines, you receive the message `Error 18 - Line too long`.

If you need to check a specific program line, you can list just that line:

```
* LIST 30
00030 PRINT A*B

*
```

You can also use **LIST** to display a range of line numbers. For example, the command line **LIST 10,20** would cause the interpreter to display lines 10 and 20. **LIST** is discussed further in the manual *Commands, Statements, and Functions in Business BASIC*.

Adding Program Statements

You can add a statement to a program in working storage by typing in the new statement. The system adds the statement sequentially. Be sure to give the new statement a unique line number; otherwise, it overwrites any statement with the same line number.

In the program you now have in working storage, you can add a statement to change the value of A (currently 300). The value, however, does not change until you execute the program again.

```
* 35 A = A*B
* LIST
00010 LET A=300
00020 LET B=3
00030 PRINT A*B
00035 LET A=A*B
00040 END

* PRINT A 300

* RUN
 900

* PRINT A 900

*
```

Business BASIC Programming Features

When a program is run, Business BASIC executes the statement with the lowest line number and then proceeds to the next higher number, unless the statement directs the system elsewhere, as **GOTO** and **GOSUB** statements do.

The **GOTO** statement transfers control to a specific statement. The **GOSUB/RETURN** statements transfer program control to a segment of code (a subroutine) and then return you to the statement following the subroutine call. These statements, especially the **GOSUB/RETURN** statements, can increase the modular structure of your program, making it easier to debug. To make your program modules easy to identify, you can include a **REM** (remark) statement to describe them.

Business BASIC also provides flow-control constructions such as the **FOR/NEXT**, **DO/WHILE**, and **IF** statements. The **FOR/NEXT** statements allow program looping with the termination test occurring at the top of the loop. The testing in a loop set up with the **DO/WHILE** statements can occur at the top or bottom of the loop. The **IF** statement provides your program with decision-making capability by transferring program control based on the value of an expression or the logical answer to a relational expression.

In addition, Business BASIC is structured so that you can use its input/output capabilities to make your programs interactive. You have already used the **PRINT** statement to display information at your terminal. The **INPUT** statement lets you enter data from a terminal. (Use the **INPUT FILE** statement to enter data from a file.) When you use the **INPUT** statement, you can supersede its question mark (?) prompt by placing a string after the keyword.

To see a simple example of how the preceding statements can be used in a program, refer to the example below:

```
10 REM *** USE THIS PROGRAM TO BALANCE YOUR CHECKBOOK.
20 PRINT "CHECK BALANCING PROGRAM"
30 PRINT @(-28);"ENTER ALL NUMBERS WITHOUT DECIMAL POINTS."
40 INPUT @(-28),"Enter your current balance: ",A
50 DIM B$[1]
60 INPUT @(-28),"Enter 'C' (check), 'D' (deposit), 'S' (stop): ",B$
70 STMA 14,B$,0
80 IF B$<>"C" AND B$<>"D" AND B$<>"S" THEN
90   PRINT @(-28);"INVALID INPUT"
100   GOTO 60
110 END IF
120 IF B$="S" THEN GOTO 220
130 IF B$="C" THEN GOSUB 230
140 IF B$="D" THEN GOSUB 270
150 PRINT @(-28);"Your new balance is $",A
160 IF A<0 THEN
170   FOR I=1 TO 3
180     PRINT "Your account is OVERDRAWN!"
190   NEXT I
200 END IF
210 GOTO 60
220 END
230 REM *** SUBTRACT THE AMOUNT OF THE CHECK.
240 INPUT @(-28),"Enter the amount of the check: ",C
250 LET A=A-C
260 RETURN
270 REM *** ADD THE AMOUNT OF A DEPOSIT.
280 INPUT @(-28),"Enter the amount of the deposit: ",D
290 LET A=A+D
300 RETURN
```

This program uses **INPUT** statements to display prompts and get values, the **IF** statement to decide where to transfer program control, the **GOTO** and **GOSUB** statements to transfer program control, the **REM** statement to identify the subroutines, and the **FOR/NEXT** statements to print a series of messages. When you list the program, Business BASIC pads all

the line numbers, appends the text of the **REM** statements to the subroutine calls, and sets the space indentations in the **FOR/NEXT** loop.

FOR/NEXT, **INPUT**, **GOTO**, **GOSUB/RETURN**, **LIST**, and **REM** are explained in the manual *Commands, Statements, and Functions in Business BASIC*.

Modifying Your Program

To modify a program in working storage, you can

- Replace a statement by typing in a new statement with the same line number or add a new statement by giving it a unique line number.
- Delete statements by using the **ERASE** command, by typing in a line number without any information following it, or by typing in either a range of line numbers (10,20) or a line number followed by a comma (20,), which removes all the lines from that number through the end of the program.
- Bring in statements from a listing file with the **ENTER** command.
- Use Business BASIC keyboard editing commands to change statements.
- Use your operating system's SCREENEDIT feature if your operating system supports one, or on UNIX systems, use Business BASIC's SCREENEDIT feature.

One way to change a program statement is to retype the line. Whenever Business BASIC encounters duplicate line numbers, it overwrites the existing statement with the new statement. This also happens when you use the **ENTER** command to bring a program or subroutine into working storage and a program is already there. As Business BASIC merges the new program statements with the current program, it checks the line numbers of both pieces of code. Each time it finds duplicate line numbers, it replaces the existing statement with the statement being brought into working storage.

As was mentioned above, you can also delete program statements. Business BASIC does not automatically renumber your program when you delete lines. However, you can restore even spacing between your line numbers by using the **RENUMBER** command. If you want to renumber only a section of your code, you use the **RENUM** utility.

Business BASIC provides several methods for changing an incorrect statement. If you type a character and then want to erase it, use the Del or Backspace key. And to tell the system to ignore the line you are typing, press an interrupt key. To do other editing, use the keyboard editing commands discussed below (DG/RDOS and AOS/VS only) or your system's SCREENEDIT feature (AOS/VS and UNIX systems only).

Keyboard Editing Commands

To modify the contents of program statements on DG/RDOS and AOS/VS systems, you can use the Business BASIC keyboard editing commands (the dot editor). These commands work with program statements in working storage. For you to use these commands, the statement you want to modify must be in the edit buffer, a special buffer that holds one line from working storage at a time. You place lines in the edit buffer in one of three ways:

- Any statement you enter that causes an error is automatically placed in the edit buffer by the Business BASIC interpreter.
- The last line displayed following a **LIST** command remains in the edit buffer.
- Any line specified by the **LIST** *line-number* command remains in the edit buffer.

The line stays in the edit buffer until you use a command to move it to working storage.

Most of the editing commands echo the edited line on your terminal but do not change the actual line in working storage. This lets you make additional changes to the line. When the line is correct, you must move it to working storage with the . (period) command. This command empties the edit buffer. The only editing command that does not work this way is the .C command, which always places the modified line in working storage and leaves the edit buffer empty.

Several of these commands use delimiters to separate the command from the text. The delimiter is always the first character to follow the command and must be used consistently in each command line. When a command requires more than one delimiter, the delimiter must be the same each place it is used. Unless otherwise specified, the delimiter can be anything that is not part of the string of text. Table 2-1 contains a summary of the keyboard editing commands.

Table 2-1 Keyboard Editing Commands

Command	Function
.(period)	Sends the line in the edit buffer to working storage.
.A *string*	Appends *string* to the line in the edit buffer. A space is frequently used as a delimiter with this command.
.C/*string1*/*string2*[/G]	Changes the first occurrence of *string1* to *string2*. If you include the /G switch, .C changes all occurrences of *string1* to *string2*. You cannot use a space as a delimiter with the .C command. The .C command passes the line to working storage.
.E/*string1*/*string2*[/G]	Changes the first occurrence of *string1* to *string2*. If you include the /G switch, .E changes all occurrences of *string1* to *string2*. You cannot use a space as a delimiter with the .E command.
.I *string*	Changes the entire line in the edit buffer to *string*. A space is frequently used as a delimiter with this command.
.P	Displays the contents of the edit buffer.

The **.P** command displays the line in the edit buffer; it does not affect the contents of the edit buffer or of working storage. To place a line in the edit buffer and then check it, type in the following commands:

```
* 10 REM *** INCREASE THE VALUE OF A

* LIST 10
00010 REM *** INCREASE THE VALUE OF A

* .P
00010 REM *** INCREASE THE VALUE OF A

*
```

The **.E** and **.C** commands change text in a program statement. The existing text (*string1*) and the new text (*string2*) are set off by delimiters. Both **.E** and **.C** accept any character as a delimiter except the space character.

The **.E** and **.C** commands differ in that **.C** places the changed line in working storage, while the **.E** command leaves the revised line in the edit buffer. With the **.E** command, the actual program statement is not changed unless you use the . command to move the new line to working storage. If you use the **.C** command, the Business BASIC interpreter checks the syntax of the line being passed to working storage. If the edited line causes an error, the system displays an error message and returns the line to the edit buffer.

Without the **/G** switch, both **.E** and **.C** change only the first occurrence of *string1* to *string2*. When you use the **/G** switch, all occurrences of *string1* are changed to *string2*.

To see how **.E** and **.C** work, first place a line in the edit buffer.

```
* LIST 10
00010 REM *** INCREASE THE VALUE OF A

*
```

Now use the **.E** command to change the line:

```
* .E/INCREASE/MODIFY
00010 REM *** MODIFY THE VALUE OF A

*
```

Use the **.P** command to view the contents of the edit buffer and then the **.** (period) command to move the line to working storage. You can use the **LIST** command to verify that the revised line has been moved to working storage.

```
* .P
00010 REM *** MODIFY THE VALUE OF A

* .
00010 REM *** MODIFY THE VALUE OF A

* LIST 10
00010 REM *** MODIFY THE VALUE OF A

*
```

If you had used the **.C** command instead of the **.E** command, you would have received the error message `ERROR 73 - Edit buffer is empty` when you used the **.P** command.

The keyboard commands **.A** and **.I** change lines instead of strings in a line. The **.A** command appends a string to the end of a line, and the **.I** command replaces the existing line with the line you type in. Both commands leave the new line in the edit buffer instead of placing it in working storage.

To change line 20 of an imaginary program by appending +A to it, use the **.A** command:

```
* LIST 20
00020 LET B=3

* .A +A
00020 LET B=3+A

* .
00020 LET B=3+A

*
```

With the **.I** command, you type in what you want to be the new program statement, including the line number (the edit buffer treats line numbers as characters in the line). The **.I** command does not echo the modified line on your terminal.

```
* LIST 20
00020 LET B=3+A

* .I 20 PRINT B*A

* .
20 PRINT B*A

*
```

In addition to the keyboard editing commands, Business BASIC for DG/RDOS and AOS/VS systems has an **EDIT** utility that you can use to create or edit a text file. To conserve memory, **EDIT** uses a disk-resident buffer to hold the lines being modified. The **EDIT** utility is explained in the manual *Subroutines, Utilities, and the Business BASIC CLI.*

SCREENEDIT Control Characters

If you are working in Business BASIC on an AOS/VS or UNIX system, you have access to a SCREENEDIT feature that allows you to use control characters to modify program statements and commands. The SCREENEDIT control characters and their functions are listed in Table 2–2 below:

Table 2–2 SCREENEDIT Control Characters

Control Character	Function
Ctrl-A	Moves the cursor to the end of the line.
Ctrl-B	Moves the cursor to the end of the previous word.
Ctrl-E	Turns insert mode on. Entering this character a second time turns insert mode off.
Ctrl-F	Moves the cursor to the beginning of the next word.
Ctrl-H	Moves the cursor to the beginning of the line.
Ctrl-K	Erases all characters to the right of the cursor.
Ctrl-U	Erases the entire line and moves the cursor to the beginning of the line.
Ctrl-X	Moves the cursor one character to the right.
Ctrl-Y	Moves the cursor one character to the left.

Saving Programs

Use either the **LIST** or **SAVE** command to save the contents of working storage. **LIST** by itself displays the contents of working storage at your terminal; **LIST** with a filename preceded by a double quotation mark creates a listing file (text file) that contains the program statements in working storage and is stored on disk. **SAVE** followed by a filename places the metacode version of your program in a disk file. The format lines for these commands are

LIST "*filename*

SAVE "*filename*

Use the **ENTER** command or statement to bring a listing file into working storage and the **LOAD** command to bring a saved file into working storage. (You can also execute a saved file directly by typing in **RUN** "*filename*, **CHAIN** "*filename*, **SWAP** "*filename*, or !*filename*.) Before you bring any program into working storage, clear working storage by executing a **NEW** command. The format lines for these commands are shown below.

NEW	(This clears working storage)
ENTER "*filename*	(This is a listing file)
LOAD "*filename*	(This is a saved file)

When you create a program in working storage or when a program completes its execution, it stays in working storage until one of the following commands or statements is executed:

- A **BYE** command or statement to log the user out of Business BASIC.
- A **NEW** command or statement to clear working storage.
- A **RUN** command to execute another program.
- A **CHAIN** command or statement to execute another program.
- A **LOAD** command to bring a new program into working storage.

If you use a **SWAP** command to execute another program, the system saves the current contents of working storage, brings the new program into working storage, executes the new program, clears the new program from working storage, and places the old contents back in working storage.

RUN, **CHAIN**, and **LOAD** clear working storage before bringing in the new program.

Executing Business BASIC Programs

When you create a program in keyboard mode or use the **ENTER** or **LOAD** command to bring a program into working storage, you can execute the program by typing

* **RUN**

The system clears all values currently assigned to variables and starts executing the program beginning with the lowest line number. The system stops executing the program if it encounters a **STOP** or **END** statement or the last statement in the program. It also stops executing the program if an error occurs, or if you press an interrupt key. If you resume execution of a program by entering **RUN** *line-number* or the command **CON** (continue), the system continues executing the program without clearing the values from the variables.

Other ways to execute programs include typing **SWAP** *"filename*, **CHAIN** *"filename*, and *!filename* (which uses the BASIC CLI). You can also execute a program by typing *"filename* without a command in front of it while in keyboard mode; this causes Business BASIC to swap to that file. In addition, your system manager can set up your system so that it automatically executes a program when you log in to Business BASIC.

Both **SWAP** and **CHAIN** can be executed in keyboard mode. Generally, however, they are program statements and are used by a program running in working storage to execute a program stored on disk. **SWAP** returns control to the program originally in working storage, but **CHAIN** does not. Figures 2-2 and 2-3 show the flow of program control resulting from the **SWAP** and **CHAIN** statements.

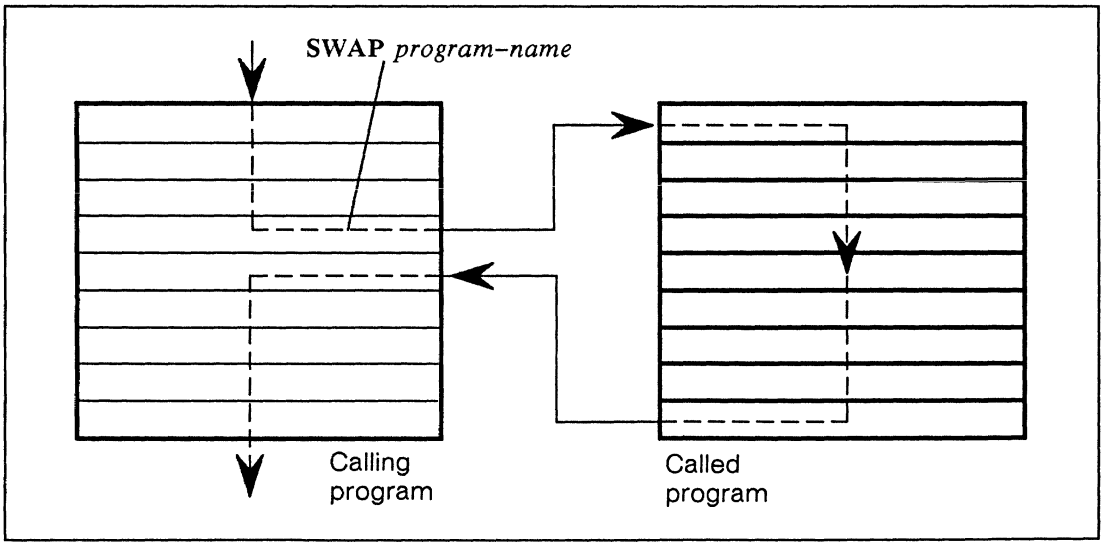

Figure 2–2 Flow of Program Control with SWAP Command

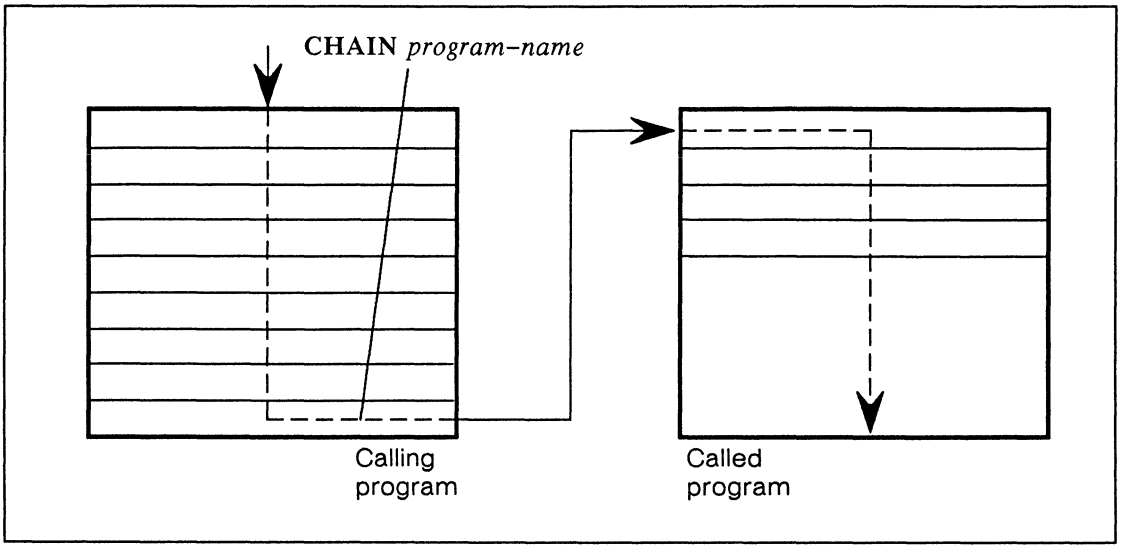

Figure 2-3 Flow of Program Control with CHAIN Command

Continuing Execution of a Program

Business BASIC lets you bring a program into working storage and execute it from the point where it last stopped. Do this by combining the **SWAP** or **CHAIN** command with the **CON** command: **SWAP THEN CON** or **CHAIN THEN CON**. These versions of **SWAP** and **CHAIN**, like the **RUN** command with a line number, do not clear values from variables when used to continue execution of a program.

Enter the following program:

```
* NEW
* 10 LET X=4
* 20 PRINT X
* 30 LET X=100
* 40 STOP
* 50 PRINT "X= ";X
* 60 END
```

Run the program. It stops at line 40:

```
* RUN
 4


STOP at 0040
*
```

Now save the program as **PROG2**:

```
* SAVE "PROG2
*
```

If you use **CON** to continue executing **PROG2**, the program displays X with a value of 100, which was assigned in line 30, and then ends. Instead of continuing **PROG2**, type in the following program that swaps to **PROG2**.

```
* 10 LET X=2
* 20 SWAP "PROG2" THEN CON
* 30 PRINT "BACK ALREADY!"
* 40 PRINT "X now = ";X
* 50 END
```

When executed, the new program assigns the value 2 to X and swaps to **PROG2**. **PROG2** indicates that the value of X is 100:

```
* RUN

X= 100
BACK ALREADY!
X now = 2

*
```

In this example, the **SWAP THEN CON** statement performs the following actions:

1. In most environments, it stores the calling program and its values, including the value 2 for X, in a swap file. On a UNIX system into which you have generated the in-memory-swap option, the interpreter stores the program and its values in memory.
2. It brings **PROG2** into working storage from disk.
3. It resumes execution of **PROG2** at the point where the program stopped before it was saved. In this case, **PROG2** stopped at line 40, so execution resumes with line 50. In addition, the variables in **PROG2** retain the values assigned to them during the last execution.
4. It returns the calling program to working storage when **PROG2** ends. The values assigned in the calling program are retained; they are not affected by any action **PROG2** took.

Therefore, the first time X is printed (by line 50 in **PROG2**), it equals 100, the value assigned to it the last time **PROG2** ran. The second time X is printed, it equals 2, the value assigned to it in the calling program.

To pass values for variables between a calling program and a program to which it swaps, you can use the **BLOCK WRITE** and **BLOCK READ** statements. These statements transfer data through the common area, a section of memory used solely for storing information (see Chapter 4). To transfer single-word values, use **STMA 1** and **STMA 2**. The manual *Commands, Statements, and Functions in Business BASIC* explains the **BLOCK READ**, **BLOCK WRITE**, **STMA 1**, and **STMA 2** statements.

Table 2-2 illustrates the differences between the program execution commands.

Table 2–3 Program Execution Commands

Command Format	Usage	Location of Executed Program	Program Kept in Working Storage after Execution	Variable Values Retained from Last Execution
RUN	Command	Working storage	Original	No
RUN *line-number*	Command	Working storage	Original	Yes
RUN "*program*	Command	Disk file	New	No
CHAIN "*program*	Program statement or command	Disk file	New	No
CHAIN...THEN GOTO...	Program statement or command	Disk file	New	Yes
CHAIN...THEN CON	Program statement or command	Disk file	New	Yes
SWAP "*program*	Program statement or command	Disk file	Original	No
SWAP...THEN GOTO...	Program statement or command	Disk file	Original	Yes
SWAP...THEN CON	Program statement or command	Disk file	Original	Yes
CON	Command	Working storage	Original	Yes

Interrupting and Debugging Programs

You can interrupt Business BASIC programs by pressing an interrupt key. Normally, when an interrupt occurs while a program is executing, the program halts and your terminal returns to keyboard mode. From keyboard mode, you can check the values assigned to your program variables. You can also assign new values to variables by entering a keyboard assignment command, such as **LET X=2**. Then you can execute the program from a line number to see what the program does with new values.

The **CON** command is a useful debugging aid. If your program stops for any reason (a **STOP** statement, an error, or an interrupt), you can use the **CON** command to continue execution at the next higher line. **CON** does not affect either the value of the variables or the status of files (that is, where the file pointers are or whether the files are open or closed).

Enter the following program. It causes an error because Z has not been assigned a value.

```
* NEW
* 10 LET X=2
* 20 LET Y=4
* 30 PRINT Z
* 40 PRINT "Z TIMES 10 EQUALS: ";Z*10
* 50 STOP
* RUN

Error 17 at 30 - Unassigned variable
*
```

To get around the error temporarily, assign a value to Z from the keyboard and enter the **CON** command:

```
* Z=X+Y
* CON
Z TIMES 10 EQUALS: 60

STOP at 50
*
```

CON started execution at line 40, the line after the one that caused the error. The program accepted the value you assigned Z in working storage and used it to calculate the value of Z times 10. However, if you run the program again, you get the following results:

```
* RUN

Error 17 at 30 - Unassigned variable
*
```

The error still exists in the program. This is true because **RUN** clears all values for variables, including the keyboard assignment for Z. To correct the program, type in a line 25, which assigns a value to Z within the program, and then run the program from line 25.

```
* 25 LET Z=X+Y
* RUN 25
 6
Z TIMES 10 EQUALS: 60

STOP at 50
*
```

Debugging Aids

Business BASIC provides several commands, statements, and utilities to help you debug programs. However, not all of them are available on all operating-system platforms. To determine which commands and statements are available to you and how to use them, consult *Commands, Statements, and Functions in Business BASIC*, and for utilities consult *Subroutines, Utilities, and the Business BASIC CLI*.

A complete list of the commands, statements, and utilities you can use for debugging purposes follows:

- The **TRACE ON** command or statement, which displays the line numbers of subsequent statements as they are executed.
- The **TRACE OFF** command or statement, which disables the tracking of line numbers as statements are executed.
- The **STEP** command, which executes the next statement of the program currently in memory.
- The **PROGRAM DISPLAY** command or **PD** utility, which provides information about a save file or the program in working storage.
- The **VAR RENAME** command or **VAR** utility, which renames program variables in a save file.
- The **SIZE** command or utility, which displays the size of the program in working storage.
- The **VAR DISPLAY** command, which lists the variables in a save file or a program in working storage.

Handling Interrupts Within Programs

Business BASIC also has two program statements to help you deal with interrupts and runtime errors: the **ON IKEY** statement and the **ON ERR** statement. If Business BASIC processes one of these statements before encountering an interrupt or an error, the statement dictates the action the system takes when an interrupt or error occurs.

The ON IKEY Statement

With the **ON IKEY** statement, you can trap any interrupts in your program. An **ON IKEY** statement has the following format:

ON IKEY THEN *statement*

The argument *statement* must be a valid Business BASIC statement excluding **FOR, NEXT, DATA, END, REM,** and **DEF**.

Here is the way **ON IKEY** works:

1. You place the **ON IKEY** statement in your program.
2. Business BASIC processes the **ON IKEY** statement.
3. After that, any time Business BASIC encounters an interrupt, it executes the statement that follows the keyword **THEN**.

When an interrupt occurs before an **ON IKEY** statement has been executed, program execution halts and Business BASIC returns you to keyboard mode.

You can suspend the interrupt condition by executing an **STMA 6,5**. This disables interrupts so that **ON IKEY** neither traps an interrupt nor halts the program. If an interrupt occurs after **STMA 6,5** has been set, Business BASIC sets **SYS(26)** to 1 to let you know that an interrupt occurred. You can restore interrupt handling by using **STMA 7,5** to re-enable the **ON IKEY** statement.

IKEY THEN INT cancels the previous **ON IKEY** statement and returns interrupt handling to Business BASIC.

If you interrupt an input/output statement, you risk losing data. You can continue the program, but **CON** starts execution at the next statement and does not resume interrupted input/output operations.

ON ERR Statement

The **ON ERR** statement traps errors in your program the same way **ON IKEY** traps interrupts. An **ON ERR** statement has the following format:

ON ERR THEN *statement*

The argument *statement* must be a valid Business BASIC statement excluding **FOR, NEXT, DATA, END, REM,** and **DEF**.

Here is the way **ON ERR** works:

1. You place the **ON ERR** statement in your program.
2. Business BASIC processes the **ON ERR** statement.
3. After that, any time Business BASIC encounters an error, it executes the statement following the keyword **THEN.**

If an error occurs before the **ON ERR** statement has been executed, Business BASIC halts program execution, displays an error message, and returns you to keyboard mode.

ON ERR THEN INT cancels the previous **ON ERR** statement and returns error handling to Business BASIC.

ON IKEY and **ON ERR** are explained more fully in the manual *Commands, Statements, and Functions in Business BASIC.*

Documenting and Storing Programs

Business BASIC stores a program as either a save file or listing file based on how you place the file on disk. If you create the file in working storage and then write it to disk using the **SAVE** command, you have a save file; if you use the **LIST** command or write your program in an editor, you have a listing file.

The type of file you have and how you created the file (in keyboard mode or with an editor) affect the internal documentation of your program. Documentation in Business BASIC programs takes two main forms: colon comments, which are in-line comments, and **REM** statements.

You can include **REM** comments in any kind of file; however, colon comments go in listing files only. To include colon comments in a file, you can either

- Write the file using an editor and type in the colon comments.
- Add colon comments to an existing listing file by using the **EDIT** utility.

You cannot place colon comments in files in working storage. Thus, if you are in keyboard mode and type in a program statement that contains a colon comment, no comment is displayed if you list that line. A line you type in as

```
* 10 A=12 :one dozen apples
```

appears on your screen when you list the line as

```
* LIST
00010 LET A=12

*
```

Likewise, if you use the **LIST** command to display a program that contained colon comments before you entered it, the comments are not displayed. To see a listing file's colon comments, you must create your file with an editor (or the **EDIT** utility) and then look at the file with an editor or use the BASIC CLI command **TYPE** to display the file at your terminal.

Save Files

The save file is a program that you have stored on disk using either the **SAVE** or **REPLACE** command or statement (that is, **SAVE** "*filename* or **REPLACE** "*filename*). The content of this file is Business BASIC metacode. In this format, each keyword is assigned a number so that the number, not the larger alphanumeric word, is stored. Using this format reduces the amount of space a file takes up on disk, but it also prevents you from using an editor to modify the file.

To preserve internal program documentation, use the **REM** statement or keep a copy of the program in a listing file.

Listing Files

A listing file is an ASCII format file. To get a listing file, either write the program in an editor, or create it in working storage and then use the command **LIST** "*filename* (or **LISTH** "*filename*) to store the file on disk. Using **LIST** "*filename* always produces a listing file (that is, an ASCII text file that contains a program and its **REM** comments).

You can create an ASCII version of a save file by loading the file into working storage and then using the **LIST** command to store the file (with another filename) on disk. You can move a file to disk using **LIST** only when there is not already an existing file with the same filename on disk.

Table 2-3 summarizes the differences between the **LIST**, **SAVE**, and **REPLACE** commands.

Table 2-4 Commands to Save Programs

Command	Output	Access Command	Effect
LIST	ASCII	**ENTER**	Creates an ASCII format file; the name of this file cannot coincide with that of an existing file. Does not preserve variable values or the last line number executed. Allows you to display results of typing and editing in sequential line number order and to use keyboard editing but not colon comments.
SAVE	BASIC SAVE FILE	**LOAD, RUN, CHAIN, SWAP**	Creates a program in Business BASIC save file format; the name of this file cannot coincide with that of an existing file. Preserves variable values and the number of the last line executed. Retains REM comments but not colon comments.
REPLACE	BASIC SAVE FILE	**LOAD, RUN, CHAIN, SWAP**	Creates a program in Business BASIC save file format or overwrites a disk file with the contents of working storage. Preserves variable values and the number of the last line executed. Retains REM comments but not colon comments.

End of Chapter

Chapter 3
Numeric and String Variables

This chapter deals with Business BASIC variables and how you can use them in your programs. In addition, it describes Business BASIC arithmetic, the use of expressions in your programs, and string functions.

Variables

Business BASIC features two types of variables and two derived types: numeric variables, string variables, numeric arrays, and string arrays (UNIX systems only). Numeric variables have no default values; you must assign a value to them before you use them, or you will get an error message. Numeric array elements have a default value of 0. Strings and string array elements have a default value of the null string, which has a length of 0. To find the maximum number of variables you can include in a single program, see the manual that explains how to use Business BASIC on your operating system.

The first character in a variable name must always be a letter, but the following characters can be uppercase letters, lowercase letters, digits, or, on a UNIX platform, an underscore (_).The number of characters allowed in a variable name varies depending on the underlying operating system. Consult the manual that explains how to use Business BASIC on your platform to see how many characters your system permits.

In certain cases, a variable name can include one more than the usual maximum number of characters. In these instances, the last character is a special character that provides additional information about the variable. The special characters are

% to indicate a numeric variable that allows 2 bytes of data to be transferred with **READ/WRITE FILE** statements or **PACK/UNPACK** statements

to indicate a numeric variable that allows 6 bytes of data to be transferred with **READ/WRITE FILE** statements or **PACK/UNPACK** statements

& to indicate a numeric variable that allows 8 bytes of data to be transferred with **READ/WRITE FILE** statements or **PACK/UNPACK** statements (UNIX systems only)

$ to indicate a string variable

Another restriction on variable names is that they cannot be reserved words. The file **APERM.PS** in the library directory of your Business BASIC system contains a list of reserved words. Table 3–1 lists some examples of legal and illegal variable names:

Table 3-1 Examples of Legal and Illegal Variable Names

Legal	Illegal	Reason
A	1	Variable names must begin with a letter.
A303	1AB	Variable names must begin with a letter.
WAGE	$WAGE	Variable names must begin with a letter. In addition, a special character can be used only as the last character in a variable name.
WAGE$	#WAGE	Variable names must begin with a letter. In addition, a special character can be used only as the last character in a variable name.
WAGE#	A$%#	Special characters (other than the underscore on UNIX) can be used only as the last character in a variable name. All other characters in a variable name must be either letters or numbers.

You can assign a value to a variable or change the value of a variable with the following statements:

- **READ/DATA**
- **READ FILE**
- **PACK**
- **UNPACK**
- **LET**
- **INPUT/INPUT USING**
- **TINPUT**
- **PRINT USING**

Numeric Data

Numeric data is limited to integers; however, Business BASIC provides formatting statements (such as **PRINT USING**) that allow you to maintain numeric precision and print numbers with decimal points. In Business BASIC, numeric data includes numeric constants, numeric variables, and numeric array elements.

A numeric constant (also called a numeric literal) is written as a signed or unsigned decimal number. Neither commas nor periods are permitted. Examples of numeric constants are

```
59
-771083
+941
```

Numeric Variables

A numeric variable is a data item that has a numeric value assigned to it during program execution. Examples of numeric variables include

```
A
A3
A#
NUM%
NUM1
OUTPUT
```

Precision

Precision has to do with the number of bytes used to store the value of a numeric variable or array element. If you have an AOS/VS or DG/RDOS system, you can generate either a double-precision or triple-precision interpreter. The latter allows you to use both double- and triple-precision variables. On UNIX systems, you always create a quadruple-precision interpreter; however, you can use a runtime switch that instructs your interpreter to emulate a double- or triple-precision system.

Storage of Numeric Variables

There is a distinction between storage precision and data transfer precision. Only double, triple, and quadruple precision are available for storage, while single, double, triple, and quadruple precision are available for data transfer. Double precision numeric variables store numbers using 4 bytes and can range in value from -(2^31) to (2^31)-1. Triple precision numeric variables store numbers using 6 bytes and can range in value from -(2^47) to (2^47)-1. Quadruple precision numeric variables store numbers using 8 bytes and can range in value from -(2^63)+1 to (2^63)-1.

Data Transfer of Numeric Variables

Business BASIC supports four forms for transferring numeric information to and from a file:

- 2 bytes of data. Variables used for transferring 2 bytes of data are indicated by a percent sign (%) at the end of the variable name. The values assigned to these variables can range from -32,768 to 32,767.
- 4 bytes of data. Variables used for transferring 4 bytes of data have no special character at the end of the variable name.

- 6 bytes of data. Variables used for transferring 6 bytes of data are indicated by a pound sign (#) at the end of the variable name.
- 8 bytes of data. Variables used for transferring 8 bytes of data are indicated by an ampersand (&) at the end of the variable name.

This means that for a **READ/WRITE FILE** statement or a **PACK/UNPACK** statement, 2 bytes are read in or written out for each numeric variable whose name ends with a percent sign; 4 bytes for each numeric variable that has no special character in its name; 6 bytes for each numeric variable whose name ends with a pound sign; and 8 bytes for each numeric variable whose name ends with an ampersand.

For example, if you enter the statement:

```
* 10 WRITE FILE (0), VAR#
```

it transfers 6 bytes (triple precision).

Triple- and quadruple-precision variables cannot be used on a double-precision system, nor can quadruple-precision variables be used on a triple-precision system.

Numeric Arrays

A numeric array is an ordered set of integer values. Each member of the set is an array element. BASIC stores each array element according to the precision of the system. On double-precision systems, an array element holds a 4-byte value; on triple-precision systems, an array element holds a 6-byte value; and on quadruple-precision systems, an array element holds an 8-byte value. The only restriction on the number of array elements you can have is the amount of memory free in the data segment of your program (DG/RDOS and AOS/VS systems) or in your program space as a whole (UNIX systems).

Numeric arrays can have one or two dimensions on DG/RDOS and AOS/VS systems and up to eight dimensions on UNIX systems. Indexing for arrays is zero-based; thus, arrays always start at element 0. Figure 3–1 shows how one- and two-dimensional arrays are set up.

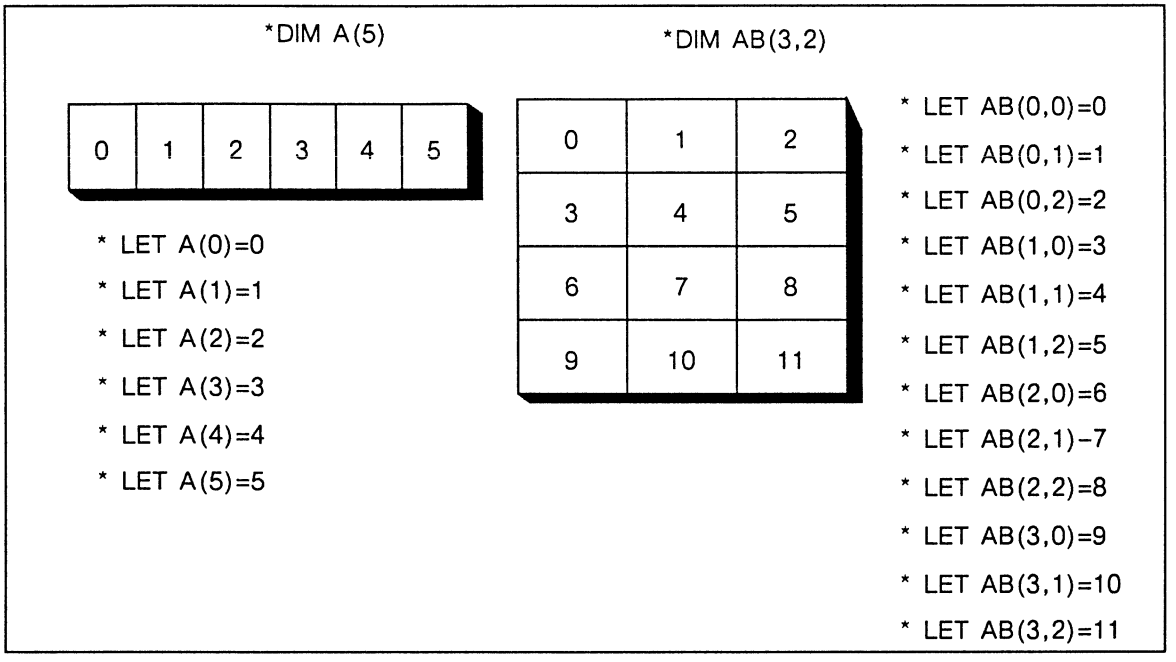

Figure 3-1 One- and Two-Dimensional Numeric Arrays

Creating Numeric Arrays

You create an array by using the DIM statement to specify the number of elements in the array. For example, typing

```
* 10 DIM A(5),B(2,6)
```

sets up array A as a one-dimensional array with six elements and array B as a two-dimensional array with three rows and seven columns that holds 21 elements.

Default Array Dimensions

If you use an array without dimensioning it, Business BASIC acts as if you had specified a value of 10 for each dimension. For example, if you enter the following commands,

```
* NEW
* LET C(3)=7
```

you create array C with 11 elements. To achieve the same effect, you could have entered **DIM C(10)**.

If you enter these statements,

```
* NEW
* LET D(3,2)=6
```

you create a 121-element array D with 11 rows and 11 columns.

You can conserve space by using a dimension statement to declare explicitly an array requiring fewer than the default number of elements.

Accessing Array Elements

You refer to an array element by specifying the array's name followed by a series of subscripts enclosed in parentheses or brackets. Subscripts can be numbers, variables, or expressions that identify the specific position of an element. A subscript must evaluate to a number between zero and the value declared in the array's dimension statement.

One-dimensional array elements are identified by a single subscript. For example, elements of an array declared as **DIM B(5)** are referred to as B(0), B(1), B(2), B(3), B(4), B(5).

An element in a two-dimensional array is referred to by two subscripts separated by a comma. If you use only one subscript to refer to an element in a two-dimensional array, the system uses a default of 0 for the column subscript. For example, A(1,2) refers to row 1, column 2, but A(1) refers to row 1, column 0. Some elements of an array declared as **DIM E(24,5)** are

```
E(I-3,5)
E(O,J*K)
E(24,RND(6))
```

You get an error message if I-3, O, or J*K evaluates to a number outside the range of valid subscripts.

In the third example, RND(6) invokes a system-supplied function to compute a random number between zero and five (the **RND** function is explained in the manual *Commands, Statements, and Functions in Business BASIC*).

Changing Array Dimensions

You can change the dimensions of a declared array by issuing a **DIM** command. This command lets you redimension an array so that it has fewer elements than, or the same number of elements, it originally had. You cannot use the command to make an array larger. For example, you can redimension an array declared as B(2,3) to B(1,2) or B(3,2), but not to B(3,3) since that array would be larger than the original.

Redimensioning an array alters the way you refer to array elements. It does not change the values in the array or the amount of space the array occupies. If you make an array smaller, you do not free unused memory locations; you just make it impossible to refer to them. If you change the one-dimensional array E from 11 elements to 10 as follows,

```
* DIM E(10)
* DIM E(9)
```

you cannot access the element E(10). In addition, references to subscripts outside the newly defined range of subscripts cause an error.

Assigning Values to Numeric Elements

You can assign values to array elements and other numeric variables using the following Business BASIC statements: **LET**, **READ/DATA**, **READ FILE**, **INPUT**, **INPUT USING**, **TINPUT**, **PACK**, and **UNPACK**. The following example uses **LET** statements to assign values to elements in the arrays A and AB.

```
* 10 DIM A(8)
* 20 LET A(4)=5
* 30 LET AB(4,4)=6
* 40 LET AB(2,1)=7
```

In this example, the programmer uses the **DIM** statement to create an array A that contains 9 elements and then in line 30 creates a two-dimensional array that contains the default 11 rows and 11 columns. The **LET** statement operates the same way regardless of how the programmer dimensions the array.

This example uses the **READ** statement to assign values to the variables X and Y and the array element ARA(1,3). The values are obtained from line 80, which contains the **DATA** statement. You can also use the **INPUT**, **INPUT USING**, and **TINPUT** statements to assign values to these variables.

```
* 10 DIM ARA(12,12)
* 20 READ X,Y
* 30 READ ARA(1,3)
* 40 PRINT "X=";X
* 50 PRINT "Y=";Y
* 60 PRINT "ARA(1,3)=";ARA(1,3)
* 70 STOP
* 80 DATA 32,46,1400
* RUN
X= 32
Y= 46
ARA(1,3)= 1400

STOP at 0070
*
```

Use different variable names when you use numeric variables and arrays in the same program. If you introduce a variable X and then try to create an array named X, you receive a subscript error. And if you create an array X and then attempt to assign a value to a variable X, the value is assigned to the first element of the array X; no variable X is created. Refer to the example below.

```
* 10 DIM X(5)
* 20 X(0) = 9
* 30 X = 3
* 40 PRINT "Element 0 in array X = ",X(0)
* RUN
Element 0 in array X =  3

*
```

In this example, line 30 does not cause Business BASIC to create a variable X, but gives a value of 3 to the first element in the array X.

Numeric Expressions

A numeric expression is a combination of numbers, numeric variables, array elements, and numeric functions linked together by some combination of arithmetic operators, relational operators, Boolean logic operators, and parentheses.

Arithmetic Operators

You can use arithmetic operators in any numeric expression to add, subtract, divide, multiply, and perform exponentiation. Business BASIC evaluates numeric expressions according to the precedence shown in Table 3-2.

Table 3-2 Precedence of Arithmetic Operations

Precedence	Operator	Action That Is Performed	Example
1	( )	Evaluation of expressions in parentheses (if parentheses are nested, the expression in the innermost set is evaluated first).	(5-(1+1))
2	^	Exponentiation.	A^B
3	+,-	Multiplication by 1 or -1.	A-(+B) A+(-B)
4	*,/	Multiplication, Division.	A*B A/B
5	+,-	Addition, Subtraction.	A+B A-B

When two operators have equal precedence, Business BASIC evaluates those operators and their operands from left to right. If you enter the statement

```
* LET X=Z+(-A)+B*C^D
```

Business BASIC calculates X in the following order:

1. A is negated.
2. C is raised to the power of D.
3. B is multiplied by the result of step 2.
4. The result of step 1 is added to Z.
5. The result of step 3 is added to the result of step 4.

Using parentheses changes the order of numeric operations. For example, when you enter the statement

```
* LET X=Z-((A+B)*C)^D
```

Business BASIC evaluates X as follows:

1. A+B is evaluated.
2. The result of step 1 is multiplied by C.
3. The result of step 2 is raised to the power D.
4. The result of step 3 is subtracted from Z.

Relational Operators

Business BASIC expressions can also include relational operators. All expressions that contain such an operator are evaluated as true or false and reduced to a value of 1 (true) or 0 (false). Business BASIC then performs an action based on this value. These operators are often used in decision-making statements, such as **IF** statements. Table 3-3 summarizes the relational operators.

Table 3-3 Relational Operators

Operator	Meaning	Example
=	Equals	A=B
<	Less than	A<B
<=	Less than or equal to	A<=B
>	Greater than	A>B
>=	Greater than or equal to	A>=B
<>	Not equal to	A<>B

The following program lines use relational operators as part of an assignment statement:

```
40 LET A=(X>10)
50 LET B=10*(A$>B$)+20*(A$=B$)+30*(A$<B$)
```

Line 40 results in A=1 if the variable X is greater than 10; otherwise A=0. Carrying this technique one step further, line 50 results in B=10 if only A$>B$ is true, B=20 if only A$=B$ is true, and B=30 if only A$<B$ is true.

You can also use relational operators to determine what action your program takes.

```
10 IF A^B>=32767 THEN GOTO 0300
20 IF ABS(X*Y)<>A THEN GOSUB 0900
```

When the expression A^B>=32767 evaluates to true, program control goes to line 300. Line 20 is executed only when the expression A^B>=32767 evaluates to false. In line 20, program control goes to the subroutine that starts at line 900 when the expression ABS(X*Y)<>A evaluates to true; otherwise, the next sequential statement is executed.

Boolean Logic Operators

Business BASIC contains three Boolean logic operators: AND, OR, and NOT. (Business BASIC also has the functions **AND** and **OR**, which work with binary expressions. The system determines whether you are using the Boolean operator or the function based on the placement of AND or OR in the statement.) The Boolean operators can be used anywhere an expression is valid.

Before the overall expression is evaluated, the operands or expressions linked by the Boolean operator are evaluated as true or false. In Boolean evaluation, a value of 0 is considered false, and any other value is considered true. Thus, when you use Boolean operators,

- the expression 3 AND 0 is the same as 1 AND 0 (TRUE and FALSE)
- the expression -1 OR 0 is the same as 1 OR 0 (TRUE or FALSE)
- the expression -1 AND 3 is the same as 1 AND 1 (TRUE and TRUE)

The Boolean operators work in the following way:

- NOT reverses the logical value of an expression (that is, if an expression evaluates to true, NOT causes it to evaluate to false).
- An expression containing AND evaluates to true only if the two operands or subordinate expressions that AND is used with evaluate to true. If either expression is false, the Boolean expression evaluates to false.
- An expression containing OR evaluates to true if either or both of the operands or expressions it is used with evaluate to true. OR expressions evaluate to false only if both subordinate expressions are false.

You can use the Boolean operators in a variety of statements. They are often used in connection with decision-making statements, such as **IF**. For example, the following statement,

```
10 IF A>1 AND B=2 THEN GOTO 100
```

causes Business BASIC to check to see whether the expression A>1 is true or false. The expression is true when A is greater than 1 and false when A is less than or equal to 1. Next, Business BASIC evaluates the expression B=2. This expression is true only when B has a value of 2. If both expressions evaluate to true, Business BASIC executes the **GOTO 100**. If either expression or both expressions evaluate to false, Business BASIC executes the next sequential statement.

You can also combine Boolean operators in a statement.

```
20 IF C AND NOT D GOSUB 200
```

This statement is set up so that expression C evaluates to false only when C equals 0; otherwise, it is considered true. The expression NOT D evaluates to true only when D equals 0. This is true because NOT changes the value of 0 (false) to nonzero (true). The **GOSUB 200** is executed only when C is a nonzero value and D is 0.

The following statement uses NOT together with an OR expression:

```
30 IF NOT (E OR F) GOTO 0100
```

Control goes to line 100 when both E and F equal 0, making the OR expression evaluate to false.

You can also use Boolean operators in assignment statements.

```
40 LET G=NOT (H AND I)
```

This statement assigns G a value of 0 when both H and I are nonzero and a value of 1 when either H and I or both equal 0.

The next statement uses an OR expression to determine what value to print.

```
50 PRINT J OR K
```

Business BASIC prints 0 when both J and K equal 0; otherwise, it prints 1.

Table 3-4 summarizes the precedence in Business BASIC of Boolean operators, relational operators, and arithmetic operators.

Table 3-4 Hierarchy of Operators in Business BASIC

Precedence	Operator
1	Parentheses
2	Exponentiation
3	Unary plus, unary minus, NOT
4	Multiplication, division
5	Addition, subtraction
6	Not equal to, greater than, greater than or equal to, equal to, less than or equal to, less than
7	AND
8	OR

Handling Decimals

All numbers in Business BASIC are 4-byte, 6-byte, or 8-byte integers, depending on whether yours is a double-, triple-, or quadruple-precision system. Use the following procedures when you are working with decimals:

- To add decimal places, multiply a number by an appropriate power of 10.
- To dispose of decimal places, divide a number by an appropriate power of 10.

- When multiplying, remember that the number of decimal places in the multiplier plus the number of decimal places in the multiplicand equals the number of decimal places in the product.
- When dividing, remember that the dividend must have as many decimal places as are needed in the quotient.

The following program adds decimal numbers:

```
:
: This program adds two numbers X and Y and prints the result with two
: decimal places and again with one decimal place.
:
: X=.73 and Y=47.6
:
10 DATA 73,476,73,476
20 READ X,Y
30 LET X=X+Y*10                    :for two decimal places
40 PRINT USING "E6.2,T10,Z",X
50 READ X,Y
60 LET Y=(X+(5*SGN(X)))/10+Y       :for one decimal place
:
: Note the rounding that adjusts for the correct sign:
:      -73.65 rounded = -73.7 not -73.6
:       88.05 rounded =  88.1 not  88.0
:
70 PRINT USING "E5.1",Y
:
: When this program is run, the following appears on the screen:
: 48.33    48.3
:
```

The next program multiplies decimal numbers:

```
:
: This program multiplies two numbers X and Y and prints the result
: with one decimal place and then with two decimal places.
:
: X=.73 and Y=47.6
: X*Y=34.748
:
10 LET X=73
20 LET Y=476
:
: Z will have 1 decimal place.
:
30 LET Z=((X*Y)+(50*SGN(X))*SGN(Y))/100
40 PRINT USING "E6.1,T10,Z",Z
:
: Z will have 2 decimal places.
:
```

```
50 LET Z=((X*Y)+(5*SGN(X))*SGN(Y))/10
60 PRINT USING "E6.2",Z
:
: When this program is run, the following is shown on the screen:
: 34.7     34.75
:
```

The next code segment illustrates dividing decimal numbers:

```
:
: This program divides two numbers X and Y. First X is divided by Y;
: then Y is divided by X. The result is always shown with one decimal
: place.
:
: X=84.73 and Y=47.6
: X/Y=1.78
: Y/X=.561
: Z will always have 1 decimal place.
:
10 LET X=8473
20 LET Y=476
30 LET Z=X/Y                                    : X/Y unrounded
40 PRINT USING "E6.1,T10,Z",Z
50 LET Z=(((X*10)/Y)+SGN(X)*SGN(Y)*5)/10        : X/Y rounded
60 PRINT USING "E6.1,T10,Z",Z
70 LET Z=Y*100/X                                : Y/X unrounded
80 PRINT USING "E6.1,T10,Z",Z
90 LET Z=(((Y*1000)/X)+(SGN(X)*SGN(Y))*5)/10 : Y/X rounded
100 PRINT USING "E6.1",Z
:
: When this program is run, the following output is shown on the
: screen:
: 1.7      1.8       0.5       0.6
:
: Note that in the unrounded examples the answer is not accurate beyond
: the integer portion. If accuracy is important, then rounding is
: required.
:
```

Predefined Numeric Instructions

Business BASIC provides a number of functions and statements that you can use in computations. These are summarized in Table 3-5. You can use the functions in any Business BASIC statement that allows numeric expressions. The functions and statements are explained in detail in the manual *Commands, Statements, and Functions in Business BASIC*.

Table 3-5 Numeric Functions and Statements

Format	Usage	Description
ABS	Function	Computes the absolute value of a numeric expression.
AND	Function	Sets bits based on the result of the logical AND of two expressions.
ASC	Function	Returns the ASCII value for a string.
COMP	Function	Returns the complement of a numeric expression.
CHR$	Function	Places the binary value of a number into a string.
DEF	Statement	Creates user-defined functions.
INT	Function	Truncates a number to make it an integer.
LEN	Function	Finds the current length of a string.
MAX	Function	Finds the larger of two numeric expressions (DG/RDOS and AOS/VS systems) or the largest of up to 16 expressions (UNIX systems).
MIN	Function	Finds the smaller of two numeric expressions (DG/RDOS and AOS/VS systems) or the smallest of up to 16 expressions (UNIX systems).
MOD	Function	Finds the remainder after dividing one numeric expressions by another.
OR	Function	Sets bits based on the result of a logical inclusive OR of two expressions.
POS	Function	Determines the position of a substring in a string.
RND	Function	Produces a random number.
SGN	Function	Determines the sign of a numeric expression.
SHFT	Function	Shifts bits left or right.
SQR	Function	Computes the square root of a numeric expression.
VAL	Function	Converts a string to a number.
VALUE	Statement	Converts a string to a number.
XOR	Function	Sets bits based on the result of a logical exclusive OR of two expressions (UNIX only).

Character Data

Business BASIC uses strings to handle character data. A string is a combination of characters. It can include letters, digits, spaces, special characters, and sometimes binary values. You can have a string literal (also called a string constant), a string variable, or an array of strings (UNIX systems only).

Anytime you use a string variable, you must dimension it first. Business BASIC does not provide default dimensioning for strings as it does for elements of numeric arrays. After you dimension a string, it has a default value of the null string until you assign it a value. The maximum dimension for a string variable varies depending on your operating system. To find the maximum dimension allowed on your operating system, consult the manual that explains how to use Business BASIC on that system.

String Literals

A string literal is a combination of characters that you delimit with quotation marks. For example:

```
"Data General Corporation"
```

You can also include a control character in a string literal. You do this by entering the character's ASCII value (in decimal) in angle brackets in the string. Use the form *<n>*, where *n* is a number from 0 to 255. The angle brackets do not appear when the string is displayed. When you type in

```
* PRINT "At the sound of the tone <7>"
```

the string "At the sound of the tone" is displayed, and the terminal bell rings (7 is the ASCII code for bell).

NOTE: If you want to avoid using embedded control characters in order to produce more generic code, you can produce the same results using the following command:

```
* PRINT "At the sound of the tone";@(-25)
```

Just as you can assign numeric constants to numeric variables, you can also assign string literals to string variables. To do this, use one of the Business BASIC assignment statements, such as **LET** or **READ/DATA**. If you use the **INPUT** statement, you can use one line of code to display a string literal and to assign a string literal to a string variable:

```
* 10 DIM A$(20),B$(20)
* 20 INPUT "Enter your string: ", B$
* 30 LET A$="This is a string: "
* 40 PRINT A$;@(-28);B$
* 50 END
* RUN
Enter your string: Good morning.
This is a string:
Good morning.

*
```

You can also use string literals with **PRINT**:

```
* PRINT "    THIS IS A STRING LITERAL"     THIS IS A STRING LITERAL

*
```

String Variables

String variables are data items that have string values assigned to them. Business BASIC requires that the names of string variables end in dollar signs (for example, A$). String variables use 1 byte to hold the ASCII code for each character of the string. Unlike numeric and string arrays, string variables start with an index of 1 (that is, positions 1-8 instead of 0-7).

Since Business BASIC does not provide a default dimension for strings, you must use the **DIM** statement to allocate the maximum number of characters (bytes) for the string variable before you assign a string value to it.

```
* DIM A$(25),B3$(215)
```

This statement dimensions two string variables: A$ can store a maximum of 25 bytes (1-25) and B3$ a maximum of 215 bytes (1-215). You can redimension a string; however, the new maximum length must be less than or equal to the original maximum length.

You can dimension both arrays and strings in a single line of code:

```
* 10 DIM ARRAY(5,6),C(20),STRNG$(30)
```

Line 10 dimensions ARRAY as a 42-element (6 by 7), two-dimensional numeric array; C as a 21-element, one-dimensional numeric array; and STRNG$ as a 30-character string variable. (As you will see in the next section, you could also create a string array in this statement.) To change these dimensions, enter

```
* 20 DIM ARRAY(6,5),C(2,6),STRNG$(28)
```

This statement redimensions ARRAY to a 7 by 6, two-dimensional array (still 42 elements), redimensions C to a 3 by 7, two-dimensional array (still 21 elements), and redimensions STRNG$ to a maximum of 28 characters.

String Arrays

String arrays are available on UNIX systems only.

A string array is similar to a numeric array: it is an ordered set of strings. Each member of the set is called an array element, and each element occupies a number of bytes that you declare when you create the array. The only restriction on the number of array elements you can have is the amount of memory free in your program space.

Like numeric arrays on UNIX systems, string arrays can have up to eight dimensions. The indexing for each dimension is zero based; thus, the first element in each dimension is element 0. (Remember that the first character in each string array element is character 1.)

Creating String Arrays

You create a string array by using the **DIM** statement to specify the length of each element in the array and the number of elements in each dimension of the array. The integer representing the length in bytes of each element is separated from the rest of the integers in the statement by a semicolon. The remaining integers, those representing the number of elements in each dimension, are separated by commas. For example, the statement

```
10 DIM A$(80;1,2)
```

defines a string array made up of two rows and three columns of 80-byte elements.

Accessing String Array Elements

To refer to the entire string in one element of the array, do not specify the length of the string, but do place a semicolon after the left parenthesis of the index. For instance, this command places in X$ the entire string at element 1,1:

```
* 10  X$=A$(;1,1)
```

You refer to a substring of a particular element's string in this way:

```
* 20  X$=A$(4,9;3,4)
```

This statement finds the fifth column of the fourth row of the string array A$ and places bytes 4 through 9 of the element at that location into the string variable X$.

Changing Array Dimensions

As with numeric arrays, you can redimension a string array. However, you must be aware of these two restrictions:

- The string length in the redimensioned array must be less than or equal to the length in the original array.
- You may change the number and size of the array's dimensions as long as the number of elements in the redimensioned array does not exceed the number in the original array.

If the length of the strings in the redimensioned array is less than the length of the strings in the original array, the current string length for each element in the new array is set to zero. On the other hand, if the length remains constant, the interpreter assumes that you want to keep the contents of the elements in the array, but be able to refer to them in a different way. This is consistent with the way the interpreter handles numeric arrays.

Accessing Strings

Business BASIC allows you to access entire strings and subsections of the strings (called substrings). You refer to the complete string by specifying the name of the string variable. To access a substring, specify the string variable's name followed by one or two subscripts. Subscripts indicate the character locations of the substring within the string. Use only one subscript if your substring continues to the end of the string. Use two subscripts if your substring ends before the main string. The first subscript specifies the starting character position of the substring, and the second one specifies the ending character position.

A subscript can be a number, a numeric variable, or an expression that evaluates to a number between 0 and the value declared in the string's dimension statement. The subscript 0 refers to the position that follows the last character of the string (you use this subscript to add a substring to the end of a string). Table 3-6 illustrates different ways to refer to a string:

Table 3-6 References to Strings

Reference	What It Specifies
A$	Entire string.
A$(2)	Second through last character.
A$(R)	Rth through last character, where R is a number in the range 1 to the maximum length of the string.
A$(3,7)	Third through seventh characters.
A$(I,J)	Ith through Jth characters, where I and J are both numbers in the range 1 to the maximum length of the string and where I is less than or equal to J. If I is greater than J, A$(I,J) is a null string.
A$(0)	The position immediately following the last character in A$ (a string can contain fewer than the maximum number of characters allowed). This is equivalent to A$(LEN(A$)+1).
A$(4,9;3,4)	Fourth through ninth characters of element 3,4 of the string array A$.

Using Strings in Expressions

You can use string expressions (string constants, string variables, elements of string arrays, and subscripted string variables) in **LET**, **PRINT**, **INPUT**, and **READ** statements, and in relational expressions in **IF** statements. Table 3-7 contains examples of how string expressions are used.

Table 3-7 Uses of String Expressions

Statement	Action
`PRINT A$(1,4)`	Prints the first four characters of A$.
`LET B$= "RESULTS ARE: "`	Assigns a string literal to B$.
`IF A$(I)=B$(J) GOTO 10`	Transfers program control to line 10 if the substring containing the Ith through the last character of A$ is the same as the substring containing the Jth through the last character of B$.
`INPUT C$,D$(1,1;2,3)`	Lets you enter a string literal for C$ and a single character for element 2,3 of the string array D$.

Assigning Values to Strings

Use Business BASIC assignment statements to assign values to entire strings or to store characters in different locations of a string.

With the **LET** and **READ** statements, you can use

- String variables
- Elements of a string array
- Subscripted string variables
- String functions
- The concatenation operator (comma)

The **PRINT**, **INPUT**, **PACK**, and **UNPACK** statements work with all variable forms except string functions. (A string function is a built-in Business BASIC function that evaluates to a string value.)

There are two restrictions on string assignments:

- You cannot assign more characters to a string than the string was dimensioned to hold. When you try to assign too many characters, Business BASIC truncates the data to fit the string. You do not receive an error message.
- Strings must be filled beginning from position 1. For example, you cannot assign "ABC" to A$(3,5) if A$(1,2) are empty.

Table 3-8 contains examples of string assignments.

Table 3-8 Assigning Characters to String Locations

Assignment	Action
`LET A$=B$`	Replaces the contents of A$ with the contents of B$.
`LET A$=""`	Sets the length of A$ to 0.
`LET A$=A$,B$`	Appends the contents of B$ to the current contents of A$.
`LET A$(0)=B$`	Appends the contents of B$ to the current contents of A$.
`LET A$=B$(;2,3)`	Moves the string stored in element 2,3 of the array B$ to A$.
`LET A$=FILL$(0)`	Fills A$ to its dimensioned length with nulls.
`LET A$=B$,A$`	This produces unpredictable results because A$ has been changed by the time it is to be appended.
`LET A$=4+5`	Replaces the contents of A$ with the string constant 9.
`LET A$=12345`	Replaces the contents of A$ with the string of digits 12345, not the number 12345.
`LET A$=CHR$(12,4)`	Replaces the contents of A$ with a 4-byte string holding the binary value of the number 12.

Concatenating Strings

You can concatenate strings in an assignment statement by separating the string expressions with commas. In the following program, commas are used to concatenate strings A$ and B$ and two string literals when string C$ is formed:

```
* LIST
00010 DIM A$[16],B$[23],C$[50]
00020 LET A$= "THE YEAR IS 19XX"
00030 LET B$= "THE MONTH IS XXXXXXXXXX"
00040 LET C$=A$[1,14],"89; ",B$[1,13],"JUNE"
00050 PRINT C$
00060 END
* RUN
THE YEAR IS 1989; THE MONTH IS JUNE

*
```

String Functions

Business BASIC supplies several functions and statements that provide additional string assignment capabilities. The functions can be used in assignment statements or commands. Table 3-9 lists these functions and statements. They are explained more fully in the manual *Commands, Statements, and Functions in Business BASIC*.

Table 3-9 String Functions and Statements

Format	Usage	Description
ASC	Function	Returns the ASCII value of a string.
CHR$	Function	Places the binary value of a number in a string.
CRM$	Function	Crams 3 bytes of a string into 2 bytes.
EXTRACT	Statement	Extracts the next field from a string.
FILL$	Function	Fills a string or substring with a value.
LEN	Function	Finds the current length of a string.
PACK	Statement	Encodes string and numeric information into a string variable known as a record string.
POS	Function	Determines the position of a substring in a string.
SCANUNTIL	Statement	Scans a string until characters in a substring are found.
SCANWHILE	Statement	Scans a string while characters match those in a substring.
STRPOS	Statement	Finds the starting position of a substring in a string.
TRUN$	Function	Truncates a string.
UCM$	Function	Expands a crammed string so that 2 bytes are again stored in 3 bytes.
UNPACK	Statement	Takes information from a record string containing binary information and places it in separate variables.
VAL	Function	Converts a string of digits to a number.
VALUE	Statement	Converts a string of digits to a number.

Business BASIC also provides three string functions that deal with error messages:

- **ERM$**, used with **SYS(7)**, **SYS(40)**, or **SYS(41)**.
- **AERM$**, used with **SYS(31)** or **SYS(42)**.
- **UERM$**, used with **SYS(43)**. Currently, this function is available on UNIX systems only.

You use the **ERM$** function to retrieve the text associated with a Business BASIC error or a DG/RDOS I/O error. **AERM$** returns the text for an AOS/VS I/O error that can not be translated to a DG/RDOS error message, and **UERM$** returns the text for a UNIX I/O error that can not be translated to an AOS/VS error message. For more specific information on how to use these functions, see *Commands, Statements, and Functions in Business BASIC.*

Using Variables to Transfer Data

File access statements (**READ FILE**, **LREAD FILE**, **WRITE FILE**, **LWRITE FILE**, **BLOCK READ FILE**, **BLOCK WRITE FILE**, **INPUT FILE**, and **PRINT FILE**) use variables to transfer data to and from files. **BLOCK READ** and **BLOCK WRITE** use variables to transfer data to and from the common area. These methods for transferring data apply to string variables as well as to numeric variables.

The size of the variables you supply as arguments determines the number of bytes transferred. String variables transfer their maximum (dimensioned) length even when they are only partially filled with data (empty string bytes transfer as null bytes). Substrings transfer the number of bytes specified in their subscripts. Both substrings and entire strings transfer 1 byte per character.

Numeric/String Conversions

There are three Business BASIC functions and three statements that convert numeric data to string data and vice versa. These are

- **CHR$**, which puts the binary value of a number into a string.
- **ASC**, which returns the ASCII value of a string.
- **PACK**, which encodes string and numeric information in a single string variable, known as a record string.
- **UNPACK**, which decodes string and numeric information from a record string.
- **VAL**, which converts a string of digits to a number.
- **VALUE**, which converts a string of digits to a number.

These functions and statements are explained more fully in the manual *Commands, Statements, and Functions in Business BASIC*.

End of Chapter

Chapter 4
Subroutines and Utilities

The Business BASIC software package contains subroutines and utilities to aid you in programming. A subroutine is a segment of code that is designed to perform a specific function from within a program. A utility is a Business BASIC program that performs a specialized task.

This chapter discusses the subroutines and utilities that come with the Business BASIC package. In addition, it covers modifying existing subroutines, writing your own Business BASIC subroutines, and calling assembly-language subroutines.

Subroutines

Subroutines fall into three categories:

- Prewritten subroutines. These are the subroutines that come with the Business BASIC software package. They are located in the library directory. Use the **ENTER** command to place them in working storage and merge them with your program.
- User-created Business BASIC subroutines. These are subroutines that you write for your own specialized processing requirements. You can store them in the library directory and then access them repeatedly just as you access the Business BASIC-supplied subroutines.
- Assembly-language subroutines (DG/RDOS and AOS/VS only). These are also subroutines that you write yourself. You give yourself access to assembly-language subroutines by placing your routines in a source file, assembling that file to produce a relocatable binary or object file, and then linking the resulting file with your interpreter.

Business BASIC Subroutines

Business BASIC features a number of prewritten subroutines that you can use in application programs. These subroutines have .SL extensions on their filenames to distinguish them from utility programs and other files.

The Business BASIC subroutines, their entry points, and the line numbers they occupy are discussed fully in the manual *Subroutines, Utilities, and the Business BASIC CLI.*

Using Subroutines

There are two general procedures for using subroutines:

- Write them as elements of a specific program and type in the program and its subroutines at the same time.
- Write them as individual code segments, test them, and then store them in the library directory until you need them. You can add them to your program later.

The second method helps you debug your program because you have program elements that you know work. Error tracing is more difficult when you debug a complete Business BASIC program at one time.

To use an existing subroutine, you must merge it with a program. To do this, follow these steps:

1. Place the main program in working storage either by creating it there or by loading or entering it.
2. Use the **ENTER** command to load the subroutine into working storage and merge it with the main program.
3. Use the **SAVE** command to store the now-complete program. This makes the subroutine a permanent part of the program so that you no longer need to enter the subroutine each time you run the program.

Instead of performing the last two steps, you can include an **ENTER** *"subroutine-name"* statement in your program. Then the subroutine is automatically added each time you run the program. You no longer need to save the program to retain the subroutine. The disadvantage to this method is that adding subroutines at runtime slows down program execution. The advantage is that, since the prewritten subroutines can change each time a new revision of Business BASIC is released, you will always have the most up-to-date version of each subroutine.

If you use the **ENTER** command to bring the Business BASIC-supplied subroutines into your program once and then save the program, you should keep track of which programs use which subroutines. This will aid you in replacing subroutines that have been updated when you get a new revision of Business BASIC. Another way to maintain up-to-date subroutines is to keep a copy of your program with an **ENTER** statement for each subroutine. Place a **STOP** statement after each **ENTER** statement. Only run this version of your program when you get a new revision of Business BASIC. Then delete the **ENTER** and **STOP** statements and save the program with the new subroutines in it.

Before you use the **ENTER** command to read a file into working storage, check the subroutine's line numbers to see if any of them match line numbers in the program already there. If there are matches, Business BASIC replaces the existing program statements with the program statements that you are entering. You can use the Business BASIC CLI **TYPE** command to check the line numbers of a subroutine. Use the following format:

!TYPE *subroutine-name*.**SL**

The **TYPE** command displays the subroutine at your terminal. It also displays the colon comments that are included with Business BASIC subroutines to explain how they work. These comments are stripped from the subroutine when it is entered into working storage. You can also use the BASIC CLI **PRINT** command to print subroutines with their comments. In addition, two other BASIC CLI commands work together to create a file containing only program comments. These are **BLDCOM** and **PRTCOM**. These BASIC CLI commands are explained in the manual *Subroutines, Utilities, and the Business BASIC CLI.*

To execute a subroutine from within a program, use the **GOSUB** or **ON GOSUB** statement. Your program can contain several **GOSUB** statements to the same subroutine. Each subroutine has a **RETURN** statement that returns control to the statement immediately following the **GOSUB** or **ON GOSUB** statement.

Writing Business BASIC Subroutines

Business BASIC lets you write your own subroutines or modify the subroutines supplied with the software package. The following list contains guidelines for writing or changing subroutines:

- Start your subroutine with a **REM** statement that describes the subroutine. When you list a program, Business BASIC displays next to each **GOSUB** statement the comments you typed in at the entry point of the subroutine being called.
- Select line numbers for your subroutine that are outside the range you normally use in your programs. This makes it easier to merge your subroutine with other programs by reducing the chance that the subroutine line numbers will conflict with program line numbers. The exact line numbers in each subroutine supplied with your Business BASIC software package are documented in the manual *Subroutines, Utilities, and the Business BASIC CLI.*
- If you have variables in your main program whose values your subroutine should not change, make sure that the names of those variables are not used as variable names in the subroutine.
- Always include a **RETURN** statement. It returns program control to the line after the **GOSUB** statement in the calling program. You can have more than one **RETURN** statement in your subroutine. Using this technique enables you to use program logic that has the subroutine end at different places, according to which conditions are met.
- Use the extension .SL with the filename you use to identify your subroutine. Your subroutine will work without this extension; however, using it will ensure that the names of your subroutines and those supplied with Business BASIC follow the same conventions.
- If you want to use your subroutine with other programs, use the **LIST** command to store it in the library directory.

Business BASIC allows one subroutine to call another, but there is a limit on nested subroutines. This limit depends on the operating system you are using, so consult the manual that explains how to use Business BASIC on your operating system to find the maximum number of **GOSUB** levels. If you exceed this maximum or disrupt the logic of a nested subroutine, an error occurs. You can reset the **GOSUB/RETURN** stack by using the **STMA 8** command/statement.

To add a subroutine to your program, follow the steps listed in the previous section, "Using Subroutines."

Errors with Subroutines

When you use subroutines in a program, certain errors can occur, in particular `Error 13 - Line number` and `Error 19 - RETURN - NO GOSUB`.

If your program executes a **GOSUB** statement and the subroutine that the statement addresses is not in working storage, error 13 occurs. The **GOSUB** refers to a line number that does not exist. You can avoid this problem by saving the program once its subroutines have been added.

Error 19 occurs whenever BASIC executes a subroutine that was not called by a **GOSUB** statement. This can happen if you write a program or use the **ENTER** command to bring a program into working storage without issuing a **NEW** command to clear working storage. A subroutine from a previous program could have been left in working storage. In such a case, since Business BASIC executes all working storage program statements sequentially, it would also execute the subroutine. You can avoid this error by clearing working storage before writing programs and including an **END** statement in all programs. Both the **END** and **STOP** statements halt program execution.

Errors also occur when you do not provide proper values for variables or proper variable names for subroutine variables. Each subroutine requires specific input variables and returns specific output variables.

You must also supply the proper entry point to the subroutine. Most subroutines have more than one entry point.

Subroutine Example

The following program, GET, uses the Business BASIC-supplied subroutine **GETCM.SL** to create a BASIC CLI command. This program returns the combined values of the switches /B and /C. To use the subroutine, you have to dimension T9$ (**GETCM.SL**'s input variable) and X$ (**GETCM.SL**'s output variable) in your main program. GET has one **GOSUB 7550** to initialize the **GETCM.SL** values. Then, in line 60, GET loops back to the **GOSUB 7500** until S (another of **GETCM.SL**'s output variables) returns a -1. In line 50, GET prints the BASIC CLI command and the value of S.

```
* LIST
00010 DIM T9$[512],X$[24]
00020 GOSUB 7550
00030 GOSUB 7500
00040 IF S=-1 THEN STOP
00050 PRINT X$, TAB(35),S
00060 GOTO 00030

* ENTER "GETCM.SL
* SAVE "GET
```

```
* LIST
00010 DIM T9$[512],X$[24]
00020 GOSUB 07550 : \ INITCM
0030 GOSUB 07500 : \ GETCM.SL
00040 IF S=-1 THEN STOP
00050 PRINT X$, TAB(35),S
00060 GOTO 00030
07500 REM \ GETCM.SL
07505 IF T9$[Q9,Q9]="<255>" THEN GOTO 07540
07510 LET X$=TRUN$(T9$[Q9])
07512 IF X$="" THEN GOTO 07540
07515 LET Q9=Q9+LEN(X$)
07520 IF Q9>508 THEN STOP
07525 UNPACK "L",T9$[Q9+1],S
07530 LET Q9=Q9+5
07535 RETURN
07540 LET S=-1
07545 RETURN
07550 REM \ INITCM
07552 LET Q9=1
07554 BLOCK READ T9$
07556 RETURN
07559 REM * END GETCM.SL

* !GET/B/C
GET                                          1610612736

STOP AT 40
*
```

Since you used the **ENTER** command to merge **GETCM.SL** with the program in working storage and then saved the complete program, you do not have to enter the subroutine each time you execute the program. However, when you entered the subroutine, Business BASIC stripped out the colon comments explaining how **GETCM.SL** works. To see those, you need to use the Business BASIC CLI command **TYPE** to list the subroutine. (Instead of saving GET with **GETCM.SL** in it, you could include the statement **5 ENTER "GETCM.SL** in your program. It would merge **GETCM.SL** with your program each time you executed GET.)

For more information on **GETCM.SL** and other Business BASIC subroutines, see the manual *Subroutines, Utilities, and the Business BASIC CLI*. In addition, the entries in that manual for the screen utilities CSM and SM both contain examples of programs that use subroutines.

Assembly-Language Subroutines

On DG/RDOS and AOS/VS systems, you can call assembly-language subroutines if you follow the procedure outlined below:

1. Enter your assembly-language subroutines in the source file **USERSUBS.SR**.
2. Assemble this source file to produce a relocatable binary or object file.

3. Link the relocatable binary or object file with your interpreter.
4. Use a **UCALL** statement in your Business BASIC program to transfer control to one of the assembly-language routines.

You can pass up to eight arguments to an assembly-language subroutine. Each argument can be a string expression, a numeric expression, a string variable (including an element of an array), or a numeric variable (including an element of an array). To find further information on this subject, see the manual that explains how to use Business BASIC on your operating system.

Utilities

Utility programs written in Business BASIC are included in your software package. These programs are designed to help with file processing tasks and maintaining databases. These utilities are discussed fully in the manual *Subroutines, Utilities, and the Business BASIC CLI.*

Using Utilities

Business BASIC provides two types of utilities—those that can be executed in a variety of ways and those that you must execute by using the **SWAP** command.

In general, you can execute the first kind of utility by entering **RUN** "*utility-name*, **CHAIN** "*utility-name*, or **SWAP** "*utility-name*. You can also execute these utilities through the BASIC CLI by entering either the utility name while in the BASIC CLI or !*utility-name* while in keyboard mode. When you use the BASIC CLI, it executes the utility by closing any open files and performing a swap to the program named.

Some utilities, however, are run-only. Do not use any of the three modes of executing the **SWAP** command (**SWAP** "*utility-name*, !*utility-name*, or "*utility-name*) with these utilities. For example, you should only execute the **DBGEN** utility by entering:

```
* RUN "DBGEN
```

The second kind of utility must be run in conjunction with another program. Information is passed to it, and it returns the results through the common area.

When utilities return information to the calling program or use the common area to pass information to programs, you should swap to them. They do not work properly unless they can read the common area with a **BLOCK READ** statement and interpret its data. If you use the **CHAIN** command to chain to one of these utilities, the information passed from the utility to the common area must be retrieved using keyboard mode commands. The following is a list of the **SWAP**-only utilities:

- **FILESORT**
- **IBUILD**
- **OPEN**

- QFILESORT
- SIZE
- TBUILD
- XBUILD

You also need to swap to a utility if that utility works with open files. This is the case with the **LOCKS** utility. **LOCKS**, like some other utilities, is structured internally so that it checks to see how it was executed. If it was not executed by a **SWAP** command, **LOCKS** issues a **NEW** command to close all open files. You cannot execute these utilities through the BASIC CLI because the BASIC CLI issues a **CLOSE** command to close all open files.

Common Area

The common area is a set of memory locations used to store information and pass it from one program to a subsequently running program. You can access the common area directly by using the **GETCM.SL** subroutine or the **BLOCK READ** and **BLOCK WRITE** commands. The size of the common area depends on your operating system, so consult the manual that explains how to use Business BASIC on your operating system to find out the size of the area you have to work with.

Each user has only one common area. You cannot send data into another user's common area. Use the **BLOCK WRITE** statement/command to put data into the common area and the **BLOCK READ** statement/command to retrieve that data. With **BLOCK WRITE** and **BLOCK READ**, you can use a string to hold the information going into or coming from the common area. The string must be dimensioned to at least 512 bytes because **BLOCK WRITE** and **BLOCK READ** each transfer a minimum of 512 bytes. The maximum size of the blocks these statements/commands transfer is the size of the common area on your system.

Each **SWAP**-only utility also uses a string to access the common area. Build this string according to the instructions given with that utility. When a string of information from a utility is retrieved from the common area, you can use the **UNPACK** command/statement or the **ASC** function to extract the binary values from the string.

Some utilities, like **OPEN**, require literal filenames and numbers in the string. Others, like **FILESORT**, require binary values in the string. Use the **CHR$** function to put a binary value in a string.

You can also use an array with **BLOCK READ** and **BLOCK WRITE** for passing data to and from the common area. Again, the array must be able to hold at least 512 bytes. In a double-precision system, each numeric array element can store 4 bytes; therefore, the array used for passing information to and from the common area must have at least 128 elements (512 bytes). In a triple-precision system, a numeric array must contain at least 86 elements, and in a quadruple-precision system, a numeric array must contain at least 64 elements.

End of Chapter

Chapter 5
Business BASIC Files

Business BASIC places restrictions on naming files. Any file, however, that follows the Business BASIC filename conventions can be used by Business BASIC programs and can be accessed using one of these three methods:

- Sequential access
- Direct or random access
- Indexed sequential access

This chapter provides an overview of files and file handling under Business BASIC. The chapter discusses:

- Filename conventions
- File access
- Business BASIC system files

The information in this chapter is general. In some cases you will need to consult the Business BASIC reference manuals to determine which commands, subroutines, and utilities work with a given file. In addition, Business BASIC supports two database structures, and some commands, subroutines, and utilities only work with one of the structures. Information on these database structures and accessing the data in them appears in Chapter 6.

Filename Conventions

Some of Business BASIC's filename conventions depend on the operating system you are using. These include the maximum length of a filename and the characters you may use in a filename. For information on these conventions, consult the manual that explains how to use the language on your system.

One convention applies, however, in all environments: the filename extensions Business BASIC uses to indicate the content of a file. Table 5-1 lists these extensions and their meanings.

Table 5-1 Filename Extensions

Extension	Meaning
.BA	A utility source file listing that has comments. The **.BA** files are contained in the $DOC directory, which was supplied with your Business BASIC software.
.DB	A database file in the logical file database structure. Business BASIC appends this extension to the filename you supply.
.SL	A Business BASIC subroutine in ASCII format.
.S*n*	A screen file. This is a file used with the Conversational Screen Maintenance (CSM) utility or the Screen Management (SM) utility. The *n* represents the terminal type.
.TB	A table file. It is used with the File Maintenance (FM) utility.
.VL	A volume label file in the logical file database structure. Business BASIC appends this extension to the filename you supply.

Creating Simple Disk Files

There are four commands you can use to create simple disk files with Business BASIC:

- The BASIC CLI command **CCONT**.
- The BASIC CLI command **CRAND**.
- The BASIC CLI command **CREATE**.
- The **OPEN FILE** command.

Under DG/RDOS, the **CCONT** command sets up a contiguous file; the **CRAND** command sets up a random file; and the **CREATE** command sets up a sequential file. Under UNIX and AOS/VS, however, there is only one type of file at the operating-system level, so these commands all create the same type of file.

The **OPEN FILE** command creates a file in your directory only if you open the file in mode 0, 1, or 2. These modes also specify the type of file you will create.

The BASIC CLI commands are explained fully in the manual *Subroutines, Utilities, and the Business BASIC CLI*. The **OPEN FILE** command is discussed in the manual *Commands, Statements, and Functions in Business BASIC*.

Accessing Files

Accessing files in Business BASIC involves several steps. You need to open the files you are going to use as well as set up the variables you will use in transferring information between files. With numeric variables, the amount of data you can transfer depends upon the variable's precision, while with string variables, the amount is determined by the string's dimension.

When you open a file, you must associate its name with a channel number and specify an access mode.

The channel number represents a line of communication from your working storage area to a data file or device. The range of channel numbers available to you depends on the operating system on which you are running; however, channel 16 always refers to the terminal. (To find out the number of channels available on your system, refer to the manual that explains how to use Business BASIC on your operating system.) You can associate a channel number with only one file at a time, and you cannot reuse that number again until you close the file with a **CLOSE FILE** statement. Once a file has a channel number, you use the channel number instead of the filename when reading data from or writing data to the file.

The file access mode is determined in part by the file organization (that is, sequential, random, or contiguous). You specify a file access mode with the **OPEN FILE** statement. When a data file uses sequential file organization, you must open it in sequential access mode. You can open files using random or contiguous organization in either random or sequential access mode.

To access data in the file, use the file pointer. It moves each time you access a record. At the end of the operation, the file pointer points to the byte immediately after the last byte that was read or written. (You can use the **GPOS**, or get position, function to check the position of the file pointer.)

With files opened in random mode, use the **POSITION FILE** statement to move the file pointer. **POSITION FILE** lets you go directly to the record you want. This statement does not work with files opened in sequential mode, since sequential access mode does not allow direct access of records.

To access a record directly, use the record's relative position to place the file pointer. With fixed-length records, you calculate the record's relative position as

record number * record length

To use the sequential access method with data files that are randomly or contiguously organized, position the file pointer to byte 0 of the file. Then process the records according to their order in the file. The file pointer moves with each read or write, so it is automatically positioned and ready to access the next record. You can also position the file pointer to any record in the file and read the file sequentially from that point.

With sequential files, you can use the end of file function (**EOF**) to determine when the data has ended.

Since random access mode lets you access files faster, it is usually used for files containing records that require frequent updating, such as those accessed by a store billing program. Sequential mode is used for files to which new records are constantly being added, for example, a list of company products.

Table 5-2 lists the commands and functions provided for file input and output. These commands are explained in the manual *Commands, Statements, and Functions in Business BASIC*.

Table 5-2 File Input and Output Commands

Keyword	Purpose
BLOCK READ FILE	Gets data from a file in multiples of 512 bytes.
BLOCK WRITE FILE	Places data in a file in multiples of 512 bytes.
CLOSE	Closes all opened files.
CLOSE FILE	Closes a specific file.
EOF	Checks for end of file.
GPOS	Determines the current position of the file pointer.
INPUT FILE	Gets ASCII data from a file.
INPUT FILE USING	Gets ASCII data from a file and allows an error and a terminator trap.
LOPEN FILE	Opens a database file.
LREAD FILE	Gets binary data from a record.
LWRITE FILE	Places binary data in a record.
OPEN FILE	Opens a file.
POSITION FILE	Positions the file pointer.
PRINT FILE	Sends ASCII data to a file, terminal, or device.
PRINT FILE USING	Sends ASCII data according to a format to a file, terminal, or device.
READ FILE	Gets binary data from a file.
WRITE FILE	Sends binary data to a file.

File Types

Since Business BASIC supports operating system files and files specific to the Business BASIC system, files used on a Business BASIC system fall into one of three file types:

- Simple disk files (including direct random format data files).
- Linked-available-record format data files.
- Index files.

These file types give you the flexibility to select a method of data file organization that is appropriate for your needs, from simple to complex. With the exception of RDOS sequential files, you can use any access method with these files. The last two categories (linked-available-record files and index files) both have an embedded Business BASIC structure that tells the BASIC interpreter how the file is organized.

Simple Disk Files

Simple disk files are flat files or operating system files. These files consist of a stream of bytes that you can access at any point. The internal organization of the files can be sequential, random, or contiguous. You must access DG/RDOS sequentially organized files using sequential access; however, you can use sequential or direct random access with files that are organized randomly or contiguously.

If you are using direct random access with your files, then you handle all record assignments. These files must contain fixed-length records. You can position directly to a specific record.

Linked-Available-Record Files

The linked-available-record format uses dynamic record allocation, which allows a file to reuse space left by deleted records. Each deleted record contains a pointer to the next available record in the file. Use either the **LFU** utility or the **INITFILE** utility to create linked-available-record files.

The first record (record 0) in this type of file is reserved by Business BASIC and holds information on the next available record. The rest of the records contain user data.

Business BASIC uses the first 2 bytes of each record following record 0 to determine whether a record is active or deleted. It is your responsibility to set the status of the first 2 bytes to a value greater than 0 when writing an active record. In an active record, data is stored in bytes 2 through the end of the record. In a deleted record, bytes 2 through 5 point to the next record available to receive data, thus creating the deleted-record chain.

Record 0 is always the same size as the other records in the file. When you use the **INITFILE** utility to create a linked-available-record file, the minimum record size is 10 bytes. When you use the **LFU** utility to create the file, the minimum record size is 6 bytes. If the record size is less than 6 bytes, the information normally contained in bytes 2 through 5 of record 0 will be corrupt. When you use **LFU** to create a file with a record size of less than

10 bytes, all of the data records are initially linked together and then allocated from the deleted-record chain.

Table 5-3 describes the contents of record 0. Tables 5–4 and 5–5 describe the contents of an active data record and a deleted data record respectively.

Table 5–3 Contents of Record 0 of a Linked-Available-Record File

Bytes	Description
0-1	Status flag (always equal to –2).
2-5	Record number of next available record (–1 if no records are on the deleted-record chain).
6-9	Record number of last record used in the file (present only when the record size is greater than or equal to 10 bytes).
10-13	Active record count, initially 0 (present only when record size is greater than or equal to 14 bytes). This value is incremented by the **GETREC** statement and decremented by the **DELREC** statement.
14-end	Reserved (present only when record size is greater than 14 bytes).

Table 5–4 An Active Data Record in a Linked-Available-Record File

Bytes	Description
0-1	Status flag (greater than 0 when the record is active).
2-end	User data.

Table 5–5 A Deleted Data Record in a Linked-Available-Record File

Bytes	Description
0-1	Status flag (usually 0, though it can also be less than 0).
2-5	Record number of next deleted record in deleted chain or –1 if last value in chain.
6-end	Unused; it contains old data from when record was active.

If the deleted-record chain for your linked-available-record file is destroyed, you can rebuild it using either the **LRELINK** utility or the **RELINK** utility. You should also run one of these utilities if a system crash occurs while a linked-available-record format file is being updated. These utilities are explained in the manual *Subroutines, Utilities, and the Business BASIC CLI.*

Index Files

An index file provides fast access to information in a data file independent of the information's physical location. An index file does not contain user data; instead, it contains keys that point to entries in a data file.

Business BASIC uses the indexed sequential access method (ISAM) of working with index files. Usually, an index file and its corresponding data file are referred to collectively as an ISAM file. In Business BASIC, an ISAM file consists of a data file and one or more index files. (The **DBGEN** and **FM** utilities limit you to three index files per data file.)

Some additional characteristics of index files are that

- Duplicate keys are permitted, but only when you specify that they are allowed. These are keys with identical values that point to different data file entries.
- The key values are maintained in sorted order, thus allowing you to read a data file sequentially without going through the time-consuming task of sorting the data file.
- The maximum key size is 122 bytes.
- The maximum index file size is 65,535 blocks (0-65,534).
- Unlike data files, index files are organized internally into fixed-length blocks, not records.

Creating Index Files

There are three major steps to setting up an index:

1. Calculate the index file parameters.
2. Create and initialize the index.
3. Build the index by adding keys.

Business BASIC provides several utilities and subroutines for you to use in performing these steps.

INDEXCALC computes and prints information for the index, including

- the maximum number of keys per index block
- the number of keys per block at the user-specified blocking factor
- the number of blocks (either 512 or 2048 bytes per block) at each index level

- the number of blocks in the index
- the number of sectors (512 bytes) in the index
- the number of sectors in the associated data file

The **LFU** and **INITFILE** utilities create and initialize an index file or a linked-available-record file by writing the first record with header information.

IBUILD creates an index file from a sorted data file, a sorted tag file, or an index file.

TBUILD creates a temporary (tag) file from a linked-available-record data file or an index file. A tag file is sorted faster than a data file. The final index file is maintained in sorted order.

XBUILD creates an index file with a blocking factor of 50 percent. Use **INDEXBLD** or **IBUILD** to create an index file with a different blocking factor.

LINDEXBLD and **INDEXBLD** create an index file from an unsorted data file, a tag file, or another index file, or re-create an index file. (Each of these programs use **IBUILD**, **XBUILD**, or **TBUILD**, depending on the options you select when you execute the program.)

IREBLD rebuilds an index file.

LINITINDEX.SL and **INITINDEX.SL** initialize an index file but do not initialize a data file. If you use these subroutines with existing ISAM files, the ISAM files are initialized and any data that was in them is lost.

These utilities and subroutines are explained fully in the manual *Subroutines, Utilities, and the Business BASIC CLI*.

Index File Formats

Index files differ from data files in their internal makeup. Instead of being organized in records, they contain fixed-length blocks of 512 bytes or 2048 bytes each. The larger blocks are available on AOS/VS (not AOS) and UNIX systems only. You specify the index block size when you use the **LFU** utility or the **INITFILE** utility to create the index.

The first block, block 0, is reserved as a header block that describes the index file. Business BASIC sets up this block when you create the index. Table 5-6 describes the format of block 0. The remaining index file blocks contain key and pointer information. The format for these blocks is illustrated in Table 5-7.

Table 5-6 Contents of Block 0 of an Index File

Bytes	Description
0-1	Number of bytes per entry (key and pointer).
2-3	Number of keys per block.
4-5	Last usable block number.
6-7	Next available block.
8-9	Level 0 block number (sometimes referred to as a root node).
10-11	Number of entries per block. (This number depends on the blocking factor requested when the file was created.
12-13	Bit flags. These determine whether the index allows 512-byte or 2048-byte blocks and whether duplicate keys are allowed. The flags can have these meanings: 0 512-byte blocks, no duplicates 1 512-byte blocks, duplicates allowed 2 2048-byte blocks, no duplicates 3 2048-byte blocks, duplicates allowed
14-end	Reserved.

Table 5–7 Format of a Block Containing Keys for an Index File

Bytes	Description
0-1	Number of entries in the block.
2-3	Number of the next block in sequence.
If this is a 512-byte block index, then	
4-end	Key entries. Each entry consists of a key and a pointer to a record in the data file. Keys are fixed length in size and must be an even number of bytes. Record pointers require 4 bytes and must be positive signed numbers.
If this is a 2048-byte block index, then	
4-5	Number of the previous block in sequence.
6-7	Update counter.
8-end	Key entries. These consist of a key and a pointer to a record in the data file, if duplicates are not allowed. If duplicates are allowed, the key entry consists of a key, an occurrence number, and a pointer to a record in the data file. Keys are fixed length in size and must be an even number of bytes. The occurrence number requires 2 bytes and is used to distinguish between duplicate keys. Record pointers require 4 bytes and must be positive signed numbers.

The blocks that make up an index file are created and maintained by Business BASIC when you use any of the keywords for creating and initializing index files or when you add a new key or delete an existing key with the keyword **KADD** or **KDEL**. A block always contains at least one entry. If an addition uses the last available entry, the block is divided into two new blocks. When a block with an odd number of keys is split, the extra key stays in the original block.

If you add keys using **KADD** and the block needs to split, the blocking factor is automatically 50 percent.

Using Index Files

Business BASIC provides five K statements for working with existing index files:

- **KADD** adds a key.
- **KFIND** finds a key.
- **KNEXT** finds the next key.

- **KPREV** locates the previous key. This statement works only with index files built of 2048-byte blocks (AOS/VS and UNIX systems).
- **KDEL** deletes a key.

These statements use block 0 of an index file.

KADD adds a key entry (string and record pointer) to the index file described in a descriptor string. **KADD** searches the index to find the proper location for the key. If you did not allow duplicate keys in the file and attempt to add such a key, **KADD** returns the record pointer with a value of 0 and does not add the new key.

KFIND searches the index file for a match to the key you supply. If **KFIND** finds an exact match, it returns the data record pointer associated with the key. If **KFIND** cannot find the exact key, it finds the first key with a value greater than the specified key and returns the negative value of the record pointer associated with the key found. Suppose you specify key ABC and there is no ABC, but two keys, ABCA and ABCB, exist. **KFIND** locates the key ABCA. If KFIND cannot find a key value equal to or greater than the key you supply, it returns a value of 0 for the record pointer.

KNEXT locates the next key in sequence. You must execute a **KFIND** before your first **KNEXT**. After that, just execute **KNEXT**. **KNEXT** returns the record pointer associated with the key found. If **KNEXT** reaches the end of the index, it returns a 0 for the record pointer.

Use **KNEXT** to read an index sequentially from a given key. For example, use **KFIND** with a null key to return the first key in the index. Then use **KNEXT** repeatedly to reach the key you want. You can also use **KNEXT** to find all occurrences of duplicate keys.

Use **KPREV** after a **KFIND** to find the immediately preceding key, and use subsequent **KPREV**s (or a **KPREV** loop) to read the index sequentially from that point. A common use of **KPREV** is to find duplicate keys or approximate matches; another is to process a data file in a sorted order by reading an index sequentially from the end.

KDEL deletes keys from an index file. If **KDEL** does not find a match for the key you entered, it returns 0 for the record pointer. If it does find a key, it returns the record pointer of the deleted key.

These commands are discussed fully in *Commands, Statements, and Functions in Business BASIC*.

When you add keys to and delete keys from an index file, the multi-leveled ISAM key structure dynamically expands, but it does not dynamically contract. Business BASIC reuses space freed by deleted keys; however, the index file can become full even when there appears to be room for additional key entries. This occurs when a large number of keys are deleted and then replaced by new keys with a different range of values. If this problem occurs, rebuild the index file. Several Business BASIC utilities rebuild index files. These utilities accept an unsorted data file, a tag file, or another index as input files for the new index.

When a data file is used as the input file, the location of a key field must be the same in each data record, and there can only be one key per record per index. This is also true when building tag files from a data file.

If the data file contains multiple record types with a separate index for each type, then the data file cannot be used as an input file with the ISAM utilities. In this case, use the old index file as the input file unless the structure of the old index file is corrupt. If the structure is corrupt, you need to write a program to rebuild the index.

You also need to write your own program if you want an index that contains several keys for each data record. **INDEXBLD**, for example, asks for the number of fields in the key and the starting byte locations within the data record for each field in the key. It writes one key in the index for each record, regardless of type, and it uses the same locations within each data record. **FM** is the only Business BASIC utility that supports multiple record types in one data file with separate indexes for each type.

Two utilities that you can use once you have built an index are **INDEXPRT** and **INDEXVRFY**. These help you check an existing index.

Index File Example

The following example sets up a small database that uses an index file. The data file is **EMPDATA** and the index file is **EMPINDX**. To set up **EMPINDX** and **EMPDATA**, first run **INDEXCALC** to get the information you need to create the index file, including the number of keys per index block and the number of blocks in the index, and then to determine the number of sectors needed by **EMPDATA**. After you get this information, run **INITFILE** to create these two files.

The dialog for **INDEXCALC**, which calculates numbers for the files **EMPINDX** and **EMPDATA**, is

```
* RUN "INDEXCALC
INDEXCALC VERSION X.XX

BYTES PER KEY : 4
BYTES PER DATA RECORD : 25
MAXIMUM NUMBER OF DATA RECORDS : 100
INDEX BLOCKING FACTOR (% PERCENT) [50]: 50
INDEX BLOCK SIZE (512 or 2048) [512]: 512
DUPLICATE KEYS ALLOWED? (Y or N) [N]: N

 63 MAXIMUM KEYS PER INDEX BLOCK
 32 KEYS PER BLOCK WITH A 50 PERCENT BLOCKING FACTOR
 4 BLOCK(S) AT LEVEL 1
 1 BLOCK(S) AT LEVEL 0
 6 BLOCKS (512 bytes each) IN INDEX
 6 SECTORS IN INDEX
 5 SECTORS IN DATA FILE

CALCULATE THE INDEX INFORMATION FOR ANOTHER FILE (Y or N) [N]: N

*
```

The dialog for **INITFILE**, which creates and initializes **EMPINDX** and **EMPDATA**, is

```
* RUN "INITFILE
INDEX (0), DATA (1), STOP(2)  [0]: 0
SUB FILE NAME EMPINDX
FILE NOT IN PARAM FILE!
DO YOU WISH TO ADD? (Y OR N) [Y]: N
MASTER FILE NAME: EMPINDX
INDEX BLOCK SIZE (512 or 2048) [512]: 512
BYTE OFFSET TO SUB FILE: 0
MAXIMUM NUMBER OF INDEX BLOCKS: 6
BYTES PER KEY: 4
BLOCKING FACTOR (% PERCENT) [50]: 50
DUPLICATE KEYS ALLOWED? (Y OR N) [N]: N
INDEX (0), DATA (1), STOP(2)  [0]: 1
SUB FILE NAME EMPDATA
FILE NOT IN PARAM FILE!
DO YOU WISH TO ADD? (Y OR N) [Y]: N
MASTER FILE NAME: EMPDATA
BYTE OFFSET TO SUB FILE: 0
BYTES PER DATA RECORD: 25
MAXIMUM NUMBER OF DATA RECORDS: 100
SHOULD FILE BE NULL FILLED: N
INDEX (0), DATA (1), STOP(2)  [0]: 2

*
```

The following program uses the two files you have set up to store information on employees at Widget Supply Co. As data is added to **EMPDATA**, keys are added to **EMPINDX**.

```
* LIST
00010 DIM NAME$[25],X$[512],KEY$[4],BUF$[544],DESC$[18],C1[1,3]
00020 ON ERR THEN GOTO 09900
00030 RSIZE=25 \ R1=0 \ IFILE=1 \ DFILE=2
00499 REM * Control module
00500 GOSUB 01000 : * Open files
00600 GOSUB 02000 : * Input data & write data records & keys
00999 END
01000 REM * Open files
01010 CLOSE
01020 OPEN FILE(IFILE,5),"EMPINDX"
01030 OPEN FILE(DFILE,5),"EMPDATA"
01040 DESC$=CHR$(IFILE,2),CHR$(0,4),CHR$(0,2),"EMPINDX",FILL$(0)
01190 RETURN
02000 REM * Input data & write data records & keys
02010 R1=R1+1
02020 IF R1>100 THEN 02900
02030 GOSUB 6000 : * Input screen
02040 INPUT USING "",@(10,41),@(-10,6),EMPNO
02050 IF EMPNO=0 THEN GOTO 02900 : * Exit
02060 INPUT USING "",@(12,41),NAME$
```

```
02070 LET KEY$=CHR$(EMPNO,4)
02080 POSITION FILE(DFILE,RSIZE*R1)
02090 WRITE FILE[DFILE],NAME$
02100 KADD DESC$,BUF$,KEY$,R1
02110 REM * Force error if any problem on KADD
02120 IF R1<=0 THEN STMA 19,67
02130 GOTO 02000 : * Input data & write data records & keys
02900 REM * Exit
02910 PRINT @(22,1)
02950 RETURN
06000 REM * Input screen
06010 PRINT @(-30);@(1,18);"W I D G E T   S U P P L Y
     C O M P A N Y"
06020 PRINT @(3,28);"Employee Information"
06030 PRINT @(10,2);"Employee Number:"
06040 PRINT @(12,2);"Name: "
06090 RETURN
09900 REM * Error handler
09910 PRINT "<7> ** Error at line";SYS(20);" - ";
09920 IF SYS(7)<>-60 THEN LET X$=ERM$(SYS(7)) ELSE X$=AERM$(SYS(31))
09930 PRINT X$
09940 END
```

Logical Files and Subfiles

Files in the Business BASIC file structure can be physical (disk) files or logical subsections of a physical file. These subsections are called logical files or subfiles. The distinction between the two terms is that logical files are used with the Business BASIC logical database structure and subfiles are used with the PARAM database structure (database structures are discussed in Chapter 6).

The advantage to using logical files and subfiles is that if you divide a physical file into subsections, Business BASIC lets you simultaneously open an unlimited number of files in the same program. Business BASIC restricts the number of physical files you can open in one program. See the manual that explains how to use the language on your operating system to find this number. Each logical file or subfile has a fixed size and begins where the previous one ends.

Use the **LFU** utility or the **INITFILE** utility to create these files.

End of Chapter

Chapter 6
Database Structures in Business BASIC

Business BASIC supports two database structures: the logical file database structure and the PARAM file database structure. These structures increase the number of files you can access from one program by letting you set up files that are subsections of a physical (disk) file. The subsections are called logical files or subfiles. Business BASIC permits you to open only a certain number of physical files simultaneously in a BASIC program (this number varies depending on your operating system, so see the manual that covers using Business BASIC on your system to determine the number of physical files you can open). However, you can open an unlimited number of logical files or subfiles simultaneously. Since the operating system does not recognize logical files or subfiles, the database structures catalog the logical files and the subfiles by noting their names, their locations within the physical file, the size of their records, and the maximum number of records they contain. You use this information to access these files. To distinguish between the two database structures, this manual uses the terms *database file* and *logical file* only with the logical structure and *master file* and *subfile* only with the PARAM structure.

While the logical structure and the PARAM structure both perform the same function, not all Business BASIC features work with both structures. For example, the code required to open files in a logical database structure and in a PARAM database structure are very different (see the section "Comparing Logical, PARAM Code" in Appendix A). Both structures, however, support data and index files. The data files can have either the direct-access or the linked-available-record format while the index files use the Indexed Sequential Access Method (ISAM).

This chapter describes the two Business BASIC database structures and how to set up and use files with them. The logical file database structure is more recent than the PARAM structure, and some of its features are more efficient and easier to use than the PARAM features. For example, the logical structure computes the byte offset for each record in a logical file; the PARAM structure does not. If you are preparing to set up a database, use the logical database structure. If you are already using the PARAM structure, you should consider whether it's feasible for you to use the **PARAMCON** utility to convert your database to a logical database. Switching database structures requires some program modifications, such as using the **LREAD** statement instead of the **READ** statement.

Logical File Database Structure

The logical file database structure consists of a file set made up of a database file (indicated by a **.DB** extension) and a volume label file (indicated by a **.VL** extension). Both the **.DB** file and the **.VL** file are physical files. The **.DB** file contains the actual data. The **.VL** file contains information on each logical file in the **.DB** file and maps each logical file to the **.DB** file. Logical files appear on disk as links to the volume label file.

If you had a simple logical file database named **CUST** that contained a data file with two indexes, you would have on disk the following two physical files:

CUST.DB	(the database file)
CUST.VL	(the volume label file)

and the following three logical files:

CUSTOMER	(the data file)
CUSTI1	(an index file)
CUSTI2	(an index file)

Each of the three logical files is linked to the volume label file, **CUST.VL**. When you execute a program using these files, the information in the volume label file is copied into the logical file table string (LFTABL$). Both the volume label file and LFTABL$ are discussed later in this chapter.

Creating a Logical File Database

Setting up a database involves the following steps:

1. Design your database so that you know the size of the index key, the record size, and the number of records you want in the data file.
2. Execute the **INDEXCALC** utility to determine the number of keys per index block and the number of blocks (sectors) needed for the index file and the number of sectors needed for the data file.
3. Create the logical and physical files using the Logical File Utility (**LFU**).

To use the information in the database, you need to

1. Dimension the string variable LFTABL$ and fill it with nulls to a length of at least 26 times the highest logical file number to be used.
2. Use the **LOPEN** statement to open your logical files.
3. Access the files by using the input/output statements listed later in Table 6-4.

Appendix A contains an example of setting up a database and a program that uses that database.

Logical Files

You can define your logical files as one of three types:

D Direct random file. The user handles all record assignments.

L Linked-available-record file. Record allocation assignments are made dynamically by Business BASIC.

I Index file. These files are maintained via the ISAM statements.

Logical data files are allocated in 512-byte increments. Index files are allocated in either 512-byte or 2048-byte increments. The increment used depends on the index block size you enter when you use the command **LFU LCREATE** to create the index. Only UNIX and AOS/VS support 2048-byte index blocks.

Volume Label File Format

The volume label file contains records that describe the size and location of the logical files in the database file. Each record is 32 bytes long. Table 6-1 describes the contents of a volume label file record.

Table 6-1 Contents of a Volume Label File Record

Bytes	Description
0-9	Name of the logical file.
10-12	Starting sector number of a logical file in the database file. The formula for calculating the starting sector is Highest starting sector number + ((Last valid record number + 1) * record length + 511) / 512 <= 4194303
13-14	Record length in bytes.
15	File type (D, L, I).
16-19	Last valid record number of logical file.
20-21	Revision mark.
22-31	Unused.

You use the **LFU** utility to maintain the volume label file. When you create logical files with **LFU**, the utility checks the information in the volume label file to determine what file space is available. If you delete a logical file using **LFU**, the utility places *DEL in the field on the volume label file where the logical name should appear. **LFU** reuses this space if you create another logical file of the same size.

LFU consists of a series of commands that let you perform these maintenance tasks. The commands are listed in Table 6-2 and are explained under the **LFU** entry in the manual *Subroutines, Utilities, and the Business BASIC CLI.*

Table 6-2 LFU Command Summary

Command	Function
LCREATE	Creates a logical file of type D, L, or I.
LDELETE	Deletes a logical file.
LINIT	Initializes a type D, L, or I logical file.
LLIST	Displays the type, location in the database file, size in blocks (sectors), record length, last valid record number, and size in bytes of the logical file.
LRENAME	Renames a logical file.
PCREATE	Creates physical database and volume label files associated with a logical database file set.
PDELETE	Deletes the database and volume label file.
PLIST	Displays information on the logical files within a database file.
PRENAME	Changes the names of the database and volume label files.
STOP	Terminates LFU.

Logical File Table (LFTABL$)

The logical file table string (LFTABL$) is a string variable you dimension that stores the definitions of all the logical files opened with the **LOPEN** statement. The information in LFTABL$ is used by the logical input/output statements.

LFTABL$ consists of a series of 26-byte records, where each record holds information on one logical file. The **LOPEN** statement places the logical file definitions in LFTABL$. **LOPEN** gets the logical file characteristics from the volume label file.

Since **LOPEN** places information in LFTABL$, you must dimension LFTABL$ and fill it with nulls to a length at least 26 times the highest logical file number before you use an **LOPEN FILE** statement in your program. Once a file has an entry in LFTABL$, you can refer to it with the input/output statements listed in Table 6-4.

The **LFDATA.SL** subroutine enables you to access information in LFTABL$; however, you should not change this information. Changing data in LFTABL$ can cause the logical input/output statements to perform incorrectly.

Table 6-3 describes the contents of an LFTABL$ record.

Table 6–3 Contents of an LFTABL$ Record

Bytes	Description
1-2	Channel number on which the file was opened with **LOPEN**.
3-6	Starting byte.
7-8	Flags. These are set when you open the file with **LOPEN**.
9-18	Name of the logical file.
19-20	Record length of the logical file in bytes.
21-24	Last valid record number of the logical file.
25	Record type (D, L, or I).
26	Reserved.

Logical File Input and Output

Business BASIC provides several commands, statements, and subroutines that perform input and output operations on logical files. Some of the commands are tailored to the logical file database while others can be used with both databases. Commands that work only with the logical file database require you to open the file using the **LOPEN** statement.

To perform input and output operations in your program:

- First use the **OPEN FILE** statement to open all the files that are not part of the logical database structure.
- Then use **LOPEN FILE** to open the files in the logical database structure.

You open files in this order because the **OPEN FILE** statement allows you to assign a channel number to the file, whereas with the **LOPEN FILE** statement, the system assigns the channel number. If you were to use the **LOPEN FILE** statement first, you would not know which channels were free to be used with the **OPEN FILE** statement. If you try to open a file on a channel that is in use, an error occurs.

Table 6-4 lists the input and output commands that you can use with logical files. These commands are explained fully in the manual *Commands, Statements, and Functions in Business BASIC*.

Table 6-4 I/O Commands Used with Logical Files

Command	Action
DELREC	Delete a logical record in a linked-available-record file. (Logical database only)
GETLAST.SL	Retrieve the number of active records and the highest record in use in a linked-available-record file. (Logical database only)
GETREC	Allocate a logical record in a linked-available-record file. (Logical database only)
KADD	Add a key entry to an index file.
KDEL	Delete a key entry from an index file.
KFIND	Find a key entry in an index file.
KNEXT	Return the next key entry in an index file.
KPREV	Return the previous key entry in an index file.
LOCK/UNLOCK	Synchronize the updating of files that are shared between programs.
LOPEN FILE	Open and/or define a logical file. (Logical database only)
LREAD FILE	Read a logical record. (Logical database only)
LWRITE FILE	Write a logical record. (Logical database only)

PARAM File Database Structure

The PARAM file database structure consists of a file set that includes a PARAM file and a master file. Both files are physical files. The PARAM file contains information on the subfiles, which are logical subsections of the master file.

The PARAM structure permits three types of subfiles:

- Direct random files. The user handles all record assignments. You cannot create this type of file using **INITFILE**.
- Linked-available-record files. Record allocation assignments are made dynamically by Business BASIC.
- Index files. These files are maintained via the ISAM statements.

Setting up a PARAM Database

To set up a PARAM database structure, perform the following steps:

1. Design your database so that you know the size of the index key, the record size, and the number of records you want in the data file.
2. Run **INDEXCALC** to determine the number of keys per index block, the number of blocks (sectors) needed in the index file, and the number of sectors needed for the data file.
3. Create the PARAM file. (You only need to create the PARAM file once.)
4. Run **INITFILE** to initialize the files and enter the necessary information in the PARAM file.
5. Use the input/output commands in Table 6-8 to access your database.

Appendix A contains an example of setting up a PARAM database and a program that uses it.

The PARAM File

The PARAM file consists of records that specify the size and location of subfiles. After you set up the PARAM file, you use the **OPEN** utility to extract the subfile information and place it in the C1 (file characteristics) array. Your program then uses the information in the C1 array to locate records in the subfile.

Each record in the PARAM file is 42 bytes long and contains information on only one subfile. Record number 0 is reserved for information describing PARAM itself.

The Business BASIC software package includes a PARAM file. This file is in the library directory and contains 10 records, three of which are blank. To increase the number of records available, change the record limit in record 0 (see Table 6-5). You can use one PARAM file for all programs, or you can have as many as one PARAM file per directory.

On AOS/VS systems, when you use more than one PARAM file, include the directory containing the PARAM file you need on your search list. On UNIX systems, if you have more than one PARAM file, make sure that the environment variable **BBPATH** contains the name of the directory containing the PARAM file you need. Also, be sure that your program uses the correct PARAM; using the wrong PARAM file can destroy parts of your database.

To set up a PARAM file, follow these steps:

1. Use a BASIC CLI command such as **CCONT** or **CRAND** to create the PARAM file.
2. Initialize record 0 of the PARAM file using the File Maintenance (**FM**) utility. (Table 6-5 contains a description of record 0, and Appendix A discusses using **FM** to set up record 0).
3. Use the interactive utility **INITFILE** to add information describing each physical file and subfile.

In addition to **INITFILE**, you can also add entries to the PARAM file by using:

- The **FM** utility.
- A user-written utility.

CCONT, CRAND, FM, and **INITFILE** are discussed in the manual *Subroutines, Utilities, and the Business BASIC CLI.*

Table 6-5 describes record 0 of a PARAM file, and Table 6-6 describes the record structure for the rest of the PARAM file.

Table 6-5 Record 0 of the PARAM File

Bytes	Contents
0-1	1; this is the active record status indicator.
2-11	PARAM (the rest of the string is filled with nulls).
12-21	PARAM (the rest of the string is filled with nulls).
22-25	0; this is the beginning byte of the PARAM file.
26-27	42; this is the length of each record in the PARAM file.
28-31	Maximum number of records to be kept in the PARAM file.
32-35	Highest record number in use. (When records are added in the PARAM file, this number should be incremented. However, this number should not be decremented when a record is deleted. If you are adding records directly to the PARAM file or if you are using **FM** to add records, you must increment this number yourself; it is not done automatically in those cases.)
36-41	Unused.

Table 6-6 Contents of a PARAM File Record

Bytes	Description
0-1	Status indicator; must equal 1.
2-11	Name of the subfile or physical file.
12-21	Name of the physical file that holds the subfile.
22-25	Byte pointer to the start of the subfile or physical file.
26-27	Length of the records in the subfile or physical file.
28-31	Number of the last record in the subfile or physical file.
32-35	Number of the last record containing data.
36-41	Unused.

C1 (File Characteristics) Array

The C1 or file characteristics array is set up in your program and contains information about your master files and subfiles. The subroutines **GETREC.SL**, **DELREC.SL**, and **POSFL.SL** use this information to compute the position of records within a linked-available-record file.

The C1 array is a two-dimensional array with four columns (0, 1, 2, 3) and *n* rows, where *n* is the number of subfiles used in your program. Remember that arrays are zero-based; thus, if you have *n* files, the maximum row number you have is *n*-1. Table 6-7 explains how the columns in the C1 array are used.

Table 6-7 Column Contents of the C1 Array

Column	Description
0	Contains the number of the channel you used to open a master file. (The channel number is the number you associate with a file for all file access.)
1	Contains the byte offset, relative to 0, to the beginning of the subfile within the master file. (The master file always starts at byte 0.)
2	Contains the number of records in the file.
3	Contains the number of bytes per record.

Building a C1 Array

You dimension the C1 array from within your program. There are three tools you can use to build the C1 array and add information to it:

- The **OPEN** utility.
- The **FINDFILE.SL** subroutine.
- The **LET** statement.

Both **OPEN** and **FINDFILE.SL** use the PARAM file to get the information necessary to build a C1 array. With the **LET** statement, you assign values to the array.

In the program that follows, the record size, channel number, or file size of any file can be changed by changing the values in the C1 array. This program uses the **LET** statement to build the C1 array.

```
10 DIM C1(2,3)              :Dimension C1 to be 3*4 array.
20 LET C1(0,0)=2            :File 0 (EMPDATA) opened on channel 2,
30 LET C1(0,1)=0            :and begins at byte 0 of EMPLOYEE,
40 LET C1(0,2)=400          :and contains a maximum of 400 records,
50 LET C1(0,3)=128          :with 128 bytes in each record.
60 LET C1(1,0)=2            :File 1 (EMPIX) also opened on channel 2
                            :because EMPDATA and EMPIX are in the same
                            :physical file, EMPLOYEE, opened on
                            :channel 2.
70 LET C1(1,1)=51712        :EMPIX begins at byte 51712 (block 101),
80 LET C1(1,2)=15           :and contains 15 index blocks
90 LET C1(1,3)=512          :with 512 bytes in each block.
100 LET C1(2,0)=3           :File 2 (FEDTAX) is opened on channel 3,
110 LET C1(2,1)=0           :starts at byte 0 since it is a
120 LET C1(2,2)=100         :physical file and contains a max of 100
130 LET C1(2,3)=50          :records with 50 bytes in each record.
140 REM -- TIME TO OPEN FILES
150 LET C%=C1(0,0)          :Channel number now in C%.
160 OPEN FILE(C%,0),"EMPLOYEE"  :This will allow access to both
                                :EMPDATA and EMPIX.
170 LET C2%=C1(2,0)         :Channel number of FEDTAX now in C2%.
180 OPEN FILE(C2%,0),"FEDTAX
.
.
.
```

You can use the subroutine **FINDFILE.SL** to have your program automatically build the C1 array. **FINDFILE.SL** creates a C1 array without opening your files. It also returns the next available channel number. If no PARAM file entry exists for the file, **FINDFILE.SL** treats the file as a physical file and asks you for the byte offset, record size, and file size. It does not create a PARAM entry.

When **FINDFILE.SL** ends, you have a C1 array with

- The 0 column (channel number) blank.
- A variable X$ (an output variable required by **FINDFILE.SL**) that contains the filename for the subfile or physical file.
- A variable C% (an output variable required by **FINDFILE.SL**) that contains the next available channel number you can use with an **OPEN FILE** statement.

The following program uses **FINDFILE.SL** to fill the C1 array with the same values used in the previous C1 array example.

```
10 DIM C1(2,3)              :Dimension C1 to be a 3*4 array.
20 DIM X$(10)               :Dimension X$ to max filename size.
30 LET X$="SUB1"            :SUB1 is a subfile in MASTER,
40 LET F%=0                 :and is logical file 0.
45 DIM T9$(42)              :Dimension T9$ to 42 bytes.
50 GOSUB 7800               :Go to FINDFILE.SL subroutine.
60 GOSUB 0200               :Go to verification routine.
70 LET X$="SUB2"            :SUB2 is in MASTER,
80 LET F%=1                 :and is logical file 1.
90 GOSUB 7800               :Go to FINDFILE.SL subroutine.
100 GOSUB 0200              :Go to verification routine.
110 LET X$="PHYS"           :PHYS is a physical file,
120 LET F%=2                :and is logical file 2.
130 GOSUB 7800              :Go to FINDFILE.SL subroutine.
140 GOSUB 0200              :Go to verification routine.
150 STOP
200 REM -- VERIFICATION ROUTINE
210 PRINT X$                :Print name of physical file.
220 OPEN FILE (C%,5),X$     :Open the file.
230 LET C1(F%,0)=C%         :Assign channel number to 0 column.
240 PRINT C1(F%,0),C1(F%,1),C1(F%,2),C1(F%,3)
250 RETURN
```

You can also use the **OPEN** utility to find a channel for your file and to supply the information you need for the C1 array. **OPEN** gets the information for the C1 array from the PARAM file. When you set up a C1 array with **OPEN**, however, you limit the number of subfiles you can have because **OPEN** passes information for the C1 array through the common area. To find the maximum number of subfiles allowed, divide the size of your common area in bytes by 16. (The size of your common area depends on your operating system; to determine its size, consult the manual that explains how to use Business BASIC on your system.) Each entry in the C1 array has 4 elements, and each element is 4 bytes long.

OPEN and **FINDFILE.SL** are explained in the manual *Subroutines, Utilities, and the Business BASIC CLI.*

Modifying a Record in a Linked-Available-Record File

Business BASIC provides three routines that you can use to modify a record in a linked-available-record file:

- **GETREC.SL** to access an available record in order to write to it.
- **POSFL.SL** to position to any record in the subfile or physical file.
- **DELREC.SL** to delete any record in the subfile or physical file.

GETREC.SL and **DELREC.SL** both maintain record 0 of a linked-available-record file.

All three subroutines are explained in the manual *Subroutines, Utilities, and the Business BASIC CLI.*

Positioning to a Record

Use **POSFL.SL** to position the file pointer to a record or to a byte offset in the record. **POSFL.SL** requires the variables F% for the file number, R1 for the record number you want, and, optionally, V% for the byte location in R1.

POSFL.SL returns three values: C% for the channel number of the file to be used with **READ FILE** or **WRITE FILE**, R9 for the byte position in the master file where record R1 starts; and R8 for the byte position of the subfile where record R1 starts.

You can use the variable R9 or R8 with the **POSITION FILE** statement and follow it with a **WRITE FILE** statement to do a quick rewrite of the record. You can also use **POSFL.SL** to position to the data record and execute a **READ FILE** to read the record found.

This program positions the file pointer to a record, reads the record, and uses **POSITION FILE** with R9 to go back to rewrite the record. Note that the code that opens the files and fills the C1 array was not deleted.

```
10 DIM X$(512),C1(2,3),REC$(48)
                              :Record size of SUB2 is 48.
20 LET X$="SUB1,5,SUB2,5,PHYS,6",FILL$(0)
30 BLOCK WRITE X$             :Send file info into common area.
40 SWAP "OPEN"                :OPEN will return X$ with C1 array.
50 BLOCK READ X$              :Retrieve info from common area.
60 LET K=1                    :Pointer to first element in string.
70 FOR I=0 TO 2               :For each file, 0, 1 and 2,
80   FOR J=0 TO 3             :and for each dimension of C1 array,
90     LET C1(I,J)=ASC(X$(K,K+3))
                              :extract element, put in C1 array.
100      LET K=K+4            :Bump pointer 4 bytes.
110    NEXT J
120 NEXT I
130 INPUT "RECORD NUMBER OF SUB2 TO BE REWRITTEN: ",NUM
140 LET R1=NUM                :Give POSFL.SL a record number R1.
150 LET F%=1                  :F% used by POSFL.SL for logical file.
```

```
160 GOSUB 9610              :Position to record R1 in file F% using
                            :POSFL.SL, returns C%, R8 and R9.
170 READ FILE (C%),REC$:C% is channel number.
180 DIM NEWREC$(48)         :For new record.
.                           :Code to build NEWREC$ for new record.
.
.
210 POSITION FILE (C%,R9)   :Use R9 from POSFL.SL to position
220 WRITE FILE (C%),NEWREC$ :and rewrite record R1.
.
.
.
```

Writing a Record

To write a record to a subfile, use the subroutine **GETREC.SL** to get the number of the next available record. **GETREC.SL** finds a record in the deleted-record chain, then updates record 0, and returns the record number. Use this record number with **POSFL.SL** to position to the record before writing to it. Then use **WRITE FILE** to modify the new record.

GETREC.SL allocates a new record in random files if the deleted record chain contains no more records. If you run out of deleted records in a contiguous file, and you've used up all the space allotted to the file, you must copy your file into a larger contiguous file or into a random file.

Deleting a Record

Use **DELREC.SL** to delete subfile records and place them on the deleted record chain so that you can reuse the space. **DELREC.SL** automatically updates record 0 and then deletes the record by setting its status (the first 2 bytes) to 0. **DELREC.SL** uses the C1 array and calls **POSFL.SL**.

The following code segment shows a partial update session. The record is deleted using **DELREC.SL**, and a new one is added using **GETREC.SL**. This update technique is good to use when the placement of new records does not matter. With indexed data files, placement is not important as long as you use **KADD** to add the new key to the index.

```
.
.
.
130 INPUT "RECORD OF SUB2 TO BE DELETED: ",NUM
140 LET R1=NUM
150 LET F%=1            :SUB2 is logical file 1 in C1 array.
160 GOSUB 8600          :Go to DELREC.SL to delete record.
170 GOSUB 8400          :Go to GETREC.SL to find next available record.
180 GOSUB 9610          :Go to POSFL.SL using R1 returned by GETREC.SL.
190 WRITE FILE (C%),NEWREC$ :C% is channel number returned by POSFL.SL.
.
.
.
```

Input and Output with the PARAM Database

Business BASIC provides several commands, statements, and subroutines for performing input and output operations on subfiles. Some commands are tailored to the PARAM file database while others can be used with both databases. To perform input and output operations on your subfiles, you must open them and set up the C1 array.

In the PARAM structure, much of the linked-available-record access is accomplished with the **GETREC.SL**, **DELREC.SL**, and **POSFL.SL** subroutines. Table 6-8 lists the input and output commands you can use with PARAM database files.

Table 6-8 I/O Commands Used with PARAM Database Files

Command	Action
DELREC.SL	Deletes a record in a linked-available-record subfile. (PARAM database only)
FINDFILE.SL	Finds a subfile and builds a C1 array. (PARAM database only)
GETREC.SL	Gets the number of the next available record in a linked-available-record chain. (PARAM database only)
KADD	Adds a key entry to an index file.
KDEL	Deletes a key entry from an index file.
KFIND	Finds a key entry in an index file.
KNEXT	Returns the next key entry in an index file.
KPREV	Returns the previous key entry in an index file.
LOCK/UNLOCK	Synchronizes the updating of files that are shared between programs.
OPEN	Opens physical files and subfiles.
OPEN FILE	Opens a physical file.
POSFL.SL	Positions the file pointer to a record in a data file. (PARAM database only)
READ FILE	Reads binary data from a file or record.
WRITE FILE	Writes binary data to a file or a record.

Converting from a PARAM Database to a Logical Database

Business BASIC provides the **PARAMCON** utility to allow you to convert from the PARAM file database to the logical file database. When you use **PARAMCON**, you need to make whatever changes are necessary to your database so that the files meet the requirements for logical files. These are primarily filename changes and are explained in the manual *Subroutines, Utilities, and the Business BASIC CLI* under **PARAMCON**.

After you convert your files, you can use the logical file input and output statements to access them. This means you will need to modify your programs; for instance, change the **READ FILE** statements to **LREAD FILE**. However, you can still use the **OPEN** utility with these files.

You can also use a special form of the **LOPEN** statement to define physical files that are not part of the logical structure. This lets you use the logical input and output statements with these files.

To maintain your converted files, use the Logical File Maintenance (**LFM**) utility.

PARAMCON and **LFM** are explained in the manual *Subroutines, Utilities, and the Business BASIC CLI*.

Comparing Databases

The logical file database structure and the PARAM file database structure are alike in many ways, but there are some important differences between them. One major difference is that the logical database calculates the byte offset to each logical file; the PARAM database does not do this with its subfiles. Second, you can use the **GETREC** and **DELREC** statements with the logical database, whereas the PARAM database uses the **GETREC.SL** and **DELREC.SL** subroutines. Third, with the logical database, you can position the file pointer to a record and perform I/O with the **LREAD** and **LWRITE** statements; with the PARAM database, you must use the **POSFL.SL** subroutine to position the file pointer. The advantages of the statements over the subroutines are that the statements perform automatic locking, they are faster than the subroutines, and they free the code space normally used by the subroutines. The three subroutines use a total of 55 lines of code.

Table 6-9 compares some of the features of the two database structures.

Table 6-9 Database Features

Logical Database	PARAM Database	Description
Database file	Master file	Physical file that contains the actual data.
Volume label file	PARAM file	Physical file that contains information on locating the logical files or subfiles.
Logical file	Subfile	A subsection of a physical file.
LFTABL$	C1 array	LFTABL$ is a string variable that holds information on logical files. The C1 array is an array that holds information on subfiles. In both cases, this information is used by input and output commands.

End of Chapter

Appendix A
Sample Programs

This appendix contains sample programs that demonstrate how to set up and use logical and PARAM database structures.

Setting Up a Logical File Database

This example demonstrates how to set up a logical database that can be used with the mailing list program that follows. The one physical file and three logical files are created with the logical file structure commands **LFU PCREATE** and **LFU LCREATE**. You use these commands after you define your file layout and run **INDEXCALC** to determine the number of sectors you need. The manual *Subroutines, Utilities, and the Business BASIC CLI* contains explanations of **LFU** and **INDEXCALC**.

The logical files consist of two index files, **MEMBER** and **NAME**, and one linked-available-record file, **CLUB**. They have been designed so that **MEMBER** uses a 4-byte key field and does not allow duplicate keys while **NAME** uses a 24-byte key field and permits duplicate keys. Both use a 50-percent blocking factor. CLUB holds 200 records, each 110 bytes long. The physical file, **CLUBFILE**, is a random file.

Once you have designed your files, run **INDEXCALC** to calculate the numbers for the index files **MEMBER** and **NAME** (in that order). Running this utility also provides you with the number of sectors required by the data file **CLUB**.

```
* RUN "INDEXCALC
INDEXCALC VERSION X.XX

BYTES PER KEY : 4
BYTES PER DATA RECORD : 110
MAXIMUM NUMBER OF DATA RECORDS : 200
INDEX BLOCKING FACTOR (% PERCENT) [50]: 50
INDEX BLOCK SIZE (512 or 2048) [512]: 512
DUPLICATE KEYS ALLOWED? (Y or N) [N]: N

 63 MAXIMUM KEYS PER INDEX BLOCK
 32 KEYS PER BLOCK WITH A 50 PERCENT BLOCKING FACTOR
 7 BLOCK(S) AT LEVEL 1
 1 BLOCK(S) AT LEVEL 0
 9 BLOCKS (512 bytes each) IN INDEX
 9 SECTORS IN INDEX
 44 SECTORS IN DATA FILE

CALCULATE THE INDEX INFORMATION FOR ANOTHER FILE (Y or N) [N]: Y
```

```
BYTES PER KEY : 24
BYTES PER DATA RECORD : 110
MAXIMUM NUMBER OF DATA RECORDS : 200
INDEX BLOCKING FACTOR (% PERCENT) [50]: 50
INDEX BLOCK SIZE (512 or 2048) [512]: 512
DUPLICATE KEYS ALLOWED? (Y or N) [N]: Y

 18 MAXIMUM KEYS PER INDEX BLOCK
 9 KEYS PER BLOCK WITH A 50 PERCENT BLOCKING FACTOR
 23 BLOCK(S) AT LEVEL 2
 3 BLOCK(S) AT LEVEL 1
 1 BLOCK(S) AT LEVEL 0
 28 BLOCKS (512 bytes each) IN INDEX
 28 SECTORS IN INDEX
 44 SECTORS IN DATA FILE

CALCULATE THE INDEX INFORMATION FOR ANOTHER FILE (Y or N) [N]: N

*
```

Next, use this information from **INDEXCALC**, and type in the following commands to create the physical file and the three logical files (this example assumes that you are executing **LFU** from the BASIC CLI and using its command line format):

```
* !LFU PCREATE CLUBFILE 0

* !LFU LCREATE MEMBER CLUBFILE I 512 8 4 50 N

* !LFU LCREATE NAME CLUBFILE I 512 27 24 50 Y

* !LFU LCREATE CLUB CLUBFILE L 110 200 N
```

These command lines create

- A physical file **CLUBFILE**.
- An index file **MEMBER** with 512-byte blocks, a last usable block number of 8 (since the 9 blocks in the index mean blocks 0-8), a 4-byte member number field, a 50-percent blocking factor, and no duplicate keys.
- An index file **NAME** with 512-byte blocks, a last usable block number of 27 (since the 28 blocks in the index mean blocks 0-27), a 24-byte name field, a 50-percent blocking factor, and duplicate keys allowed.
- A linked-available-record file **CLUB** with records of 110 bytes each, 200 records in the file, and no null filling of the file.

You can use the following program to add records to the database you created.

```
10 REM * PROGRAM ID: LCREATE - Using a logical file database structure
20 ON ERR THEN GOTO 880
30 CLOSE
40 DIM LFTABL$[78],B$[544],X$[30],ER$[132],MKEY$[4],REC$[110]
50 DIM LAST$[24],FIRST$[16],ADDR$[25],CITY$[25],STATE$[3],ZIP$[5]
55 DIM PHONE$[8]
60 LET MKEY$,REC$=FILL$(0)
70 REM
80 REM ** Routine to open files
90 LET LFTABL$=FILL$(0)
100 LOPEN FILE[1,B$],"CLUB"
110 LOPEN FILE[2,B$],"MEMBER"
120 LOPEN FILE[3,B$],"NAME"
130 RFORM ZJLA24A16A25A25A3A5A8
140 REM
260 LET R1=-1
270 REM
280 REM ** Input variables
290 PRINT @(-30);@(4,15);"Input Screen"
300 PRINT
310 INPUT "Member #: ",MEM
320 INPUT "Last Name: ",LAST$
330 INPUT "First Name: ",FIRST$
340 INPUT "Address: ",ADDR$
350 INPUT "City: ",CITY$
360 INPUT "State: ",STATE$
370 INPUT "Zip: ",ZIP$
380 INPUT "Phone: ",PHONE$
390 REM
400 REM ** SETUP RECORD
410 PACK 130,REC$,1,MEM,LAST$,FIRST$,ADDR$,CITY$,STATE$,ZIP$,PHONE$
420 REM
510 PACK "L",MKEY$,MEM
520 REM
530 REM ** UPDATE FILES
540 GETREC 1,R1
545 IF R1<0 THEN GOTO 680
550 LET SR1=R1
560 LET T=30
570 LOCK 1,1,R1,T
580 IF T=57 THEN GOTO 560
590 REM
600 LWRITE FILE[1,R1],REC$
610 KADD 2,B$,MKEY$,R1
620 IF R1<=0 THEN GOTO 700
630 KADD 3,B$,LAST$,R1
640 IF R1<=0 THEN GOTO 760
650 UNLOCK
660 REM
```

```
670 GOTO 280
680 CLOSE
690 END
700 REM ** ERROR IN ADDING KEY TO MEMBER INDEX FILE
710 LET R1=SR1
720 DELREC 1,R1
730 PRINT @(20,5);"ERROR IN MEMBER INDEX-KEY & DATA RECORD NOT
    ADDED"
740 GOTO 840
750 REM
760 REM ** ERROR IN ADDING RECORD TO NAME INDEX
770 LET R1=SR1
780 DELREC 1,R1
790 LET R1=SR1
800 KDEL 2,B$,MKEY$,R1
810 PRINT @(20,5);"ERROR IN NAME INDEX - KEYS & DATA RECORD NOT ADDED"
820 GOTO 840
830 REM
840 REM ** ERROR ROUTINE FOR FILE HANDLING
850 PRINT @(21,5);"MEMBER NO.: ";MEM;"     Last Name:  ";LAST$
860 GOTO 970
870 REM
880 REM ** GENERAL ERROR ROUTINE
890 IF SYS(30)>63 AND SYS(7)<0 THEN
900   LET ER=SYS(31)
910   LET ER$=AERM$(SYS(31))
920 ELSE
930   LET ER=SYS(7)
940   LET ER$=ERM$(SYS(7))
950 END IF
960 PRINT @(20,5);"ERROR # ";ER;"= ";ER$
970 INPUT USING "",@(23,5),"RECORD ERROR INFORMATION & PRESS 'NL'",X$
980 CLOSE
990 END
```

Setting Up a PARAM File Database

Creating a PARAM file database involves multiple steps. You must design your database. If you do not have a PARAM file, you must create one and then use the File Maintenance (**FM**) utility to build record 0 of the file. (A detailed explanation of **FM** is supplied in the manual *Subroutines, Utilities, and the Business BASIC CLI.*) You must also run **INDEXCALC** to determine the number of keys per index block, the number of blocks needed for the index files, and the number of sectors needed for the data file. Next, you must create your subfiles and then run **INITFILE** to initialize them and enter the necessary information in the PARAM file.

To create a PARAM file, type in:

```
* !CRAND PARAM

* RUN "FM
```

When **FM** prompts you for a filename, type in **PARAM**. **FM** then clears the screen and displays a series of prompts for you to respond to. To set up record 0, you must add a record (for this example, press function key 4) and then supply the following answers:

```
Parameter record #          = 0
Sub file name               = PARAM
Master file name            = PARAM
Sub file position           = 0
Sub file record length      = 42
Last record number          = 100
Highest record number used  = 0
```

The subfile position prompt is asking for the byte pointer to the start of the subfile in the physical file; in this case, it is the beginning of the file. The last record number prompt refers to the number of subfiles that will be cataloged in the PARAM file. Since you are just setting up the file, no records have been used, which is why the highest record number used is set to 0.

When you finish building record 0, you must stop **FM** (in this example, press function key 10).

When you have the PARAM file set up, you can create your database. This database contains three subfiles and one master file. The subfiles consist of two index files, **MEMBER** and **NAME**, and one linked-available-record file, **CLUB**. They have been designed so that **MEMBER** uses a 4-byte key field and does not allow duplicate keys while **NAME** uses a 24-byte key field and permits duplicate keys. Both use a 50-percent blocking factor. **CLUB** holds 200 records, each 110 bytes long. The master file, **CLUBFILE**, is a random file.

Run **INDEXCALC** to calculate the numbers for the index files **MEMBER** and **NAME** (in that order). Running this utility also provides you with the number of sectors required by the data file **CLUB**.

```
* RUN "INDEXCALC
INDEXCALC VERSION X.XX

BYTES PER KEY : 4
BYTES PER DATA RECORD : 110
MAXIMUM NUMBER OF DATA RECORDS : 200
INDEX BLOCKING FACTOR (% PERCENT) [50]: 50
INDEX BLOCK SIZE (512 or 2048) [512]: 512
DUPLICATE KEYS ALLOWED? (Y or N) [N]: N

 63 MAXIMUM KEYS PER INDEX BLOCK
 32 KEYS PER BLOCK WITH A 50 PERCENT BLOCKING FACTOR
```

```
 7 BLOCK(S) AT LEVEL 1
 1 BLOCK(S) AT LEVEL 0
 9 BLOCKS (512 bytes each) IN INDEX
 9 SECTORS IN INDEX
 44 SECTORS IN DATA FILE

CALCULATE THE INDEX INFORMATION FOR ANOTHER FILE (Y or N) [N]: Y

BYTES PER KEY : 24
BYTES PER DATA RECORD : 110
MAXIMUM NUMBER OF DATA RECORDS : 200
INDEX BLOCKING FACTOR (% PERCENT) [50]: 50
INDEX BLOCK SIZE (512 or 2048) [512]: 512
DUPLICATE KEYS ALLOWED? (Y or N) [N]: Y

 18 MAXIMUM KEYS PER INDEX BLOCK
 9 KEYS PER BLOCK WITH A 50 PERCENT BLOCKING FACTOR
 23 BLOCK(S) AT LEVEL 2
 3 BLOCK(S) AT LEVEL 1
 1 BLOCK(S) AT LEVEL 0
 28 BLOCKS (512 bytes each) IN INDEX
 28 SECTORS IN INDEX
 44 SECTORS IN DATA FILE

CALCULATE THE INDEX INFORMATION FOR ANOTHER FILE (Y or N) [N]: N

*
```

Now run **INITFILE** to create these files.

```
* RUN "INITFILE
  INDEX (0), DATA (1), STOP (2) [0]: 0
  SUB FILE NAME MEMBER
  FILE NOT IN PARAM FILE!
  DO YOU WISH TO ADD? (Y OR N) [Y]: Y
  MASTER FILE NAME:  CLUBFILE
  INDEX BLOCK SIZE (512 or 2048) [512]: 512
  BYTE OFFSET TO SUB FILE: 0
  MAXIMUM NUMBER OF INDEX BLOCKS: 9
  BYTES PER KEY: 4
  BLOCKING FACTOR (% PERCENT) [50]: 50
  DUPLICATE KEYS ALLOWED? (Y OR N) [N]: N
  INDEX (0), DATA (1), STOP (2) [0]: 0
  SUB FILE NAME NAME
  FILE NOT IN PARAM FILE!
  DO YOU WISH TO ADD? (Y OR N) [Y]: Y
  MASTER FILE NAME: CLUBFILE
  INDEX BLOCK SIZE (512 or 2048) [512]: 512
  BYTE OFFSET TO SUB FILE: 4608
  MAXIMUM NUMBER OF INDEX BLOCKS: 28
  BYTES PER KEY: 24
```

```
BLOCKING FACTOR (% PERCENT) [50]: 50
DUPLICATE KEYS ALLOWED? (Y OR N) [N]: Y
INDEX (0), DATA (1), STOP (2) [0]: 1
SUB FILE NAME CLUB
FILE NOT IN PARAM FILE!
DO YOU WISH TO ADD? (Y OR N) [Y]: Y
MASTER FILE NAME: CLUBFILE
BYTE OFFSET TO SUB FILE: 18944
BYTES PER DATA RECORD: 110
MAXIMUM NUMBER OF DATA RECORDS: 200
SHOULD FILE BE NULL FILLED? N
INDEX (0), DATA (1), STOP (2) [0]: 2
```

*

You now have in your database the following files:

- A master file **CLUBFILE**. This is the only file on disk. It contains the subfiles **MEMBER**, **NAME**, and **CLUB**.
- An index file **MEMBER** with nine 512-byte blocks, a 4-byte member number field, a 50-percent blocking factor, and no duplicate keys.
- An index file **NAME** with 28 512-byte blocks, a 24-byte name field, a 50-percent blocking factor, and duplicate keys allowed.
- A linked-available-record file **CLUB** with 200 records of 110 bytes each.

You can use the following program to add records to the database you created.

```
10 REM ** PROGRAM ID:  CREATE
20 PRINT @(-30);@(5,20);"FILES ARE BEING OPENED"
30 CLOSE
40 DIM C1[2,3],B$[544],MX$[18],NX$[18]
50 DIM X$[30]
60 DIM LAST$[24],FIRST$[16],ADDR$[25],CITY$[25],STATE$[3],ZIP$[5],
PHONE$[8]
70 DIM MKEY$[4],NKEY$[24],LNAME$[24],FNAME$[24]
80 DIM REC$[110]
90 REM
100 REM ** Routine to OPEN files and set up the C1 array
110 LET B$="CLUB,5,MEMBER,5,NAME,5",FILL$(0)
120 BLOCK WRITE B$
130 SWAP "OPEN"
140 BLOCK READ B$
145 UNPACK "JJ",B$,ERRIN,ERRNO
150 IF ERRIN <>-1 THEN GOTO 180
160 PRINT "ERROR # ";ASC(B$[3,4]);" - ";B$[5,512]
170 END
180 LET K=1
190 FOR I=0 TO 2
200   FOR J=0 TO 3
```

```
210     LET C1[I,J]=ASC(B$[K,K+3])
220     LET K=K+4
230   NEXT J
240 NEXT I
250 LET MX$=CHR$(C1[1,0],2),CHR$(C1[1,1],4),CHR$(0,2),"MEMBER",
     FILL$(0)
260 LET NX$=CHR$(C1[2,0],2),CHR$(C1[2,1],4),CHR$(0,2),"NAME",
     FILL$(0)
270 LET F%,R1=0
280 REM
290 REM ** Input variables
300 INPUT "Member #: ",MEM
310 INPUT "Last Name: ",LAST$
320 INPUT "First Name: ",FIRST$
330 INPUT "Address: ",ADDR$
340 INPUT "City: ",CITY$
350 INPUT "State: ",STATE$
360 INPUT "Zip: ",ZIP$
370 INPUT "Phone: ",PHONE$
380 REM
: This example uses LET statements to set up the record; however, you
: could have used the PACK statement instead,which would reduce the
: amount of code you have.
390 REM ** SETUP RECORD
400 LET REC$[1,2]=CHR$(1,2),FILL$(0)
410 LET REC$[3,6]=CHR$(MEM,4)
440 LET REC$[7,30]=LAST$,FILL$[0]
450 LET REC$[31,46]=FIRST$,FILL$[0]
460 LET REC$[47,71]=ADDR$,FILL$[0]
470 LET REC$[72,96]=CITY$,FILL$[0]
480 LET REC$[97,99]=STATE$,FILL$[0]
490 LET REC$[100,104]=ZIP$,FILL$[0]
500 LET REC$[105,112]=PHONE$,FILL$[0]
510 LET REC$[113,120]=FILL$(0)
520 LET MKEY$=REC$[3,6]
530 LET NKEY$=REC$[7,30]
540 REM
550 REM ** UPDATE FILES
560 GOSUB 8400
570 LET SR1=R1
580 LET T=30
590 LOCK 1,"CLUB",R1*C1[F%,3],C1[F%,3],T
600 IF T=57 THEN GOTO 580
610 GOSUB 9610
620 WRITE FILE[C%],REC$
630 KADD MX$,B$,MKEY$,R1
640 IF R1<=0 THEN GOTO 720
650 KADD NX$,B$,NKEY$,R1
660 IF R1<=0 THEN GOTO 810
670 UNLOCK
680 REM
```

```
690 GOTO 290
700 CLOSE
710 END
720 REM ** ERROR IN ADDING KEY TO MEMBER INDEX FILE
730 LET R1=SR1
740 GOSUB 8600
750 PRINT @(20,5);"ERROR OCCURRED IN MEMBER INDEX - DATA RECORD NOT
ADDED"
760 PRINT @(21,5);"Member No.: ";MEM,"     Last Name:  ";LNAME$
770 INPUT USING "",@(23,5),"DEPRESS 'NL' TO RETURN TO MENU",X$
780 CLOSE
790 END
800 REM
810 REM ** ERROR IN ADDING RECORD TO NAME INDEX
820 LET R1=SR1
830 GOSUB 8600
840 LET MKEY$=REC$[3,6]
850 LET R1=SR1
860 KDEL MX$,B$,MKEY$,R1
870 PRINT @(20,5);"ERROR OCCURRED IN NAME INDEX; NO RECORD ADDED TO
DATA FILE"
880 PRINT @(21,5);"Member No.: ";MEM,"     Last Name:  ";LNAME$
890 INPUT USING "",@(23,5),"DEPRESS 'NL' TO RETURN TO MENU",X$
900 CLOSE
910 END
920 REM
8400 REM \ GETREC.SL
8405 LET C%=C1[F%,0]
8410 LET W0=C1[F%,1]
8415 POSITION FILE[C%,W0]
8420 READ FILE[C%],X0%,Y0,Z0
8430 IF Y0=-1 THEN GOTO 8472
8440 LET R1=Y0
8450 GOSUB 9610
8460 READ FILE[C%],Y0%,X0
8470 IF Y0%>0 THEN LET Y0%=1/0
8471 GOTO 8480
8472 IF Z0=-1 THEN GOTO 8494
8474 LET Z0=Z0+1
8475 LET X0=Y0
8476 LET Y0=Z0
8478 IF Z0>C1[F%,2] THEN GOTO 8496
8480 POSITION FILE[C%,W0]
8491 WRITE FILE[C%],X0%,X0,Z0
8492 LET R1=Y0
8493 RETURN
8494 LET R1=Z0
8495 RETURN
8496 LET R1=-1
8497 RETURN
8499 REM * END GETREC.SL
```

```
8600 REM \ DELREC.SL
8602 LET X0=R1
8605 IF R1<=0 THEN LET X0=1/0
8610 LET C%=C1[F%,0]
8615 LET W0=C1[F%,1]
8620 POSITION FILE[C%,W0]
8630 READ FILE[C%],X0%,Y0
8650 GOSUB 9610
8652 READ FILE[C%],Y0%
8655 IF Y0%<>0 THEN GOTO 8658
8656 PRINT @(25,50);"RECORD ALREADY DELETED"
8657 RETURN
8658 GOSUB 9610
8660 LET Y0%=0
8670 WRITE FILE[C%],Y0%,Y0
8680 POSITION FILE[C%,W0]
8692 WRITE FILE[C %],X0%%,X0
8695 RETURN
8699 REM * END DELREC.SL
9610 REM \ POSFL.SL
9611 LET V%%=0
9612 REM \ POSFL WITH OFFSET V%%
9613 LET C%%=C1[F%%,0]
9615 IF R1<0 THEN LET V%%=1/0
9620 IF R1>C1[F%%,2] THEN LET V%%=1/0
9625 LET R8=R1*C1[F%%,3]
9630 LET R9=C1[F%%,1]+R8+V%%
9640 POSITION FILE[C%%,R9]
9645 RETURN
9649 REM * END POSFL.SL
```

Comparing Logical, PARAM Code

There is a marked difference in the code needed to open files in the logical file database structure and the PARAM file database structure. The logical files require less code. These next two pieces of code illustrate opening files in the logical database and in the PARAM database.

Here is code to open files in the logical file database structure:

```
10 DIM LFTABL$[78],B$[544]
20 LET LFTABL$=FILL$(0)
30 LOPEN FILE[1,B$],"CLUB"
40 LOPEN FILE[2,B$],"MEMBER"
50 LOPEN FILE[3,B$],"NAME"
```

And here is code to open files in the PARAM file database structure:

```
10 DIM C1[2,3],B$[544],MX$[18],NX$[18]
20 LET B$="CLUB,5,MEMBER,5,NAME,5",FILL$(0)
30 BLOCK WRITE B$
40 SWAP "OPEN
50 BLOCK READ B$
60 IF ASC(B$[1,2])<>-1 THEN GOTO 0090
70 PRINT "ERROR # ";ASC(B$[3,4]);" - ";B$[5,512]
80 END
90 LET K=1
100 FOR I=0 TO 2
110   FOR J=0 TO 3
120     LET C1[I,J]=ASC(B$[K,K+3])
130     LET K=K+4
140   NEXT J
150 NEXT I
160 LET MX$=CHR$(C1[1,0],2),CHR$(C1[1,1],4),CHR$(0,2),"MEMBER",FILL$(0)
170 LET NX$=CHR$(C1[2,0],2),CHR$(C1[2,1],4),CHR$(0,2),"NAME",FILL$(0)
```

In terms of code, another difference between the logical structure and the PARAM structure is the use of the **DELREC** and **GETREC** statements as opposed to the **DELREC.SL**, **GETREC.SL**, and **POSFL.SL** subroutines. The logical structure uses the statements, which also perform automatic locking of record 0, while the PARAM structure uses the subroutines. The three subroutines occupy a total of 55 lines of code compared to the 2 lines required by the statements.

Enlarging a Logical Database

If you discover that your logical database is too small for your needs, you can enlarge it by following these steps:

1. Use **LFU PCREATE** to create a dummy database of the desired size.
2. Use **LFU LCREATE** to create a dummy logical file in this database for each logical file in your current database. You can increase the size of the files. (Note that the dummy logical file name for each file must be different from the original logical file name.)
3. Use the Business BASIC utility **LXFER** to copy each logical file in the current database to the corresponding logical file in the larger dummy database. You can only copy one file at a time. (**LXFER** is explained in the manual *Subroutines, Utilities, and the Business BASIC CLI.*)
4. Use **LFU PDELETE** to delete the old database.
5. Use **LFU PRENAME** to rename the dummy database to the old database name.
6. Use **LFU LRENAME** to rename the logical files to the same names that existed in the old database.

In this example, a database named CUSTDB needs to be enlarged

```
* !LFU PLIST CUSTDB

   DB file: CUSTDB
  File     File     Starting     # of      Record   Last      # of
  Name     Type      Sector     Sectors    Length  Record    Bytes
CUST         L           0        10        50      100       5050
CUSTNO       I          10         6       512        5       3072
                               -----                        ------
                 Total Sectors:    16              Bytes:    8122
```

Currently, this database holds 100 records. To expand the data file and the index file to allow 500 records, you run **INDEXCALC** to determine the number of sectors needed for the new database file. Then you create a dummy database file named **TEMPDB** and dummy logical files named **TCUST** and **TCUSTNO**.

```
* !LFU PCREATE TEMPDB 67

* !LFU LCREATE TCUST TEMPDB L 50 500 N

* !LFU LCREATE TCUSTNO TEMPDB I 512 17 4 50 N

* !LFU PLIST TEMPDB

   DB file: TEMPDB
  File     File     Starting     # of      Record   Last      # of
  Name     Type      Sector     Sectors    Length  Record    Bytes
TCUST        L           0        49        50      500      25050
TCUSTNO      I          49        18       512       17       9216
                               -----                        ------
                 Total Sectors:    67              Bytes:   34266
```

Next use the **LXFER** utility to copy the original logical files to the new logical files, and then delete the old database.

```
* !LXFER/V CUST TCUST                (Copy old data file into new file.)
5120 bytes transferred

* !LXFER/V CUSTNO TCUSTNO            (Copy old index file to the new one. The
3072 bytes transferred                last-available-block pointer in 0 is adjusted.)

* !LFU PDELETE CUSTDB                (Delete the old database.)

* !LFU PRENAME TEMPDB CUSTDB         (Rename new database with the old name.)

* !LFU LRENAME TCUST CUST            (Rename new data file with the old name.)

* !LFU LRENAME TCUSTNO CUSTNO        (Rename new index file with the old name.)

* !LFU PLIST CUSTDB                  (Get a listing of the enlarged database.)
```

```
   DB file: CUSTDB
  File     File     Starting     # of      Record    Last      # of
  Name     Type      Sector     Sectors    Length   Record    Bytes
CUST         L           0        49        50       500      25050
CUSTNO       I          49        18       512        17       9216
                               -----                        ------
               Total Sectors:    67                Bytes:   34266
```

To see how the database has changed, compare the output of **INDEXVRFY** for the **CUSTNO** index file before it was moved to the new database with the output of **INDEXVRFY** for the new **CUSTNO** index file. This is the output for the original **CUSTNO** file:

```
* !INDEXVRFY CUSTNO

    INDEX FILE NAME = CUSTNO

       ** VERIFYING **

Index file size              -  6  512 byte blocks
Number of index blocks used  -  5
Empty index blocks           -  0
Key length                   -  4    Duplicates not allowed
Max keys per block           -  63
Min key count                -  3
Max key count                -  37
Avg key count                -  26
Total keys at bottom level   -  100
Number of index levels       -  2
INDEX STRUCTURE VERIFIED CORRECT

  ******   VERIFY DONE   ******
```

This is the output from **INDEXVRFY** on the new **CUSTNO** index file:

```
* !INDEXVRFY CUSTNO

    INDEX FILE NAME = CUSTNO

       ** VERIFYING **

Index file size              -  18  512 byte blocks
Number of index blocks used  -  5
Empty index blocks           -  0
Key length                   -  4    Duplicates not allowed
Max keys per block           -  63
Min key count                -  3
Max key count                -  37
Avg key count                -  26
Total keys at bottom level   -  100
Number of index levels       -  2
INDEX STRUCTURE VERIFIED CORRECT

******   VERIFY DONE   ******
```

End of Appendix

Glossary

arithmetic operator
A symbol used in a numeric expression. The following symbols are valid operators: + (unary plus or addition), - (unary minus or subtraction), * (multiplication), / (division), and ^ (exponentiation). See also *expression*.

array
An ordered set of integer or string values (string arrays are allowed only on UNIX systems). Each array element is stored according to the precision of your Business BASIC system or the dimension of your string. Numeric arrays may have from two to eight dimensions, depending on your operating system, and string arrays may have eight dimensions. If you do not specify the dimensions for an array of integers, the interpreter uses a default value of 10 for each dimension; that is, a one-dimensional array has 11 elements. Default dimensions are not allowed with string arrays.

ASCII code
The decimal code number assigned to a character (unless octal is specifically stated). All printable and nonprintable characters on the terminal's keyboard have ASCII code numbers. See also *characters*.

attributes
All DG/RDOS files can have attributes. The BASIC CLI command **CHATR** changes a file's access attributes (permanent, read-protected, and write-protected). The BASIC CLI command **CHLAT** changes a link file's access attributes.

BASIC CLI (command line interpreter)
A utility program that emulates an operating system CLI. You can use the command **RUN**, **SWAP**, or **CHAIN** to start the BASIC CLI and then execute BASIC CLI commands interactively. Or you can swap to the CLI from a program that passes a command through the common area. The BASIC CLI prompt is an exclamation point.

BBPATH
A Business BASIC environment variable (UNIX systems only) that contains a list of directories that you want the interpreter to search for program and data files. **BBPATH** must also list the paths to the directories that contain tha page-table daemon and the Business BASIC error message file.

bit
An element of storage. Eight bits make 1 byte, and 2 bytes make 1 word. Use the **SHFT**, **AND**, and **OR** functions to move bits around in a word or to compare the bits in one word to those in another (sometimes called bit checking).

Boolean logic operators
Operators that evaluate an expression as either true (1) or false (0). Business BASIC supports the Boolean logic operators NOT, AND, and OR.

byte
A sequence of eight adjacent bits (locations 0 to 7).

C1 array
Used with the PARAM file database structure. The C1 array is a Business BASIC convention necessary for using the **GETREC.SL**, **DELREC.SL**, and **POSFL.SL** subroutines, which provide fast access to subfiles. The C1 array contains the channel number of the physical file for each subfile, the byte offset in the physical file to the beginning of each subfile, the file size, and the record size. It is also called the file characteristics array.

carriage return (CR)
A key on some terminals that is used to signal the end of an entry or terminate a line. This manual uses the term *New Line key* to refer to the CR, the New Line, and the Enter key. When the manual instructs you to press the New Line key, press the line terminator key that is appropriate for your system.

character
A character is stored as an ASCII code in 1 byte (see also *byte*). The term *character data* is used to refer to string literals.

command
An instruction directing the system to do something immediately. You execute BASIC commands from keyboard mode (indicated by an asterisk prompt). Some BASIC statements can also be commands.

common area
A storage area unique to each job (process). You use it to store information temporarily so that you can swap to another program that picks up that information. To access the common area, use the **BLOCK READ** and **BLOCK WRITE** statements/commands.

concatenation operator
You can concatenate strings by using a comma to separate them in an assignment statement.

database file
A physical file in the logical file database structure that contains logical files (subsections). A database file is indicated by a **.DB** extension on its filename and is always associated with a volume label file.

delimiter
A character that indicates the start of a character string. Most keyboard editing commands require delimiters. Generally, the delimiter can be any character that is not part of the character string. When a command uses more than one delimiter, the delimiter must be same in each place it is used in the command line.

device name
The name assigned to a device, such as a terminal, line printer, or tape drive. In UNIX, device filenames are generally found in the **/dev** directory. For example, the pathname for your terminal may be **/dev/tty6**. Under AOS/VS the names usually begin with an at (@) sign. For example, **@LPT** is the line printer. Under RDOS, devices have three- or four-character names that begin with a dollar sign or end with a number. For example, **$LPT** is a line printer, and **UT10** refers to a magnetic tape drive.

directory
A file that contains entries (pointers) to other files. In UNIX you can tell whether a file is a directory by typing the command **ls -l** *filename*. If the first character in a line of output is a *d*, the file described on that line is a directory. Under AOS/VS directories have the file type DIR or CPD. Under RDOS, directory names have a **.DR** extension that you need to specify only if you are accessing the directory as a file. You do not have to specify the **.DR** extension when referring to the file in a pathname.

dynamic allocation
A form of allocating records in a file by allowing the file to reuse space left by deleted records. To allow dynamic allocation, a file maintains a deleted-record chain to indicate which records can be reused. Business BASIC includes the commands **GETREC** and **DELREC** (logical database) and the subroutines **GETREC.SL** and **DELREC.SL** (PARAM database) to allow you to allocate and deallocate records in linked-available-record database files. See also *linked-available-record format*.

edit buffer
A buffer that is used by the keyboard editing commands **.P**, **.A**, **.C**, **.E**, and **.I** to modify a single line. The edit buffer contains the last statement listed or the last statement that caused an error message to appear.

enter
The act of typing in data and then pressing a line terminator key to signal the end of the input, thus sending the data to the system. For example, to enter a program statement, you type in the statement line number followed by the statement contents and then press the New Line, CR, or Enter key. You can enter data while in keyboard mode or in response to a program query. Business BASIC also provides a command/statement called **ENTER** that you use to place programs in working storage.

expression
A numeric expression is a combination of numbers, numeric variables, numeric array elements, and numeric functions linked by some combination of arithmetic operators, relational operators, Boolean logic operators, and parentheses. A string expression is a combination of string literals, string variables, string array elements, substrings, and string functions separated by the concatenation operator (,).

file
A collection of data. Business BASIC uses program files, listing files, source files, text files, data files, subfiles of data files, logical files, index files, tag files, table files, log files, documentation files, and screen files. Directories and devices are also files.

file characteristics (C1) array
See *C1 array*.

filename
A name that refers to a file. Filenames in DG/RDOS can have up to 10 alphanumeric characters and an extension of up to two characters after a period. Filenames in UNIX and AOS/VS can be longer (see the manual that explains how to use Business BASIC on your system for details). However, some Business BASIC utilities require that filenames follow the DG/RDOS length conventions. These utilities truncate filenames that exceed this limit. The interpreter does not warn you when it truncates a filename.

ISAM file
A file that uses the Indexed Sequential Access Method to access records. An ISAM file set comprises two files. The data records, each of which contains a unique key field, are in a data file. Another file, called the index file, is made up of keys and pointers to the records in the data file that have those keys.

interrupt key
The key(s) you press to interrupt the execution of a program or command. It is frequently the Esc key. You can define interrupt keys using **STMA 4,6** and **STMA 4,7**. This procedure is explained in the manual *Commands, Statements, and Functions in Business BASIC*.

keyboard mode
A Business BASIC operational mode. In keyboard mode, you can enter a command for immediate execution or enter program statements to create a program in working storage. The asterisk prompt indicates that you are in keyboard mode.

library
The Business BASIC library contains prewritten subroutines and utilities. On UNIX systems, these subroutines and utilities are located in the directory **lb**. On other platforms they are located in the directory **$SYSLIB** (or **$SYSLIB3** for triple-precision systems).

linked-available-record format
A method of disk storage that provides dynamic record allocation for random files. It allows a file to reuse space left by deleted records. Each record contains a pointer that points to the next available record in the file.

listing file or *list file*
A text file. This can be a file created by the **LIST** command (that is, an ASCII listing of the program currently in working storage) or a file you created in an editor. You can use the BASIC **EDIT** utility or any text editor to modify a listing file. You use the command **ENTER** to bring a listing file into working storage.

log file
A file containing a record of all the changes you made to a data file while using the **FM** (File Maintenance) utility. Each data file can have a log file defined by its table file. Use the utility **FMLOG** to display the contents of this log file.

logging in
The process of executing Business BASIC.

logging out
The process of terminating Business BASIC by typing **BYE**.

logical file
A logical subsection of a physical file. The term *logical file* is used to refer to files in the logical file database structure.

logical file database structure
One of Business BASIC's two database structures. The logical file database allows you to open more than the maximum number of files you are normally allowed to open in one program. You do this by opening logical files, which are subsections of physical files. See also *PARAM file database structure*.

master file
A physical file in the PARAM file database structure that contains subfiles.

New Line
A key used on some terminals to signal the end of an entry or terminate a line. This manual uses the term *New Line key* to represent the Enter, CR, or New Line key. When you see *New Line*, use the line terminator key that is appropriate for your system.

numeric function
A function that always returns a numeric value. It can be used in a numeric expression or in an assignment statement.

PARAM file
A file that contains the information you need for using subfiles and the **OPEN** utility. The PARAM file for your system can be in the library directory, or you can set up a PARAM file in a directory you specify. You can have more than one PARAM file.

PARAM file database structure
One of Business BASIC's two databases. The PARAM file database structure allows you to open more than the maximum number of files you are normally allowed to open in one program. You do this by opening subfiles, which are subsections of physical files. See also *logical file database structure*.

PATH
A UNIX environment variable that lists the directories the shell should search for executable files.

pathname
A combination of one or more directory names and a filename that unambiguously identifies the location of a file in a file system.

precision
Indicates the number of bytes the interpreter uses to store or transfer a numeric value. Depending on your operating system, you may have to choose between a double-precision and triple-precision system at system-generation time, or you may automatically create a quadruple-precision system. A double-precision system stores numeric values in 4 bytes and transfers numeric values using either 2 bytes or 4 bytes. A triple-precision interpreter stores numeric values in 6 bytes and can transfer numeric data in 2-byte, 4-byte, or 6-byte units. A quadruple-precision system uses 8-byte segments to store numbers and transfers numeric values using either 2, 4, 6, or 8 bytes.

prompt
A character output to your terminal to signify that you must enter something. An asterisk prompt indicates that you are in BASIC's keyboard mode. An exclamation point prompt signifies that you are running the BASIC CLI program. Another prompt is the line number prompt (000:) used in the **EDIT** utility.

record
A collection of related data fields that are treated as one unit.

relational operator
A symbol used to compare two expressions. The relational operators are = (equal to), < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to), and <> (not equal to).

save file
A program file that is stored in a metacode format. You create a save file when you use the **SAVE** or **REPLACE** command to write the program in working storage to disk.

screen field
An area on your terminal's screen. Use the Conversational Screen Maintenance (**CSM**) utility or the Screen Maintenance (**SM**) utility to define screen fields in screen files.

screen file
A file created by the Conversational Screen Maintenance (**CSM**) utility or the Screen Maintenance (**SM**) utility. It usually has an **.S***n* extension, where *n* is the terminal type. A screen file can hold multiple screens.

search list
A list of directories that the AOS/VS CLI scans if it cannot find the file you want in the current directory. You can display and change your search list with the AOS/VS CLI command **SEARCHLIST**. See also *pathname*.

source file
A Business BASIC program that is in ASCII format (created with the **EDIT** utility or an editor). Source files usually contain comments.

statement
An instruction in a program. Each Business BASIC statement contains a line number and a Business BASIC keyword, and most statements include arguments.

string
A combination of characters (letters, digits, spaces, and special characters).

subfile
A file that exists as a logical subsection of a physical file. To set up and use subfiles, you need a record in the PARAM file for the subfile and a file characteristics (C1) array in your program.

subroutine
A section of Business BASIC code that performs a specialized task. Subroutines are executed by the **GOSUB** command.

table file
A file used by the **FM** utility. A table file contains information on a data file and up to three index files. The table file holds heading information, record and file sizes, a list of valid users, record format descriptions, field descriptions, and key descriptions.

tag file
A temporary index file that is accessed like a regular file. Its records are fixed in length, consisting of a key (string value) and record pointer (integer value). You usually create a tag file in sorted order using the **TBUILD** utility.

template
A special character that a shell or command line interpreter takes to represent, as examples, any character or any string of characters. Generally, you use a template (or wildcard) character as part of a filename. In this way, you can create an ambiguous file reference, one that may apply to any number of files.

utility
A Business BASIC program that performs a specialized task. You can run, chain to, or swap to most utility programs, or execute them from the BASIC CLI. Others require that you pass arguments through the common area; you can only swap to these.

variable
A named segment of memory in which your program stores a value. There are four kinds of variables: numeric, numeric array, string, and string array. (Not all interpreters support string arrays.) You assign values to variables using **LET, INPUT, READ** and **DATA, INPUT USING, TINPUT, PACK, UNPACK,** and **PRINT USING**. Variable names must begin with a letter, which can be followed by from 5 to 31 alphanumeric characters, depending on your operating system. In addition, the name of a numeric variable may end with a special sign (%, #, &) that indicates the precision of the integer stored in that variable.

volume label file
A file used in the logical file database structure to hold the information about the logical file. The volume label file maps the logical file to the database file, which contains the actual data. A volume label file is indicated by a **.VL** extension.

word
A sequence of two adjacent bytes (bit locations 0-15).

working storage
The portion of memory used to develop programs and where programs that are entered or loaded are stored. All programs and data stored in working storage are stored in save file (metacode) format. You can restore a program to ASCII format by using the **LIST** command to write the code to a file.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

TIPS ORDERING PROCEDURES

TO ORDER

1. An order can be placed with the TIPS group in two ways:
   a) MAIL ORDER – Use the order form on the opposite page and fill in all requested information. Be sure to include shipping charges and local sales tax. If applicable, write in your tax exempt number in the space provided on the order form.

      Send your order form with payment to:
      Data General Corporation
      ATTN: Educational Services/TIPS G155
      4400 Computer Drive
      Westboro, MA 01581–9973

   b) TELEPHONE – Call TIPS at (508) 870–1600 for all orders that will be charged by credit card or paid for by purchase orders over $50.00. Operators are available from 8:30 AM to 5:00 PM EST.

METHOD OF PAYMENT

2. As a customer, you have several payment options:
   a) Purchase Order – Minimum of $50. If ordering by mail, a hard copy of the purchase order must accompany order.
   b) Check or Money Order – Make payable to Data General Corporation.
   c) Credit Card – A minimum order of $20 is required for Mastercard or Visa orders.

SHIPPING

3. To determine the charge for UPS shipping and handling, check the total quantity of units in your order and refer to the following chart:

Total Quantity	Shipping & Handling Charge
1–4 Units	$5.00
5–10 Units	$8.00
11–40 Units	$10.00
41–200 Units	$30.00
Over 200 Units	$100.00

   If overnight or second day shipment is desired, this information should be indicated on the order form. A separate charge will be determined at time of shipment and added to your bill.

VOLUME DISCOUNTS

4. The TIPS discount schedule is based upon the total value of the order.

Order Amount	Discount
$1–$149.99	0%
$150–$499.99	10%
Over $500	20%

TERMS AND CONDITIONS

5. Read the TIPS terms and conditions on the reverse side of the order form carefully. These must be adhered to at all times.

DELIVERY

6. Allow at least two weeks for delivery.

RETURNS

7. Items ordered through the TIPS catalog may not be returned for credit.
8. Order discrepancies must be reported within 15 days of shipment date. Contact your TIPS Administrator at (508) 870–1600 to notify the TIPS department of any problems.

INTERNATIONAL ORDERS

9. Customers outside of the United States must obtain documentation from their local Data General Subsidiary or Representative. Any TIPS orders received by Data General U.S. Headquarters will be forwarded to the appropriate DG Subsidiary or Representative for processing.

TIPS ORDER FORM

Mail To: Data General Corporation
Attn: Educational Services/TIPS G155
4400 Computer Drive
Westboro, MA 01581 - 9973

BILL TO:	SHIP TO: (No P.O. Boxes - Complete Only If Different Address)
COMPANY NAME _____	COMPANY NAME _____
ATTN: _____	ATTN: _____
ADDRESS _____	ADDRESS (NO PO BOXES) _____
CITY _____	CITY _____
STATE _____ ZIP _____	STATE _____ ZIP _____

Priority Code _____ (See label on back of catalog)

_____ _____ _____ _____ _____
Authorized Signature of Buyer (Agrees to terms & conditions on reverse side) | Title | Date | Phone (Area Code) | Ext.

ORDER #	QTY	DESCRIPTION	UNIT PRICE	TOTAL PRICE

A SHIPPING & HANDLING

☐ UPS	ADD
1-4 Items	$ 5.00
5-10 Items	$ 8.00
11-40 Items	$ 10.00
41-200 Items	$ 30.00
200+ Items	$100.00

Check for faster delivery

Additional charge to be determined at time of shipment and added to your bill.
☐ UPS Blue Label (2 day shipping)
☐ Red Label (overnight shipping)

B VOLUME DISCOUNTS

Order Amount	Save
$0 - $149.99	0%
$150 - $499.99	10%
Over $500.00	20%

Tax Exempt # or Sales Tax (if applicable) _____

ORDER TOTAL	
Less Discount See B	-
SUB TOTAL	
Your local* sales tax	+
Shipping and handling - See A	+
TOTAL - See C	

C PAYMENT METHOD

☐ Purchase Order Attached ($50 minimum)
P.O. number is_____. (Include hardcopy P.O.)
☐ Check or Money Order Enclosed
☐ Visa ☐ MasterCard ($20 minimum on credit cards)

Account Number | Expiration Date

_____
Authorized Signature
(Credit card orders without signature and expiration date cannot be processed.)

THANK YOU FOR YOUR ORDER

PRICES SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.
PLEASE ALLOW 2 WEEKS FOR DELIVERY.
NO REFUNDS NO RETURNS.

* Data General is required by law to collect applicable sales or use tax on all purchases shipped to states where DG maintains a place of business, which covers all 50 states. Please include your local taxes when determining the total value of your order. If you are uncertain about the correct tax amount, please call 508-870-1600.

Form 702
Rev. 8/87

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form. These terms and conditions apply to all orders, telephone, telex, or mail. By accepting these products the Customer accepts and agrees to be bound by these terms and conditions.

1. CUSTOMER CERTIFICATION
Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

2. TAXES
Customer shall be responsible for all taxes, including taxes paid or payable by DGC for products or services supplied under this Agreement, exclusive of taxes based on DGC's net income, unless Customer provides written proof of exemption.

3. DATA AND PROPRIETARY RIGHTS
Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

4. LIMITED MEDIA WARRANTY
DGC warrants the CLI Macros media, provided by DGC to the Customer under this Agreement, against physical defects for a period of ninety (90) days from the date of shipment by DGC. DGC will replace defective media at no charge to you, provided it is returned postage prepaid to DGC within the ninety (90) day warranty period. This shall be your exclusive remedy and DGC's sole obligation and liability for defective media. This limited media warranty does not apply if the media has been damaged by accident, abuse or misuse.

5. DISCLAIMER OF WARRANTY
EXCEPT FOR THE LIMITED MEDIA WARRANTY NOTED ABOVE, DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS, CLI MACROS OR MATERIALS SUPPLIED HEREUNDER.

6. LIMITATION OF LIABILITY
A. CUSTOMER AGREES THAT DGC'S LIABILITY, IF ANY, FOR DAMAGES, INCLUDING BUT NOT LIMITED TO LIABILITY ARISING OUT OF CONTRACT, NEGLIGENCE, STRICT LIABILITY IN TORT OR WARRANTY SHALL NOT EXCEED THE CHARGES PAID BY CUSTOMER FOR THE PARTICULAR PUBLICATION OR CLI MACRO INVOLVED. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO CLAIMS FOR PERSONAL INJURY CAUSED SOLELY BY DGC'S NEGLIGENCE. OTHER THAN THE CHARGES REFERENCED HEREIN, IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOST PROFITS AND DAMAGES RESULTING FROM LOSS OF USE, OR LOST DATA, OR DELIVERY DELAYS, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY THEREOF; OR FOR ANY CLAIM BY ANY THIRD PARTY.

B. ANY ACTION AGAINST DGC MUST BE COMMENCED WITHIN ONE (1) YEAR AFTER THE CAUSE OF ACTION ACCRUES.

7. GENERAL
A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts, excluding its conflict of law rules. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer. DGC hereby rejects all such different, conflicting, or additional terms.

8. IMPORTANT NOTICE REGARDING AOS/VS INTERNALS SERIES (ORDER #1865 & #1875)
Customer understands that information and material presented in the AOS/VS Internals Series documents may be specific to a particular revision of the product. Consequently user programs or systems based on this information and material may be revision-locked and may not function properly with prior or future revisions of the product. Therefore, Data General makes no representations as to the utility of this information and material beyond the current revision level which is the subject of the manual. Any use thereof by you or your company is at your own risk. Data General disclaims any liability arising from any such use and I and my company (Customer) hold Data General completely harmless therefrom.

Cut here and insert in binder spine pocket

Data General

Data General Corporation, Westboro, Massachusetts 01580

093-000684-00