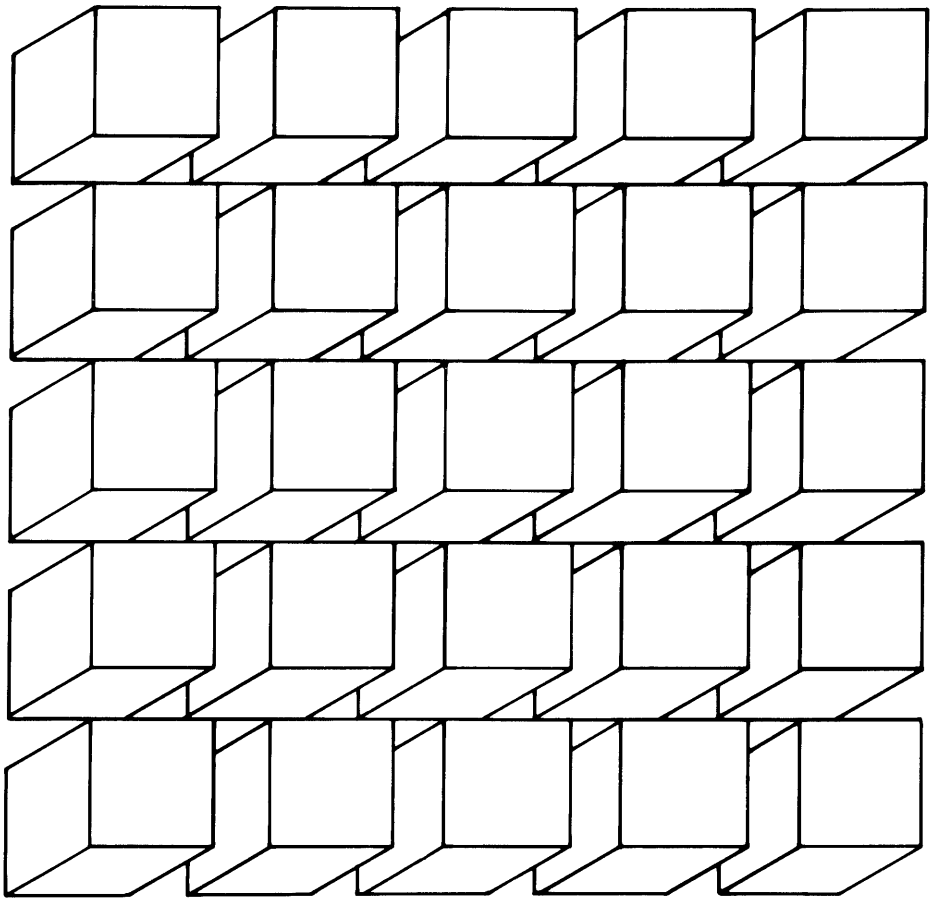
A Guide to Using Business BASIC (AOS/VS, AOS, RDOS, DOS)



DataGeneral

069-000028-01

A Small Business Systems Publication

DATA GENERAL CORPORATION, Westboro, Massachusetts 01580

Notice

Data General Corporation (DGC) has prepared this document for use by DGC personnel, customers, and prospective customers. The information contained herein shall not be reproduced in whole or in part without DGC's prior written approval.

DGC reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult DGC to determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF DGC HARDWARE PRODUCTS AND THE LICENSING OF DGC SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN DGC AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS DOCUMENT INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY DGC FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY OF DGC WHATSOEVER.

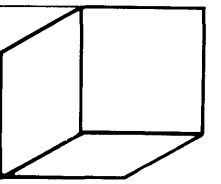
IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS DOCUMENT OR THE INFORMATION CONTAINED IN IT, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY OF SUCH DAMAGES.

Ordering No. 069-000028

Printed in the United States of America
Rev. 01, December 1982

069-000028-01

Table of Contents



Chapter 3
Numeric, Array, and String Variables

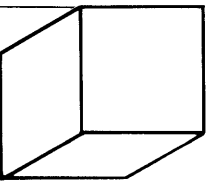
Chapter 4
Subroutines and Utilities

Chapter 5
Business BASIC File Overview

Chapter 6
Data Base Generator DBGEN, FM, and SM

Related Documents

Preface



Scope

This guide is intended as an introduction to Data General's Business BASIC, a powerful, interactive programming language. After reading it, the business application programmer will be familiar with the entire range of features built into Business BASIC and how these various features can be used in developing application programs.

Business BASIC is available under a wide range of operating systems: AOS/VS, AOS, RDOS, DOS, and MP/OS. This manual is applicable to all these operating systems with the exception of MP/OS. Those programming in Business BASIC under MP/OS should consult the equivalent "Guide", 069-400439.

Detailed information on the topics introduced in this manual is available in the other Business BASIC manuals listed in the Related Documents section.

Organization

This guide is divided into six chapters.

Chapter 1 is an introduction to Business BASIC. It gives general descriptions of the components and structure of the entire software package.

Chapter 2 describes the operator sequences involved in developing application programs with Business BASIC. Beginning with the "log-on" procedure at the terminal, this chapter covers all aspects of generating BASIC routines including writing, editing and storing application programs.

Chapter 3 covers characteristics and constraints of the numeric variables, arrays, and string variables used with Business BASIC.

Chapter 4 defines the types of subroutines and utilities included with Business BASIC. Requirements for user-written Business BASIC subroutines and assembly language subroutines are also discussed.

Chapter 5 describes the types of file structures as well as file accessing and manipulation techniques.

Chapter 6 gives a description of the procedures for using the main Business BASIC utility programs, namely Screen Maintenance(SM), the Data Base Generator, and File Maintenance (FM).

Typographical Conventions

A format is provided for some statements, commands and functions. In these formats, bold face type is used to indicate required elements. Italic type is used to indicate optional elements.

In the examples that are provided, the Business BASIC prompt (*) is used to indicate the beginning of a command or program statement, just as it does on the screen. To enter a statement or command, the user must press New Line or Carriage Return, depending on the system. In the examples, user input is distinguished from system response with these type faces:

```
This is user input.
```

This is system response.

Chapter 1
Introduction

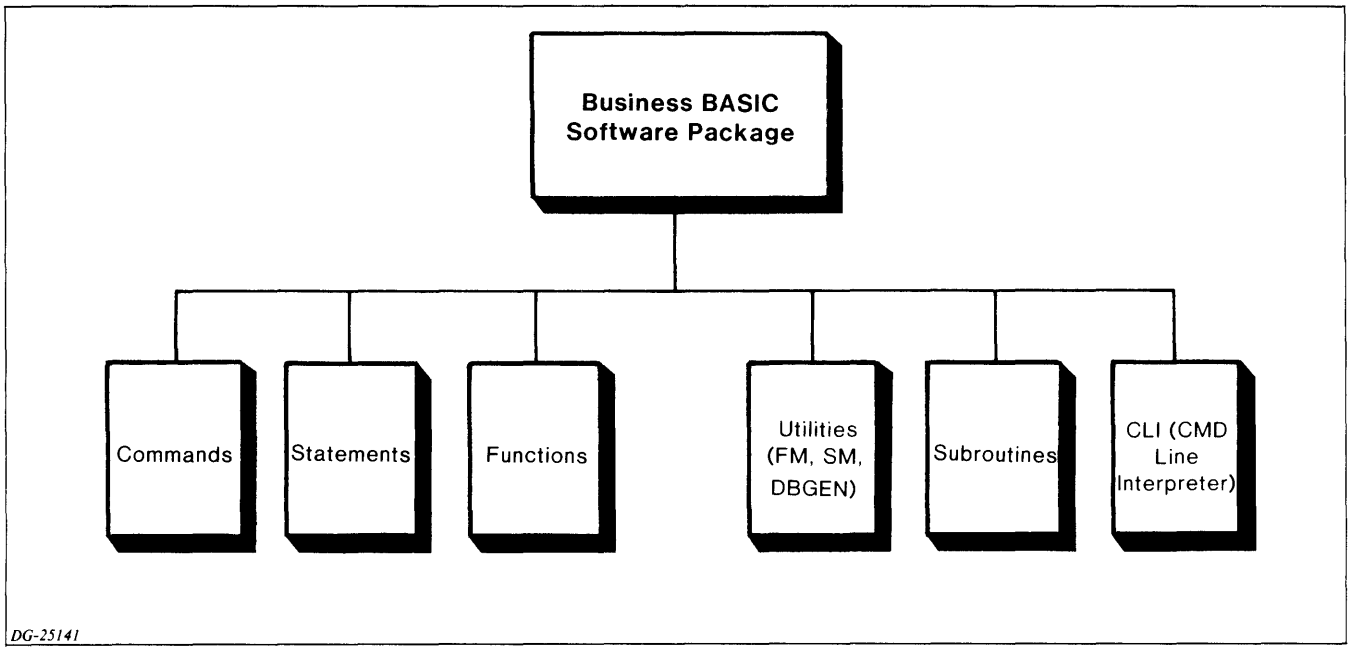


Figure 1-1 *Business BASIC Software Package*

This chapter describes the principal components of Business BASIC. It provides an overview of the structure and composition of the entire Business BASIC software package. Figure 1-1 is a representation of the major components of the Business BASIC software package.

All of the boxes in Figure 1-1 represent tools that you, the Business BASIC programmer, have at your disposal. They can be summarized as follows:

1. The three boxes at the left (Commands, Statements, Functions) make up the Business BASIC language. They are your tools for designing the application programs that perform your intended business applications.
2. Business BASIC incorporates a number of utility programs that perform a wide variety of functions. The main utilities (SM, DBGEN, FM) are programs designed to aid you in formatting screens, and structuring your database for optimum data access. Another important utility, the EDIT utility, is an editor which can be invoked for program update.
3. The subroutines within BASIC are specialized programs that are modularly designed so that they can be easily incorporated into a user's application program.
4. Command Line Interpreter(CLI). This is a Business BASIC feature (which is available on all operating systems) that can be used to simulate the RDOS/DOS CLI. This provides the advantage of not having to exit Business BASIC to perform such functions as print a file to a line printer, display the name of the current directory, or list the files in the current directory.

Commands, Statements, and Functions

Commands, statements, and functions are the fundamental tools used to design and develop application programs in Business BASIC. By definition, a "command" is an instruction that is entered without a line number and executed immediately. A "statement" is preceded by a line number and not executed until a command such as RUN is entered. A "function" can be either numeric or string and is used as an expression within statements or commands.

Writing Business BASIC Programs

The general syntax for each line of coding in a Business BASIC program is shown in Figure 1-2.

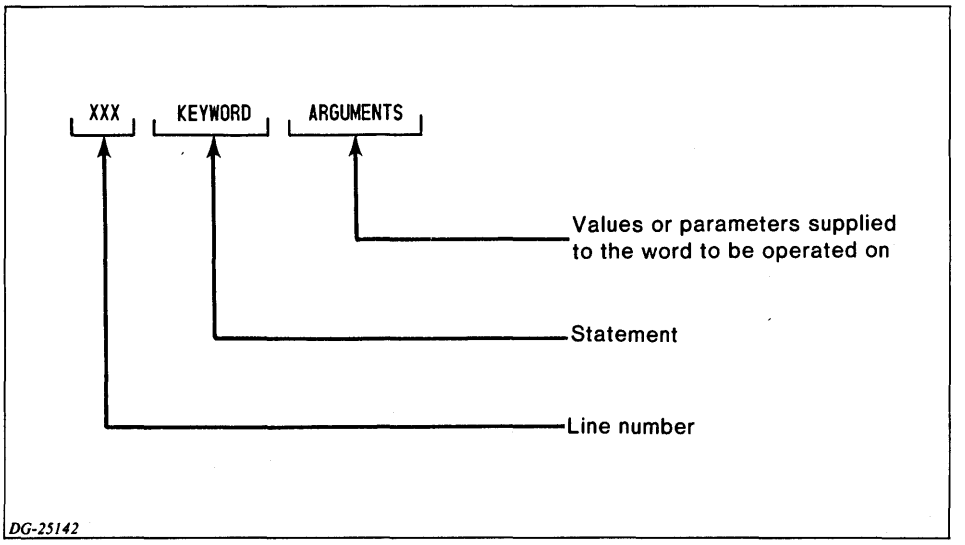


Figure 1-2 *Business BASIC Coding Line Syntax*

In almost all cases, a line of Business BASIC coding consists of the following:

1. XXX (LINE NUMBER). Every line of coding must begin with a line number.
2. Statement. Following the line number is the statement. Statements are the instructions that tell the computer to perform some action.
3. ARGUMENTS. In almost all cases, statements are followed by arguments (also known as parameters). These consist of variables, numeric/string assignments, messages to print, and subroutine destinations.

A typical line of Business BASIC coding is shown in Figure 1-3. Here we have a line number followed by the LET statement followed by a numeric assignment (a value of 500 is assigned to the letter A).

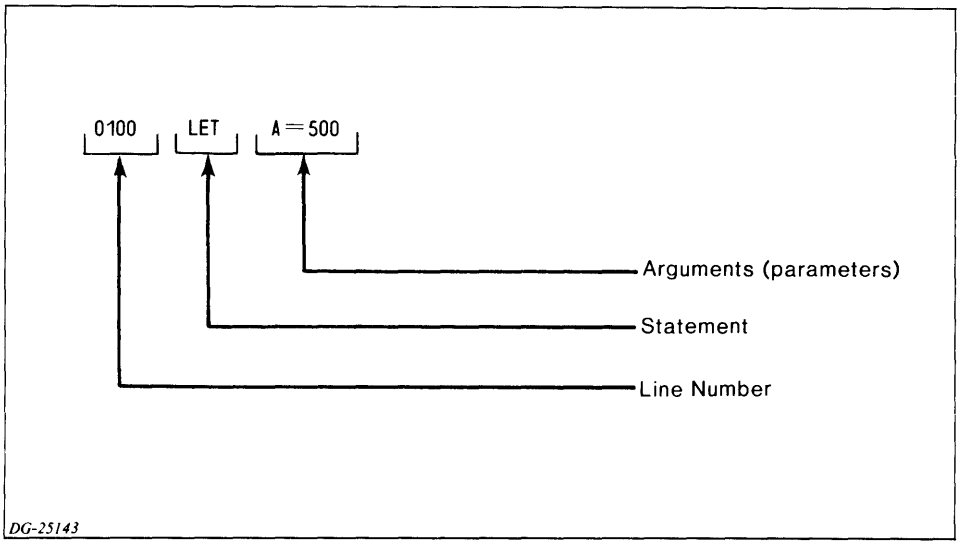


Figure 1-3 *Typical Line of Business BASIC Coding*

Two LET statements followed by a PRINT statement (that directs the program to print the product of the two LET statements) can be coded to form the very simple BASIC program shown below:

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

Typing the RUN command causes the program to be executed. In this case the product of the two variables is displayed on the screen. The above example is included here to introduce you to BASIC programming in its simplest form. More complex examples showing a wider range of command/statement/function usage are presented in Chapter 2.

Business BASIC Utility Programs

The Business BASIC software features numerous utility programs to aid you in a wide variety of data processing functions. However, we only introduce the more important ones here because they are most frequently used and play a greater role in the business applications world. The principal utilities are:

1. Data Base Generator (DBGEN)
2. File Maintenance (FM) Utility
3. Screen Maintenance (SM) Utility
4. EDIT Utility

Data Base Generator (DBGEN)

The DBGEN (Data Base Generator) program is an interactive program written in Business BASIC. It prompts the user for the information needed to create a database of one data file with up to three index files, and then initializes the files and constructs a table file automatically. The DBGEN program is described in full in the Business BASIC Subroutines, Utilities, and BASIC CLI manual.

File Maintenance (FM) Utility

The File Maintenance (FM) utility is used primarily to update files within a prestructured database. FM can be invoked to define fields for any files a user might create on his/her own. Use of FM assumes that all files have been previously initialized.

The File Maintenance utility has its own set of function key definitions (and related function key template) for searching through the database file structure and locating records. The function key definitions become operable whenever the FM utility is invoked.

The File Maintenance utility can be invoked to update files in any database structured via the Data Base Generator (DBGEN) or in databases you structure with individual Business BASIC utilities. In either case, the database must include a table file since the table file contains parameters necessary for File Maintenance utility operation.

Screen Maintenance (SM) Utility

The Screen Maintenance (SM) utility is used to format one or more screens for data entry and retrieval purposes. This utility provides a convenient means of dividing the screen into fields and columns, assigning column headings, labeling fields that are to receive input data from keyboard operators, etc.

The Screen Maintenance utility has its own specialized function (and accompanying function key template) for formatting screens. These function key definitions become operable whenever the SM utility is invoked.

When operating under the Screen Maintenance (SM) utility, the goal is to design a screen display that will effectively prompt the application user for input and provide intelligible output. In many respects, it is similar to designing a form that is to be printed. The goal is to design a clean, easily understood display which will assist rather than confuse the user.

The DBGEN, FM, and SM utilities are described in greater detail in Chapter 6 of this Guide.

EDIT Utility

The Edit utility is used to update or append comments to a previously generated Business BASIC application program. It is especially useful when users wish to insert explanatory comments to the right of each line of BASIC coding. The Edit utility is described in detail in the Business BASIC Subroutines, Utilities, and BASIC CLI manual.

Business BASIC Subroutines

The Business BASIC software package features a variety of modular subroutines that you can incorporate from within your Business BASIC application program. Provisions to accommodate user-written Business BASIC subroutines and assembly language subroutines are also included.

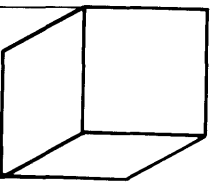
The subroutines within Business BASIC are specialized programs that are modularly designed for incorporation within an application program if the user so desires. For example, one particular subroutine (MUL.SL) calculates a percentage of a number. If this sort of processing function is required in your application, you need only incorporate the subroutine without going to the effort of coding your own subroutine to perform this function.

Command Line Interpreter (CLI)

Business BASIC is equipped with its own Command Line Interpreter that allows users to execute many of the commands which are built into the RDOS CLI. The program which is used to simulate the RDOS CLI is referred to as the "BASIC CLI". The BASIC CLI is available on AOS/VS, AOS, RDOS, DOS, and MP/OS versions of Business BASIC. This means that you need not exit Business BASIC to execute such processing activities as file creation/deletion, inter-directory file moves, and printing of files to obtain hard copy. Instead you need only invoke the Business BASIC CLI (RUN "CLI) and all such processing functions are executable from within Business BASIC.

If you are using Business BASIC on an AOS/VS or AOS system, you have the additional capability of the AOS CLI. When you use the AOS CLI you actually start up a son process which executes the AOS CLI.

Chapter 2
Program Development



This chapter describes the operational sequences carried out by a Business BASIC programmer from the time of log-on at the terminal through the point of program execution and then to eventual storage and documentation of the program. The primary steps in the overall application program development effort are:

- Logging on/off the system
- Use of keyboard mode and working storage
- Writing/editing Business BASIC programs
- Executing Business BASIC programs
- Interrupting and debugging programs
- Storing and documenting programs

Logging On/Off the System

The Business BASIC log-on/off procedure varies depending on the operating system. That is, an AOS system requires one log-on sequence while an RDOS system requires another.

Entering Business BASIC (RDOS, DOS)

The entry sequence that follows assumes that the system is up, your terminal is on, and a banner similar to the one shown below is displayed on the screen. (If not, consult your system manager.)

```
DATA GENERAL RDOS/DOS BUSINESS BASIC - REVISION #.## dd-mm-yy
```

Press the ESCape (ESC) key to begin the RDOS/DOS Business BASIC log-on procedure. Pressing the ESCape key invokes the HELLO program which now initiates the following dialogue:

```
TERMINAL TYPE:
```

RDOS/DOS recognizes Data General DASHER® display terminals as terminal type 6, so if you are using this type, the HELLO program omits the "terminal type" question. (This is true for both DASHER® D2s and DASHER® D200s.) If you are using DASHER hard copy terminals, you should respond to the "TERMINAL TYPE:" query by pressing 2 followed by pressing CR. If you are using DGC 6012 terminals, respond by pressing 3 followed by CR. (More detailed information on terminal types is presented in the *Business BASIC Technical Concepts manual.* For other terminal types, see your system manager. Your terminal type number then becomes part of your account ID. The HELLO program now displays the following:

```
ACCOUNT:
```

Answer with your account name. Account names are five characters long; the first two characters are your account type (such as AA, AB, or OP). See the *Business BASIC System Manager's Guide* for information on valid accounts. The next three characters convey your unique username. You can type all five characters in response to the "ACCOUNT:" query as shown below:

```
ACCOUNT: ABDGC
```

Or you can type just the first two characters in which case HELLO then asks for your username as shown below:

```
ACCOUNT:AB USER: DGC
```

After typing your account name,the HELLO program asks for your password:

```
PASSWORD:
```

Respond by typing your password followed by CR. For reasons of confidentiality, your password is not displayed as you type it. However it is still being sampled for validity by the HELLO program.

Unless your system manager has already specified a directory,the HELLO program now asks for a directory by displaying the following prompt:

```
DIRECTORY:
```

Respond by typing in the name of the directory in which you want to work. For instance, user ABDGC cited in the previous examples does not have a directory already specified. To assign a directory name of USER1, respond by typing the following:

```
DIRECTORY: USER1
```

The log-on procedure has now advanced to the point where USER1 has invoked Business BASIC and is in keyboard mode. This is signified by a banner message followed by the appearance of the asterisk (*) keyboard mode prompt displayed at the left of the screen. That is, the last displayed line of the sign-on message displays the following:

```
*
```

This prompt signifies that you can now begin using the Business BASIC language.

Exiting Business BASIC (RDOS, DOS)

Exiting Business BASIC under RDOS/DOS is accomplished by entering BYE at the asterisk prompt.

The system responds by displaying a sign-off message similar to the following:

```
JOB=0 TERM=0 ACCOUNT=AAAAA6 CPU=3.1 I/O=137 CONNECT=0
```

This indicates that you have concluded your Business BASIC session. You may now re-enter Business BASIC at any time by pressing the ESCape key and then repeating the log-on sequence outlined above.

Logging-on (AOS/VS, AOS)

The Business BASIC log-on sequence now described assumes the following:

1. You have been assigned an AOS/VS, AOS user name and profile, created by the System Manager.
2. Your terminal is on.

With these conditions met, entering Business BASIC can be considered a two step process:

- Logging on to the AOS/VS or AOS operating system.
- Entering Business BASIC itself.

Logging on under AOS/VS or AOS is initiated by pressing the NEW LINE key. When you do this, the system responds by displaying the AOS version number, date, and time on the top line. The second line displays the following prompt:

```
USERNAME:
```

Respond by typing in your user name. For example, if you are assigned the username of "KILROY", you respond as follows:

```
USERNAME:KILROY
```

When you press the NEW LINE key following your username, the system displays the following prompt:

```
PASSWORD:
```

Respond by typing in the password assigned to you by the System Manager. (For reasons of confidentiality, the letters of your password are not displayed as you type.) Follow entry of your password by pressing the "NEW LINE" key. If you have correctly entered your username and password, the system responds by displaying a multi-line sign-on message followed by a right-parenthesis prompt.

NOTE:

If you make an error in entering your username/password, the system displays "INVALID USERNAME-PASSWORD PAIR". You must then repeat the log-on sequence.

The right-parenthesis prompt signifies that you now have access to the AOS/VS or AOS Command Line Interpreter (CLI). The next step is to gain entry to Business BASIC itself. The procedure for doing so may vary from system to system depending on the entry technique set up by your system manager. (Consult him to determine the correct entry sequence for your terminal.) One entry technique may require you to invoke the XEQ command with the following arguments:

```
) XEQ BBASIC/C
```

Note that the /C switch is required to access the Business BASIC CLI.

Still other entry procedures may be required if so selected by your system manager. Refer to the Business BASIC System Manager's Guide for additional information.

On successful entry into Business BASIC, the system displays a multi-line sign-on message followed by the asterisk (*) prompt. This signifies that you are in Business BASIC keyboard mode. You can now begin using the Business BASIC language.

Logging-Off (AOS/VS, AOS)

To log off Business BASIC type the following from the Business BASIC "*" prompt:

```
* BYE
```

The system responds by displaying a sign-off message followed by the right-parenthesis prompt. This signifies that you have exited Business BASIC and have returned to the Command Line Interpreter(CLI) level.

If at this time you are ready to log off the system, type the following:

```
) BYE
```

The system now displays the sign-off message indicating that you have completed the log-off procedure. When you wish to re-enter Business BASIC, you must repeat the entire log-on sequence.

Keyboard Mode and Working Storage

Regardless of operating system (AOS/VS, AOS, RDOS, or DOS) the appearance of the asterisk prompt (*) indicates that you are in Business BASIC keyboard mode. From this prompt, programmers can begin entering lines of code as shown in Figure 1-2.

In keyboard mode, you type directly into an assigned buffer area called working storage. (Every system user has one.) You can key in a command to execute immediately or type program statements to create a program. The Business BASIC interpreter checks the syntax of what you type and reports any errors. If there are none, the system adds your statements to the others you have already typed into working storage.

When you type in program statements, you must precede each one with a line number. For example:

```
* 10  A=2
```

is line number 10 of the program currently in working storage (preceded by the asterisk prompt). You enter the line by pressing New Line or Carriage Return (depending upon the operating system). When you enter the line, the system immediately interprets the line you typed and checks for syntax errors. It also adds this statement (if its syntax is correct) to the other statements in working storage, or replaces any statement in working storage that has the same line number. The remainder of this Guide assumes the use of the NEW LINE or Carriage Return key to enter Business BASIC statements and commands. When you RUN the program, the system executes the statement with the lowest line number and then proceeds to the next higher number, unless the statement directs the computer elsewhere (as GOTO and GOSUB statements do).

If you typed line 10 above, then added the next statement as line 20:

```
* 20  PRINT A
```

You now have lines 10 and 20 in your working storage. To see the contents of working storage, use the LIST command:

```
* LIST

0010  LET A=2
0020  PRINT A
```

You can execute the contents of working storage with the RUN command:

```
* RUN
2

*
```

At this point, the variable A has the value of 2. The system not only stores the program in working storage, but also stores the values you've assigned to variables. After RUNning this program you may use the PRINT command to display the value of A:

```
* PRINT A 2

*
```

When a program has completed its execution, it stays in working storage until you execute one of the following:

- A NEW statement/command to clear working storage.
- A CHAIN statement/command to execute another program.
- A LOAD command to bring a new program into working storage.
- A RUN command to execute another program.

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

While your program is in working storage, you can replace a statement by typing a new statement with the same line number. For example,

```
* 10  LET A=4
```

replaces the existing statement number 10 that assigned the value 2 to A. To PRINT A as 4, use the following lines of code:

```
0010  LET A=4
0020  PRINT A

* RUN
4

* PRINT A 4
```

You can also use the keyboard editing commands to modify statements in working storage. You can use the ERASE command to erase a range of statements. You can use the ENTER command to bring statement lines into working storage from a listing file (text file) and merge them with lines you already have in working storage. An ENTERed statement replaces any existing statement with the same line number.

To save the contents of working storage, use the SAVE or LIST command. LIST by itself displays the contents of working storage on the user's console; LIST with a filename in quotes creates a listing file (text file) to hold the program statements of working storage on disk. SAVE saves the contents of working storage in a disk file. To bring a SAVEd program into working storage, use the LOAD command, or execute the program directly using RUN, CHAIN, SWAP or CON. To bring in a LISTed program from a listing file, use the ENTER statement/command.

Writing and Editing Business BASIC Programs

The discussion now centers on procedures for both writing Business BASIC programs and modifying/updating lines of code via editing techniques. Numerous additional examples appear in this section showing:

- Procedures for creating lines of code to form a program.
- Modifying lines of code that you wish to change following error detection.

To initiate the coding sequence at the terminal, first clear your working storage with a NEW command. Now there are no statements with line numbers in working storage, and you can choose any line number for your first statement. For most of our examples, we number our statements by increments of ten providing plenty of room to insert new statements between existing statements. Your line numbers can range from 1 to 9999. You can type any line number and statement out of sequence, as we do in our example, and BASIC will automatically line up the statements in ascending line number order when you enter the LIST command:

```
* NEW
* 10  PRINT "HELLO"
* 5  LET X=2
* 20  LET Y=X*10
* 40  PRINT X,Y
* 30  PRINT "THE VALUES OF X AND Y ARE:"

* LIST
0005  LET X=2
0010  PRINT "HELLO"
0020  LET Y=X*10
0030  PRINT "THE VALUES OF X AND Y ARE:"
0040  PRINT X,Y

* RUN
HELLO
THE VALUES OF X AND Y ARE:
2 20
*
```

This program computes the value of X times 10, where X equals 2. It is now in working storage. To save it, use the SAVE command with a filename:

```
* SAVE "SAMPLE"
```

The file SAMPLE now exists in your directory as a program file; you can RUN, CHAIN, or SWAP to SAMPLE to bring it into working storage and execute it, or simply LOAD SAMPLE into your working storage. RUN, CHAIN, and LOAD clear working storage before bringing in the new program, whereas SWAP saves the old contents of working storage, executes the new program, and restores the old contents of working storage.

When you type an incorrect statement line, you can change it using several methods. If you type a character and you want to erase it, use the DEL key. To erase an entire line, use the CTRL-U (AOS/VS or AOS) or CTRL-X (RDOS/DOS) sequence. This tells the system to ignore the line. When you're typing a line, you can use the interrupt key sequence (called IKEY) to tell the system to ignore it. This is the ESCape key unless some other key has been selected by your system manager.

You can modify a line in three different ways: retype it, type its line number to delete it, and use keyboard editing commands.

Retyping the Line

When you retype the line with its line number, the system deletes the old line from working storage. Here's an example:

```
* LIST 10

0010  PRINT "HELLO"

* 10  PRINT "SAMPLE PROGRAM"
* LIST 10

0010  PRINT "SAMPLE PROGRAM"

*
```

Typing the Line Number

When you type the line number and press Carriage Return or NEW LINE, the system deletes the line you specify. This has the same function as the ERASE command. Here's an example:

```
* LIST

0005  LET X=2
0010  PRINT "SAMPLE PROGRAM
0020  LET Y=X*10
0030  PRINT "THE VALUES OF X AND Y ARE:"
0040  PRINT X,Y

* 10

* LIST

0005  LET X=2
0020  LET Y=X*10
0030  PRINT "THE VALUES OF X AND Y ARE:"
0040  PRINT X,Y

*
```

Using Keyboard Editing Commands

You can also use the keyboard editing commands to modify a line. When a line causes an error, the system puts the line in a special edit buffer. Table 2-1 lists a summary of keyboard editing commands.

.(period)	Sends the line in the edit buffer to working storage for interpretation.
.A string	Appends **string** to the line currently in the edit buffer. You must separate **string** from **.A** with a space. Your string can be any series of characters (without quotation marks surrounding them). The **.A** command does not pass the edited line to working storage.
.C/string$_1$/string$_2$ *[/G]*	Changes the first occurrence of **string$_1$** to **string$_2$**. If you include the optional */G* switch, **.C** will change all occurrences of **string$_1$** to **string$_2$** in the edit buffer. The *.C* command also passes the line to working storage for interpretation.
.E/string$_1$/string$_2$ *[/G]*	Changes the first occurrence of **string$_1$** to **string$_2$**. If you include the optional */G* switch, **.E** will change all occurrences of **string$_1$** to **string$_2$** in the edit buffer. The **.E** command does not pass the line to working storage.
.I string	Changes the entire line in the edit buffer to **string**. You must separate **string** from **.I** with a space. Your string can be any series of characters (without quotation marks surrounding them). The **.I** command does not pass the edited line to working storage.
.P	Displays the contents of the edit buffer.

Table 2-1 Summary of Keyboard Editing Commands

When you LIST the contents of working storage, the last line LISTed will remain in the edit buffer. To put a line in this edit buffer, LIST the line by specifying its line number as an argument to LIST:

```
* LIST 20
0020  LET Y=X*10
*
```

Line 20 is now in the edit buffer. To display the contents of the edit buffer, use the .P command:

```
* .P
0020  LET Y=X*10
*
```

Use the .E and .C commands to change characters (called strings) in the edit buffer. Their formats are:

.E/string$_1$/string$_2$ *[/G]*
.C/string$_1$/string$_2$ *[/G]*

The .E command changes the first occurrence of **string$_1$** to **string$_2$**. If you use the optional */G* switch, it changes all occurrences of **string$_1$** to **string$_2$**. This command keeps the edited line in the edit buffer for further editing.

The .C command performs the same operations as the .E command, except that .C immediately passes the edited line back to working storage where BASIC interprets it. However, if the edited line causes an error, the system returns an error message and puts the line back into the edit buffer.

For example, type the following statement:

```
* 100 PRINT "THE LAST AMOUNT WAS";AMOUNT
*
```

Use the LIST command to put the line in working storage:

```
* LIST 100
0100  PRINT "THE LAST AMOUNT WAS";AMOUNT
```

Then use the .E command to change this line:

```
* .E/WAS/IS
0100  PRINT "THE LAST AMOUNT IS";AMOUNT
*
```

Note that the system automatically echoes on your console the edited version of any line that you edit.

Now the line is ready for working storage, so you use the period (.) command to send the contents of the edit buffer to working storage:

```
* .  0100  PRINT "THE LAST AMOUNT IS";AMOUNT
*
```

If you want to change the second occurrence of AMOUNT to AMT, first LIST line 100 to put it in the edit buffer, and then use either .E or .C as follows:

```
* LIST 100

0100  PRINT "THE LAST AMOUNT IS";AMOUNT

* .C/AMOUNT/AMT/G

0100  PRINT "THE LAST AMT IS";AMT

*
```

Ooops! You changed all occurrences of AMOUNT to AMT when you wanted to change only the second occurrence. That's easy to correct. Use .P to display the contents of the edit buffer:

```
*.P
ERROR 73-EDIT BUFFER IS EMPTY
*
```

Since we used .C above, the system passed the line to working storage, emptying the edit buffer. We have to LIST the line again:

```
* LIST 100

0100  PRINT "THE LAST AMT IS";AMT

*.C/AMT/AMOUNT
0100  PRINT "THE LAST AMOUNT IS";AMT
*
```

Now our statement is correct, and .C passed it back to working storage.

In addition to the .P, .C, .E, and period (.) commands, you can use the .A command to append a string to the end of a line, and the .I command to change the entire line. Their formats are:

.A string
.I string

For example, to append the string "+(A-B)" to line 20, use the following sequence:

```
* LIST 20

0020  LET X=Y*10
```

```
* .A +(A-B)
0020  LET X=Y*10+(A-B)

*
0020  LET X=Y*10+(A-B)

*
```

Since .A does not pass the line back to working storage, you must use the period (.) command. If you want to change this line entirely to "LET X=(A-B)*10", use the .I command:

```
* LIST 20

0020  LET X=Y*10+(A-B)

* .I 20 LET X=(A-B)*10

*
20 LET X=(A-B)*10

*
```

Since .I does not pass the line back to working storage, you must use the period (.) command. Note that you must also type the line number of the line to change the entire line. The edit buffer sees line numbers as characters in the line.

NOTE:

Business BASIC has an EDIT utility which can be used to create or edit a text file. To conserve memory, EDIT uses a disk-resident buffer to contain the lines being operated on. For details on EDIT, consult the *Business BASIC Subroutines, Utilities, and BASIC CLI* manual.

You can also use keyboard editing commands on BASIC commands and statements, and you can change a statement to a command by omitting its line number or change a command to a statement by including a line number.

Executing Business BASIC Programs

When you finish typing your program into working storage, you can execute the program by typing

```
* RUN
```

When you issue the RUN command, the system clears all values currently assigned to variables and starts executing the program in order from the lowest to the highest numbered line. The computer 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 hit the interrupt key known as IKEY.

You can also execute your program by invoking the SWAP and CHAIN commands as discussed in the subsequent paragraphs.

If you want to bring a new program into working storage and execute it from the point where it last stopped (i.e., do a CON), use SWAP THEN CON and CHAIN THEN CON. These versions of SWAP and CHAIN, like the RUN command with a line number, do *not* clear values from variables when they continue execution of the new program.

For example, enter the sample program below as follows:

```
* NEW
* 0010  LET X=4
* 0020  PRINT X
* 0030  LET X=100
* 0040  STOP
* 0050  PRINT "X=",X
* 0060  END
*
```

Then RUN it. It will stop at line 40:

```
* RUN
4

STOP AT 0040
*
```

Now SAVE the program as "THIS.BB". Since line 30 assigns a new value to X, if we use CON to continue execution, the program displays the new value of 100 before ending. Instead, we write a new program that SWAPs to THIS.BB:

```
* NEW
* 10  LET X=2
* 20  SWAP "THIS.BB" THEN CON
* 30  PRINT "BACK ALREADY!"
* LIST

0010  LET X=2
0020  SWAP "THIS.BB" THEN CON
0030  PRINT "BACK ALREADY!"

*
```

When we run this new program, it assigns the value 2 to X, and SWAPs to the program THIS.BB. THIS.BB displays the value of X:

```
* RUN

X= 100

BACK ALREADY!

*
```

SWAP THEN CON first stores the working storage program and values, including the value 2 for X, in a swapping file. SWAP THEN CON then executes the program THIS.BB from the point after the previous STOP without clearing its values for variables; therefore, X still equals 100.

You cannot directly pass values for variables between SWAPping programs. You must use the BLOCK READ/WRITE statements to transfer data through the common area. (To transfer single-word values, the STMA 1 statement may be used.)

Interrupting and Debugging Business BASIC Programs

Interrupting programs written in Business BASIC is accomplished by pressing the ESCape key (or any other key which the user has designated as the interrupt key as described in STMA 4,6 and STMA 4,7 in the *Business BASIC Statements, Commands, and Functions* manual).

However, you can execute the program from a line number you specify without clearing all values assigned to variables. This variation is especially useful for debugging BASIC programs. You can change the values of variables and RUN the program starting at a specified line number to see what the program will do with new values.

The CON command is also useful for debugging. If your program stops for any reason (a STOP or END statement, an error, or interrupt), you can give the CON command to continue execution at the next higher line. CON will retain the current values for variables.

The following is a simple program that causes an error:

```
* NEW
0010  LET X=2
0020  LET Y=4
0030  PRINT Z
0040  PRINT "Z TIMES 10 EQUALS:";Z*10
0050  STOP
* RUN
ERROR 17 AT 0030--UNASSIGNED VARIABLE
*
```

Line 30 causes an error because Z does not have a value. To see the line that caused the error, we LIST 30:

```
* LIST 30
0030  PRINT Z
*
```

Now we assign a value to Z and CONtinue execution:

```
* Z=X+Y
* CON
Z TIMES 10 EQUALS: 60
STOP AT 0050
*
```

CON started execution at line 40, the line after the one that caused the error. The program gave us a value for Z times 10, but since we never finished the execution of line 30, we still don't have a value for Z. So we will RUN the program again:

```
* RUN
ERROR 17 AT 0030--UNASSIGNED VARIABLE
*
```

Since RUN clears all values for variables, our keyboard assignment for Z disappeared. Therefore, we have to reassign a value to Z, but this time we'll correct the bug by introducing a new line 25 and RUNning the program from line 25 as follows:

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

STOP AT 0050
*
```

Business BASIC also has two statements that serve as useful tools in interrupting and debugging programs. They are the ON IKEY/ON IKEY THEN, and ON ERR statements.

ON IKEY Statement

To interrupt a program's execution, hit the interrupt key known as IKEY (the ESCape key). Normally, when an interrupt occurs during program execution, the program halts and your terminal returns to keyboard mode.

Optionally, you can trap any interrupts in your program using the ON IKEY statement. An ON IKEY statement takes the following form:

ON IKEY THEN statement

where **statement** is any valid Business BASIC statement except FOR, NEXT, DATA, END, REM, and DEF. It can be a **statement** such as SWAP, CHAIN, ENTER, SAVE, and REPLACE. BASIC will not execute statement unless it encounters an interrupt after it encounters the ON IKEY statement. Then BASIC will execute **statement** rather than halt the program, and will set SYS(26) equal to 1. Your statement handles the interrupt condition. Some users prefer to transfer control to a subroutine; others prefer to use the form ON IKEY THEN NEW or ON IKEY THEN BYE to clear the program from working storage if BASIC interrupts its execution. ON IKEY THEN INT or ON IKEY THEN STOP cancels the previous ON IKEY statement, and returns interrupt handling to Business BASIC.

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. The interrupt sets SYS(26) to 1 so that you can test this value to see if an interrupt occurred. If an interrupt did occur, you can restore the interrupt condition by re-enabling interrupts with an STMA 7,5.

CAUTION:

If you interrupt an input/output statement during its execution, you risk losing data, since the statement could have been partially executed. CON continues execution only at the next statement. Reposition your file pointer and try to use RUN with a line number to execute the statement.

ON ERR Statement

ON ERR traps errors in the same manner as ON IKEY traps interrupts. An ON ERR statement takes the following form:

ON ERR statement

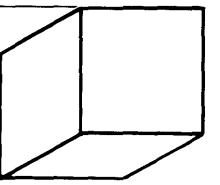
Normally, when an error occurs during program execution, the program halts, the system displays an error message, and your terminal returns to keyboard mode. If an error occurs after BASIC encounters an ON ERR statement, BASIC executes the statement specified in ON ERR. In all cases, whenever an error occurs, SYS(7) or SYS(31) equals the error code of the appropriate error message. You can use SYS(7) with the ERM$ function (on RDOS/DOS) or SYS(31) with the AERM$ function on AOS/VS or AOS to return the error message. ON ERR THEN INT cancels any previous ON ERR statement.

For examples of ON ERR and ON IKEY, see the *Business BASIC Commands, Statements, and Functions* manual.

Storing and Documenting Programs

There are two terms relevant to the discussion of storing and documenting programs: program file and listing file. The program file (a term unique to Business BASIC) is a BASIC program that you SAVEd (using the BASIC CLI SAVE or REPLACE statement/command). A listing file is an ASCII text file containing a program and its comments.

Chapter 3
Numeric, Array, and String Variables



This chapter describes Business BASIC arithmetic and the use of numeric, array, and string variables and string functions.

Variables

There are three types of variables: numeric, array, and string. Each variable represents a value. You can assign a value to a variable or change the value of a variable using any of the following statements: READ, DATA, LET, INPUT, INPUT USING, and TINPUT.

Variable names must begin with a letter. They can have up to five alphanumeric characters following the first letter. In addition, the last character of a variable name can be a special character. The special characters are: $, #, and %. The special character indicates a special variable type. A dollar sign ($) at the end of a variable name indicates that it is a string variable (see "Strings"). The percent sign (%) and the number sign (#) are used to specify the precision of the variable, that is, the number of bytes, when transferring data to and from a file. The concept of precision is discussed in greater detail later in this chapter.

Here are some examples of legal and illegal variable names.

Legal	**Illegal**
A	1
A303	1AB
Z6B8	12345678
WAGE	2WAGE
WAGE2	$WAGE
WAGE$	#WAGE
WAGE%	%WAGE
WAGE#	A$%#

Expressions

A numeric expression is any combination of numbers, numeric variables, array elements, and numeric functions, linked together by arithmetic operators and parentheses. The arithmetic operators are as follows:

Operator	Meaning	Example
+	Unary plus sign	A+(+B)
-	Unary minus sign	A+(-B)
^	Exponentiation	A^B
*	Multiplication	A*B
/	Division	A/B
+	Addition	A+B+C
-	Subtraction	A-B

Business BASIC calculates numeric expressions in the following order:

1. Any expression in parentheses; the innermost expression in nested parentheses.
2. Unary plus and minus.
3. Exponents, left to right in a series of exponents.
4. Multiplication and division, left to right.
5. Addition and subtraction, left to right.

For example:

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

The computer calculates X in the following order:

1. A is negated.
2. C^D is calculated.
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.

Relational Operators

Use relational operators to compare two expressions in an IF decision statement. Relational operators are as follows:

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	A<>B

The following is an example:

```
0010  IF  A^B>=32767 THEN GOTO 0300
0020  IF  ABS(X*Y)<>A THEN GOSUB 0900
```

Numeric Variables

All numbers in Business BASIC must be whole numbers. You cannot assign a fractional number to a variable. You can, however, use formatting techniques (such as PRINT USING and VAL.SL) to print numbers with decimal points.

Precision

"Precision" refers to the number of bytes used to store or transfer a numeric variable or array element. It is important to keep clear the distinction between storage precision and data transfer precision.

Storage Precison

Only double and triple precision are available for storage. Double precision numeric variables store numbers using 32 bits (4 bytes or 2 16-bit words), and can range in value from -2,147,483,648 to +2,147,483,647. Triple precision numeric variables store numbers using 48 bits (6 bytes or 3 16-bit words) and can range in value from -14,073,748,855,328 to +14,073,748,855,327.

Data Transfer Precision

Business BASIC supports three precisions for transferring information to and from a file. They are:

- Single Precision Format
- Double Precision Format
- Triple Precision

Data transfer precision is selectable via the special symbol you append to a variable in a line of Business BASIC code. These symbols and the related precisions they select are:

- Percent sign (%). Appending this character at the end of a variable selects single precision.
- Blank or no character. This is the the default condition that selects double precision. When no character is appended at the end of a variable, Business BASIC selects double precision.
- Pound sign (#). Appending this character at the end of a variable selects triple precision.

Hence, for READ FILE and WRITE FILE, respectively, single precision (%) numeric variables read in and write out 2 bytes each, double precision (regular, no symbol) numeric variables read in and write out 4 bytes each, and triple precision (#) variables read in/write out 6 bytes each. Triple precision (#) variables cannot be used in a double precision system. The type of precision a system uses is determined by the system manager at sysgen time. Instructions for this procedure are presented in the *Business BASIC System Manager's Guide.*

For example, the statement

```
0010  WRITE FILE (0), VAR#
```

transfers six bytes (triple precision).

Arrays

An 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. If it is single precision, each array element holds a 2-byte value; if it is double precision, each array element holds a 4-byte value; if it is triple precision, each holds a 6-byte value. Business BASIC uses arrays like variables -- to represent values. However, a single array can hold as many values as you want. There is no restriction on the number of elements you can place in an array other than restrictions due to available memory.

Arrays can have one dimension, where each element forms a column of the array, or two dimensions, where the elements form rows and columns. Figure 3-1 shows both.

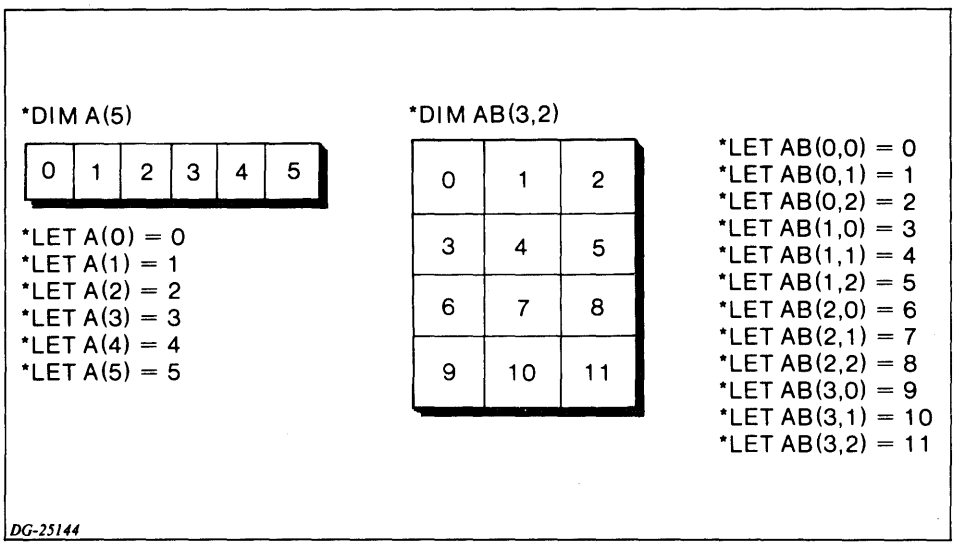


Figure 3-1 *One- and Two-Dimensional Arrays*

Creating Arrays

In Figure 3-1, we use a DIMension command to allocate space for the array. In your program, you would use a DIMension statement. Notice that arrays start at 0. Thus, when you DIM A(5), you get a one-dimensional array that can hold 6 elements as in Figure 3-1. When you DIM AB(3,2), you get a 4 by 3 two-dimensional array that can hold 12 elements. In Business BASIC, you always reference one-dimensional arrays by appending one subscript to the array. For example, A(1) refers to the second element.

You reference two-dimensional arrays by appending two subscripts to the array variable. If you reference a two-dimensional array and use only one subscript, it specifies column 0 of the row indicated by the subscript. For example, AB(1,2) refers to row 1, column 2, but AB(1) refers to row 1, column 0 (which contains the value 3 in Figure 3-1).

If you want to refer to an element in a two-dimensional array, specify the row and the column. For a one-dimensional array, specify the column. For example, in Figure 3-1, if X=A(4), then X=4; if Y=AB(2,1), then Y=7.

You can redimension an array to its original size or smaller. For example, you can redimension the two-dimensional array in Figure 3-1 to DIM AB(2,3) because it will still hold 12 elements.

Redimensioning an array does not change the values in the array; it just changes the location of certain values in the array. If you redimension an array to have fewer elements, you will not free the unused locations; you'll just make it impossible to refer to them.

Example

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

Now, E(10) is nonreferenceable.

It's possible to use an array without declaring it in a DIM statement or command. In this case, the BASIC default procedure is to set aside 11 elements (column positions 0 to 10) for a one-dimensional array and 121 elements (columns 0 to 10, rows 0 to 10) for a two-dimensional array. It's important to remember this because an undeclared one-dimensional array can have only 11 elements, and an undeclared two-dimensional array can have only 121 elements. If you need more space, you *must* dimension the space using DIM. There are no restrictions on the number of elements you can place in a DIMensioned array other than the restrictions due to available memory. Use the LET statement to create such nondimensioned arrays.

For example:

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

creates a default dimension for array C as (10), which means that C contains 11 elements.

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

creates a default dimension for array D as (10,10), which means that D contains 121 elements.

You can also use LET to assign values to array elements of arrays you have created with LET or DIM statements.

For example:

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

A is explicitly dimensioned as (8) by the DIM statement, while AB is implicitly dimensioned as (10,10), the default dimension for two-dimensional arrays.

Another way to assign values to elements of an array is with the READ command.

```
* 0010  DIM ARA(12,12)
* 0020  READ X,Y
* 0030  READ ARA(1,3)
* 0040  PRINT "X=";X
* 0050  PRINT "Y=";Y
* 0060  PRINT "ARA(1,3)=";ARA(1,3)
* 0070  STOP
* 0080  DATA 32,46,1400
* RUN
X=32
Y=46
ARA(1,3)=1400

STOP AT 0080
*
```

Strings

A string is a combination of characters. It may include letters, digits, spaces, special characters, and sometimes binary values. Strings must always be dimensioned, there is no default dimensioning for strings as there is for arrays.

A string literal is a string enclosed in quotation marks. If the string's characters include decimal ASCII code numbers, you must enclose them in angle brackets (<>) as well.

A string literal is another name for a string constant. You assign a string literal to a string variable as you would assign a numeric constant to a numeric variable.

You can also use string literals in PRINT and INPUT statements. INPUT statements use the string literal as a prompt for the input query (see INPUT in the Business BASIC Commands, Statement, and Functions Manual).

PRINT statements simply print the string literal:

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

Here's an example of a string assignment which includes ASCII code numbers:

```
* DIM A$(20)
* LET A$="THIS IS A STRING <7>"
* PRINT A$THIS IS A STRING
*
```

The number "7" in angle brackets is the decimal ASCII character code for the terminal bell. When Business BASIC executes this statement, the string "THIS IS A STRING" appears on the terminal screen, and you hear a bell or beep. Since every character has a decimal ASCII code number (see the ASCII chart in the *Business BASIC Technical Concepts* manual), you can use the decimal code number with angle brackets to put special and nonprinting ASCII characters into a string.

String Variables

String variables represent strings, and their names always have a dollar sign ($) suffix. String variables use 1 byte (8 bits) to hold each character value. Each character has an ASCII character code number that BASIC stores in binary (base 2) in the 8 bits. Use the DIMension statement to allocate the maximum number of bytes (characters) for the string variable before you assign a string value to it.

Unlike arrays, string variables start at byte 1. Therefore,

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

dimensions two string variables: A$ has a maximum of 25 bytes, and B3$ has a maximum of 215 bytes. You can't assign to a string variable more characters (bytes) than its maximum length. If you redimension a string, your 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 as shown below:

```
* 0010  DIM ARRAY(5,6),C(20),STRING$(30)
```

ARRAY is a 6 by 7 element, two-dimensional array. C is a 21-element, one-dimensional array. STRING$ is a 30-character string variable.

```
* 0020  DIM ARRAY(6,5),C(2,6),STRING$(28)
```

This statement redimensions the ARRAY to a 7 by 6, two-dimensional array, redimensions the 21-element, one-dimensional array C to a 3 by 7 (21-element), two-dimensional array, and redimensions the length of STRING$ to 28 characters. For more information on strings, see the *Business BASIC Commands, Statements, and Functions* Manual.

Substrings

You can select portions of a string by subscripting your string variable. If you specify one subscript, you are referring to that portion of the string from the byte position you specify to the last byte position. If you specify two subscripts, the first is the starting byte position, and the second the ending byte position. The subscript "0" refers to the byte position that follows the last byte (you must use this to add a substring to the end of a string).

For example:

A$	refers to the entire string.
A$(2)	refers to the 2nd through the last characters in the string.
A$(I)	refers to character position I through the last character.
A$(3,7)	refers to the 3rd through 7th characters in the string.
A$(I,J)	refers to positions I through J, where I<=J.
A$(2,2)	refers to only the 2nd character in the string.
A$(0)	refers to the end of the string's current length, equivalent to A$(LEN(A$)+1).

You can use subscripted variables to refer to substrings in LET, PRINT, INPUT, READ, and other statements.

Assignment statements like LET and READ allow inclusion of the following in their string expressions:

- String variables
- Subscripted string variables
- String functions
- The concatenation operator (comma)

The expressions of the PRINT and INPUT statements can include all of the above except string functions which are described in the next paragraph.

Some additional examples of string assignments and types of string expressions appear below:

LET A$=B$	replaces the contents of A$ with the contents of B$.
LET A$=""	replaces the contents of A$ with a null string.
LET A$=A$,B$	appends the contents of B$ to the current contents of A$.
LET A$(0)=B$	appends contents of B$ to the current contents of A$.
LET A$=B$,A$	produces unpredictable results, since A$ becomes B$ before it can be appended to B$.
LET A$=4+5	replaces the contents of A$ with the digit 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.

Note that strings must be filled beginning from position 1. This means that, for example, you could not assign A$(3,5)="ABC" if A$(1,2) had not already been filled.

String Functions

A string function is a built-in Business BASIC function that evaluates to a string value.

Here are the string functions in Business BASIC:

AERM$	returns system and Business BASIC error messages (SYS(31))
CHR$	puts the binary value of a number into a string.
CRM$	crams every 3 bytes of a string into 2 bytes.
ERM$	returns RDOS-compatible error messages(SYS(7))
FILL$	fills a string or substring with a value.
TRUN$	truncates a string.
UCM$	uncrams a crammed string.

For more information on string functions, refer to the discussions of the specific functions in the *Business BASIC Commands, Statements, and Functions* Manual.

Using String Variables to Transfer Data

File access statements (READ FILE, WRITE 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 variables apply to string variables as well as to numeric variables and arrays.

The size of the variable(s) you supply in the argument(s) determines the number of bytes transferred. String variables transfer their maximum length (the length you specified when you DIMensioned them) whether or not they are filled with actual data (empty string bytes transfer as null bytes). Substrings transfer the number of bytes specified in their subscripts; this results in a transfer of one byte per character. Entire strings also transfer one byte per character.

Numeric/String Conversions

There are several string functions in Business BASIC that accomplish numeric to string conversions and vice versa. We describe each one below.

CHR$

CHR$ puts the binary value of a number into a string. Recall the previous example of the CHR$ function:

```
* LET A$=CHR$(12,4)
```

A$ is the string variable that receives the 4 bytes containing the binary value of the number 12. Remember, a string holds the binary value of an ASCII character in a byte. Thus, a string of digits would be a string of ASCII values. When you use the CHR$ function, you specify the number of bytes you are assigning. (In this case the number is 4.) You fill those bytes with the actual binary value of a number (in this case 12), *not* with ASCII values. You then extract that number from the string using the ASC function.

ASC

The ASC function extracts a number from a string. It takes the form:

*** var=ASC(strvar)**

where:

var is the variable to receive the numeric value of strvar.

strvar is a string variable or substring.

In all cases a straight byte transfer occurs. Your var receives the binary value of whatever is in strvar. If strvar contains the letter "A" only, you receive the value 65 in var, because 65 is the decimal ASCII code (the binary value) of "A".

Use the ASC function wherever numeric expressions are allowed.

An example of how the ASC function might be used in a routine appears below:

```
0010  DIM A$(12)
0020  INPUT B
0030  LET A$(1,4)=CHR$(B,4)
.
.
.
0100  LET B=ASC(A$(1,4))
.
.
.
```

In the above example, line 30 makes a four byte string of B, and line 100 extracts B from A$.

For additional examples using the ASC function, refer to the Business BASIC Statements, Commands and Functions Manual.

VAL

The VAL function has the following format:

VAL { **strvar** / **strlit** } **,er**

where:

strvar or **strlit** is a string variable in quotes that contains a string of digits. The special characters plus (+), minus (-), and decimal point (.) are allowed.

er is a numeric variable that can receive a -1 if an error occurred in a conversion, or it's the location of the decimal point.

VAL converts a string of digits to a number (binary value). If the string contains a minus (-) sign, the number returned is negative. If the string contains a decimal (.) point, Business BASIC determines the location by starting from the right. VAL stops converting when it encounters nonnumeric characters; for example, if the string literal is "3.2X", the number returned is a 32 and er equals -1. The VAL function is similar to the VAL.SL subroutine.

For example:

```
0100  LET OUT=VAL(IN$,M)
```

Use the VAL function wherever BASIC allows numeric functions. You can assign the result of VAL to a numeric variable, or PRINT the result.

Refer to the *Business BASIC Commands, Statement, and Functions* manual for additional examples of the VAL function.

VAL.SL

Similar to the VAL string function, VAL.SL is a subroutine that converts a string to a number. As an input variable, it requires X$, a string of digits and special characters that you want to convert to a number.

Its output consists of: X, a signed numeric value of X$; Y%, an error termination code (0 if the conversion ended with a comma, space, or end of string; otherwise, the ASCII value of the terminating character); and Z%, an implied decimal point location.

SGN

SGN determines the sign (positive or negative) of an expression. The format is quite simple:

SGN (expr)

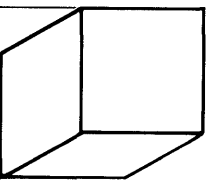
where:

expr is the numeric expression or variable for the number whose sign you want to know.

The following is an example.

```
* 0010  INPUT X
* 0020  IF SGN(X)=-1 THEN PRINT "NEGATIVE"
* 0030  IF SGN(X)=1 THEN PRINT "POSITIVE"
* 0040  IF SGN(X)=0 THEN PRINT "EQUALS ZERO"
.
.
.
```

Chapter 4
Subroutines and Utilities



This chapter describes the complete range of subroutines and utilities within the Business BASIC software package. When building a supply of BASIC programs, Business BASIC prewritten subroutines and utilities should be very useful.

In Business BASIC, you can create your own subroutines and utilities or use those we supply. The supplied subroutines and utilities can also be modified. Modular programming has one distinct advantage: program components can be more easily debugged.

Subroutines

Subroutines within Business BASIC reside in the library directory ($LIB or $LIB3). The subroutines can be accessed from the library (via ENTER statements) and inserted into business application programs to perform specialized functions.

Subroutines can originate from any of three sources:

- Business BASIC. These are the subroutines which are supplied as part of the Business BASIC software package.
- Your own subroutines. These are the subroutines that you create for your own specialized processing requirements and insert in the library for later access.
- Assembly language subroutines. These are routines that have been coded in assembly language rather than BASIC. Provisions are made for accessing such subroutines so that they execute as part of your Business BASIC application.

Business BASIC Subroutines

Business BASIC features a number of pre-written subroutines that can be used within application programs via ENTER statements. All subroutines are contained in the $LIB library (or $LIB3 wnen using triple precision Business BASIC). Their filenames have a .SL extension to distinguish them from utility programs and other files.

An alphabetical listing of the supplied Business BASIC subroutines along with the functions they perform is given below:

DELREC.SL	Deletes a record in a linked record file.
DIV.SL	Calculates extended percentage.
FINDFILE.SL	Finds a subfile and builds the file characteristics array.
FORM.SL	Handles formatted input/output of screen fields.
GETCM.SL	Reads the common area so that you can write your own BASIC CLI commands.
GETREC.SL	Gets an available record in a linked-record file.
INITINDEX.SL	Initializes an index file.
MUL.SL	Calculates the percent of a number.
POSFL.SL	Positions to a record in a data file.

PROTFORM.SL	Protects/unprotects a screen field.
UNFORM.SL	Handles unformatted input/output of screen fields.
VAL.SL	Converts a string to a number.

For detailed descriptions of the above subroutines, refer to the *Business BASIC Subroutines, Utilities and BASIC CLI* manual.

To use a prewritten subroutine, it must be merged with a program that is in working storage.

To access a subroutine within a program, use the GOSUB statement. Each subroutine has RETURN statements that return control to the statement immediately following the GOSUB statement in the main program.

We've created a program in working storage that uses the DIV.SL subroutine to calculate what percentage X is of Y. Here's the program:

```
0010  LET X=25
0020  LET Y=101
0030  GOSUB 7550
0040  PRINT USING "D5.2'%'",PCNT
0050  END
```

Line 30 transfers control to the subroutine at line 7550, the entry point for the DIV.SL subroutine. After it runs, DIV.SL returns control to line 40, where the program displays the value of PCNT, the variable returned by DIV.SL.

NOTE:

DIV.SL must be supplied with two variables: X and Y, and these variable names must be used. Each subroutine requires definite input variables, like X and Y, and most subroutines return definite output variables, like PCNT.

Now we ENTER the DIV.SL subroutine, merge it with our program, and execute the entire program:

```
* ENTER "DIV.SL"
* LIST
0010  LET X=25
0020  LET Y=101
0030  GOSUB 7550
0040  PRINT USING "D5.2'%'",PCNT
0050  END
7550  REM/DIV.SL
7552  LET PCNT=0
7555  IF Y=0 THEN RETURN
7560  LET Z1=X
7565  FOR I%=0 TO 4
7575  LET PCNT=PCNT*10+Z1/Y
7580  LET Z1=MOD(Z1,Y)*10*SGN(Z1)
7585  NEXT I%
7590  RETURN
7599  REM * END OF DIV.SL

* RUN
24.75%

*
```

Two common errors occur when subroutines are used : ERROR 13 -- LINE NUMBER and ERROR 19 -- RETURN - NO GOSUB. If a GOSUB statement is used when the subroutine has not been ENTERed, ERROR 13 occurs. The GOSUB refers to a line number that doesn't exist in working storage. This can be prevented by saving the program with its subroutines. ERROR 19 occurs whenever BASIC executes a subroutine that was not called by GOSUB. In our example above, we put an END statement on line 50 so that the system would continue and execute the subroutine.

Hard-to-detect errors occur when proper values are not supplied for variables or proper variable names are not supplied for the subroutine. Each subroutine expects certain input variables and outputs certain variables.

Since most subroutines have more than one entry point, the proper entry point must be supplied. Figure 4-1 contains a sample data entry program, which illustrates the proper use of prewritten subroutines. It is assumed that access to the subroutines in the library directory is permitted. The example also shows how a main program SWAPs to the data entry program.

Main Program

```
.
.
.
0100  PRINT "PICK A PROGRAM BY NUMBER:"
0110  PRINT "DATA ENTRY IS 1"
0120  PRINT "DATA RETRIEVAL IS 2"
.
.
.
0200  INPUT "WHICH ONE?",A
0210  IF A=1 THEN SWAP "DATAENTRY"
.
.
.
```

DATAENTRY Program

```
0010  ENTER "GETREC.SL"          :Sub to get available record from file.
0020  ENTER "POSFL.SL"           :Sub to position to record or byte in file.
0030  ENTER "UNFORM.SL"          :Sub to display screen, handle I/O.
0040  ENTER "FORM.SL"            :Sub to handle formatted screen I/O.
0050  ENTER "VAL.SL"             :Sub to verify data and catch errors.
0060  ENTER "FINDFILE.SL"        :and build file characteristics array
.                                :(C1 array). Dimension variables and
.                                :C1 array.
0100  LET X$="DATA101"           :Name of data file is DATA101.
0110  LET F%=0                   :DATA101 info will be in row 0 of C1 array.
0120  GOSUB 7800                 :Go to FINDFILE.SL and build C1 array.
                                 :FINDFILE.SL expects X$ and F% as input and
.                                :and returns full C1 array. Open data and
.                                :index files, dimension key and record
.                                :strings.


0200  OPEN FILE (15),"SCREEN.S6"  :Open screen file on channel 15 where
                                 :UNFORM.SL and FORM.SL can find it.
```

```
0210  GOSUB 9150                  :Go to UNFORM.SL to set keyboard.
0220  LET SCRN=0                  :Assign input var SCRN to screen 0.
0230  GOSUB 9700                  :Go to UNFORM.SL to display screen 0.
.                                 :Output blank fields to screen fields.
.                                 :Assign to input var F the value of the
.                                 :first field:
0300  LET F=501                   :row 5, column 1 of screen.
0310  GOSUB 9650                  :Go to UNFORM.SL to position cursor to field
                                  :defined by F
0320  GOSUB 9300                  :Go to FORM.SL to read formatted input from
                                  :field in F. FORM.SL returns X$ with data
                                  :(X$ must already be DIMmed).
0330  GOSUB 7600                  :Go to VAL.SL to convert X$ to a number X.
0340  IF E=0 THEN GOTO 0400       :If no error, go to 400.
0350  LET E=128                   :If error, make it error 128.
0360  GOSUB 9850                  :Output error message to screen.
.                                 :Routine to clear error message if key
.                                 :is hit, and return to read field again
.                                 :for correct data entry.
0400  LET F=601                   :F is now row 6, column 1 of screen.
0410  GOSUB 9650                  :Position cursor to field in F.
0420  GOSUB 9300                  :Go to FORM.SL to read formatted input.
                                  :FORM.SL returns X$ with data.
0430  LET REC$(7,36)=X$,FILL$(0)  :Build record string with data received.
.
.
.                                 :Lock record 0 of data file.
0500  LET F%=0                   :DATA101 info in row 0 of C1 array.
0510  GOSUB 8400                  :Go to GETREC.SL to get available record.
.
.                                 :GETREC.SL uses F% and C1 array, and
.                                 :returns R1 as record number.
0520  GOSUB 9610                  :Go to POSFL.SL to position to record R1.
                                  :POSFL.SL returns C% as channel of file.
0530  WRITE FILE (C%),REC$        :Write new record to DATA101.
.
.
.

Subroutines in Library

FINDFILE.SL
FORM.SL
GETREC.SL
POSFL.SL
UNFORM.SL
VAL.SL
.
.
```

Figure 4-1.

Figure 4-1 shows a fragment of a data entry program. You first ENTER the subroutines which are called in the program. Note that UNFORM.SL has many different entry points to perform different tasks. UNFORM.SL and FORM.SL are screen handling subroutines. GETREC.SL, POSFL.SL, and FINDFILE.SL are subroutines that provide linked-available-record access to files and subfiles. All

subroutines are described in detail in the *Business BASIC Subroutines, Utilities, and BASIC CLI* manual.

RDOS/DOS Business BASIC users should have access to the library directory without having to specify the directory's name. In AOS/VS and AOS Business BASIC, the library directory is usually $LIB. Since it can exist anywhere in the system, each user must be given access to it by putting its complete pathname in his or her search list.

Since prewritten subroutines exist as text files, use the BASIC CLI TYPE command to view them, or the BASIC CLI PRINT command to print them on the line printer. In addition, we have two BASIC CLI commands, BLDCOM and PRTCOM, to display or print special comments included with the text files. BLDCOM builds a temporary documentation file that PRTCOM outputs to your terminal, the line printer, or an output file. These commands recognize special symbols in the text file and insure that only comments about the subroutines are output, and not the subroutine itself.

Your Subroutines

Since it is easier to debug components of programs than entire programs, you will probably want to write many subroutines. These subroutines can be SAVEd with the programs that call them, or they can be created as listing files that are ENTERed into programs.

You should follow our naming convention by attaching .SL extensions to your prewritten subroutines (to distinguish the subroutine from a utility program), and these subroutines should be placed in the $LIB library following debugging and successful execution. After this, a user-generated subroutine can be accessed via ENTER statements just like any of the supplied Business BASIC subroutines.

There can be many GOSUBs to the same subroutine, and the RETURN statement may also be used more than once in a subroutine. The latter is useful if program logic requires the subroutine to terminate at different places depending on conditions. The ON GOSUB statement can also be used to transfer to a subroutine, depending on the conditions set by the subroutine.

Business BASIC permits the nesting of subroutines up to a depth of eight. Nesting occurs when one subroutine is called from another subroutine. Upon execution of the first RETURN statement, control passes to the statement immediately following the GOSUB statement last executed. The next RETURN statement passes control to the next to last executed GOSUB statement, and so on. If you nest subroutines to a depth of eight, and then decide to nest more subroutines or disrupt the logic of a nested subroutine, the GOSUB/RETURN stack must be reset by using the STMA 8 statement/command.

If your program contains GOSUB to a REM statement, the remark appears next to the GOSUB statement when the program is LISTed. For example, we type this simple program into working storage:

```
* 10 INPUT X
* 20  GOSUB 100
* 30  PRINT X
* 40  STOP
* 100  REM This is a label
* 110  LET X=(X*(22/7))^2
* 120  RETURN
```

When this program is LISTed, the REM label appears next to the GOSUB statement:

```
* LIST
```

```
0010  INPUT X
0020  GOSUB 0100 : THIS IS A LABEL
0030  PRINT X
0040  STOP
0100  REM THIS IS A LABEL
0110  LET X=(X*(22/7))^2
0120  RETURN
7550  REM \ DIV
7552  LET PCNT=0
7555  IF Y=0 THEN RETURN
7560  LET Z1=X
7565  FOR I%=0 TO 4
7575  LET PCNT=PCNT*10+Z1/Y
7580  LET Z1=MOD(Z1,Y)*10*SGN(Z1)
7585  NEXT I%
7590  RETURN
7599  REM * END DIV.SL 12/27/77
```

This feature is especially helpful when the subroutines supplied as part of the Business BASIC software are used since each subroutine begins with a REM statement. The REM statement can be suppressed by using the STMA 6,6 statement/command, and re-enabled by using the STMA 7,6 statement/command. The colon (:) marks the label as a comment. BASIC stores all comments following colons only when the program is LISTed to a listing file or the listing file is created as a text file using EDIT or another system editor. Business BASIC does not store the colon comments when you SAVE the program. We mention colon comments near the end of this chapter.

Assembly Language Subroutines

One advantage of Business BASIC is its ability to call assembly language subroutines, thus enabling it to interact more directly with the operating system. In Business BASIC, assembly language subroutines can be called provided the system manager included them in the file USERSUBS.RB when he generated the system. The USERSUBS.OB (AOS/VS, AOS) or the RDOS/DOS macroassemblers generate executable machine language code.

Assembly language subroutines are typically used to perform tasks that Business BASIC can't do. Usually such tasks are *dangerous,* so be careful! Improper use of assembly language subroutines, system calls, or task calls can crash the system.

If you don't know how to set up assembly language subroutines, don't try it now. Only assembly language programmers who have learned RDOS/DOS or AOS assembly language should use this interface.

All assembly language subroutines must be submitted to the system manager at system load time. In RDOS/DOS, the subroutines should be in the file USERSUBS.RB. In AOS/VS and AOS systems, the subroutines should be in the file USERSUBS.OB (or *filename.OB,* where *filename* is the name of the subroutine file). The system inputs this file to the relocatable loader when it creates the BASIC executable file. You must include a subroutine table that has the entry point SBRTB.

The subroutine table is a list of all assembly language subroutines available to a BASIC program. For each assembly language subroutine, the interface requires a four-word list containing the subroutine number, the subroutine entry point, the number of arguments, and the argument control word. We describe the subroutine table with the UCALL statement/command in the *Business BASIC Statements, Commands, and Functions* manual.

Utilities

Business BASIC utilities are programs written in Business BASIC that perform specialized tasks. An alphabetical listing of the Business BASIC utilities along with the functions they perform is presented below:

ATTACH	Attaches your terminal to a detached job (RDOS/DOS only).
CLI	Executes the Business BASIC Command Line Interpreter (CLI).
DBGEN	Generates a database structure with corresponding screen formats.
DOC	Prints a series of text files.
DOCTOC	Prints a table of contents for DOC files.
EDIT	Creates and/or edits a text file.
FILES	Displays names of files in the current directory
FILESORT	Sorts a data file on multiple keys.
FM	File maintenance package.
FMLOG	Prints an FM log file.
FMPRINT	Prints an FM data file.
FMTABPRINT	Prints an FM table file.
IBUILD	Builds an index file from a sorted tag or data file.
ICOMPRESS	Repacks index files to eliminate spaces created by deleted keys.
INDEXBLD	Builds an index file.
INDEXCALC	Calculates index information.
INDEXPRT	Prints an index.
INITFILE	Initializes an index or data file.
LIBRARY	Displays names of files in the library directory.
LOCKS	Displays your current locks (RDOS/DOS only).
LSPEED	Sets your ALM line speed (RDOS/DOS only).
MOVETABREC	Copies FM table file records.
OPEN	Opens physical files and/or subfiles.
OPT	Optimizes the size and execution time of a Business BASIC SAVEd program.
PORTS	Displays jobs on the system.
QFILESORT	Fast-sorts a data file on one key.
RELINK	Recreates a deleted record chain for a data file.
RNAM	Renames program variables.
SCHANS	Displays the system channel assignments (RDOS/DOS only).
SM	Defines screen files (Screen Maintenance package).
TABBUILD	Quickly defines arrays for field descriptor records.
TBUILD	Builds a tag file.
TERM	Changes certain terminal key functions (RDOS/DOS only).
UCHANS	Displays your channel assignments.
ULSPEED	Sets your ULM line speed (RDOS/DOS only).
VAR	Lists the variables in a program.
XBUILD	Builds an index from a data file.

For detailed descriptions of the above utilities, refer to the Business BASIC CLI, Utilities, and Subroutines manual.

There are two types of utilities in Business BASIC. The first type of utility can only be SWAPped to. It is not a program and, therefore, cannot stand by itself.

Information is passed to it, and it returns with results. The second type is one which can be RUN, CHAINed to, SWAPped to, and executed from the BASIC CLI. It is a BASIC program that stands by itself.

Utilities You Only SWAP To

Following is a list of utilities you can only SWAP to:

FILESORT	Sorts a data file on multiple keys.
IBUILD	Builds an index file from a sorted tag or data file.
OPEN	Opens physical files and/or subfiles.
QFILESORT	Fast-sorts a data file on one key.
TBUILD	Builds a tag file.
XBUILD	Builds an index from a data file.

These utilities were designed so that they can be SWAPped to from your program. These utilities cannot be RUN from keyboard mode or executed from the BASIC CLI. You can CHAIN to one from your program, but information from the common area must be retrieved using keyboard mode commands. We designed them for SWAPping to so that your program can use the information they return. These utilities do not work properly unless they can read the common area with a BLOCK READ statement and interpret its data.

The common area is a 512-byte memory location available to each user. Each job has only one common area. If you create other jobs, they have their own common areas. Data cannot be sent into another job's common area. If you want a program to pick up the data you left in your common area, that program must be SWAPped to, CHAINed to, or RUN in your job. The program picks up the data using the BLOCK READ statement. Use the BLOCK WRITE statement/command to put data into the common area and the BLOCK READ statement/command to retrieve that data.

To use the utilities listed above, a string filled with pertinent information must be sent into the common area. This string *must* be DIMensioned to 512 bytes or more, not less. Since the BLOCK WRITE and the BLOCK READ each transfer one 512-byte block, a string whose maximum length is less than 512 bytes cannot be sent, and data in the common area cannot be retrieved with a string that is less than 512 bytes.

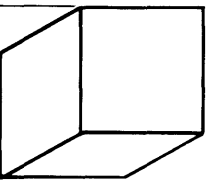
In order to pass an array through the common area to another program, the array must also be greater than or equal to 512 bytes. With a triple precision system, each array element holds 6 bytes; therefore, the array passed to the common area must have at least 86 elements, since 85 elements use only 510 bytes. When this information is retrieved from the common area, an 86-element array must be used, even though you will have only 85 elements (510 bytes) of information. Since the common area holds 512 bytes, the last 2 bytes will contain meaningless information.

In a double precision system, each array element holds 4 bytes; therefore, the array passed to the common area must have at least 128 elements. The array used to retrieve this information must also have at least 128 elements (512 bytes).

Some utilities, like OPEN, require literal filenames and digits. Others, like FILESORT, require binary values in the string. The CHR$ function must be used in order to put a binary value in a string.

Whenever a string is passed to a utility via the common area, the string must be built in accordance with the instructions given with that utility. Whenever information that was passed to the common area from a utility is retrieved the ASC function must be used to extract the binary values from that string.

Chapter 5
Business BASIC File Overview



Two file concepts should be discussed as an introduction to this chapter:

- File Organization
- File Access Methods

File organization refers to the method that is used to store a file on disk. Three basic types of internal file organization exist for storing data files in RDOS/DOS Business BASIC: sequential (RDOS only), random (RDOS/DOS), and contiguous (RDOS/DOS). In addition, AOS/VS and AOS systems have some specific storage characteristics.

Types of access common in Business BASIC include getting information from a file and putting information into memory, (reading or inputting), putting information from memory into a file (writing or outputting); determining the position (number) of a record, determining the location of the file pointer and moving the file pointer. Three modes of access are discussed in this chapter:

- sequential
- random
- indexed sequential

Each of these types of internal file organization and access methods are discussed briefly in this chapter and in detail in the *Business BASIC Technical Concepts* manual.

File Organization

SEQUENTIAL FILES are used only on RDOS/DOS versions of Business BASIC. Sequential files are designed for beginning-to-end input/output access. Business BASIC stores information in sequentially organized files in groups of disk blocks. The last word of each 512 byte block stores a pointer to the next block in the file. When you build a sequential file, the system simply appropriates the next available disk block when it needs additional storage, and constructs a pointer to that block.

RANDOM FILES provide you with the best combination of flexible structure and easily accessible data. They allow random reading and writing throughout the file. Random files do not require that you read all intervening records when positioning to a specific record in the file. The operating system maintains a small sequential file which contains pointers to data blocks in the random file. The operating system needs only to read this small file in order to find the location of the data block containing the desired record. It then accesses this block directly, resulting in faster access of records. A random file allocates space as it needs it and allows you to append records at will.

CONTIGUOUS FILES have a rigid structure, but they provide the fastest access to data. Contiguous files are composed of a fixed number of disk blocks that are

physically adjacent or contiguous on the disk. The operating system can easily calculate the location of a record and access it directly. These files cannot be expanded or reduced in size, since by definition they occupy a fixed series of disk blocks. The disadvantage of using contiguous files is that you must allocate all the disk space that the file will require when you create the file.

AOS FILES. Every AOS/VS and AOS disk file is built from one or more 256-word (512-byte) disk blocks. The system ties together disk blocks within files by employing a hierarchical index. The basic unit of storage in this file organization is a file element consisting of one or more contiguous blocks (blocks with sequential physical addresses). The file element size is specified when a file is created. The system allocates file space in multiples of file elements. Thus, if a file with a file element size of 8 grows, it will grow in units of 8 blocks.

Consider a file created with an element size of two. Initially, this file consists of two contiguous blocks. If file storage requirements exceed the two blocks, an index block is allocated which contains the addresses of each of the file elements; the index element provides no data storage of itself, and each index element is one block (not one file element size) in length.

As data storage requirements continue to grow, more file elements will be allocated and added to the list maintained in the index. If all space in the index becomes exhausted, another level of indexing is added.

In general, you will never be aware of index elements; the system stores and retrieves information so that you need not be concerned with the actual linkage between disk blocks. However, you should be aware of file organization, since it determines the tradeoffs in designing file structures. For example, large file element sizes could be used for creating data storage in contiguous disk areas that do not need indexes; these could be accessed quickly. Smaller file element sizes require more disk accesses, but permit more efficient utilization of disk storage.

Creating Data Files

There are two simple techniques for creating data files with Business BASIC. They are:

- Business BASIC CLI commands: CRAND, CCONT and CREATE. These CLI commands allow you to specify file types as you create files.
- OPEN FILE command. When this command is embedded in your application program along with proper arguments and filename, a file is created in your directory. The filename in your directory is the same as that you specified and you can now write data (WRITE FILE) to it or read data (READ FILE) from it.

  Note that only certain modes of OPEN FILE create a file, while others require that the file already exist. The organization of the file created by open depends on the mode of access.

The Business BASIC CLI commands which are used to create data files are discussed below. The OPEN FILE command is discussed later in this chapter.

Creating Data Files with BASIC CLI Commands

Among the various methods of creating files with Business BASIC are three commands invocable under the BASIC Command Line Interpreter (CLI). That is, you must call in the Command Line Interpreter (or preface the command with "!") prior to invoking any of these three commands. The three file creating commands are:

- CRAND Creates a random file.
- CCONT Creates a contiguous file.

- CREATE Creates a sequential file(RDOS) or a random file (AOS or AOS/VS). Under AOS and AOS/VS, CRAND and CREATE perform equivalent functions.

Depending on the operating system, each of these BASIC CLI commands produces a certain type of file. A summary of the various types of files produced appears in Figure 5-1.

	AOS/VS	AOS	RDOS	DOS
CRAND	Random File	Random File	Random File	Random File
CCONT	Contiguous File	Contiguous File	Contiguous File	Contiguous File
CREATE	Random File	Random File	Sequential File	

DG-25145

Figure 5-1 *Summary of File Types*

File Access

Types of access common in Business BASIC include getting information from a file and putting information into memory (reading or inputting), putting information from memory into a file (writing or outputting); determining the position (number) of a record, determining the location of the file pointer, and moving the file pointer.

This section describes input/output procedures to data files with sequential, contiguous, and random organization. Access commands for files which have sequential, contiguous, and random organization are listed in Table 5-1.

Typical business applications require quick access to records with a key (indexed sequential files). In addition, in order to provide efficient use of deleted record space, Business BASIC file design allows users to employ the use of a linked available record format. For access to linked files and indexed files, the commands listed in Table 5-1 are supplemented by Business BASIC commands as well as Data General-supplied utilities and the INITINDEX.SL subroutine which are listed as follows:

KFIND
KNEXT
KADD
KDEL
INDEXCALC
IBUILD
TBUILD
XBUILD
INDEXBLD
INITFILE
INITINDEX.SL

These commands, programs, and the INITINDEX.SL subroutine are discussed in the *Business BASIC Technical Concepts* manual. In addition, all of the "K" commands are discussed in detail in the *Business BASIC Statements, Commands, and Functions* manual. All of the utilities and the INITINDEX.SL subroutine listed above are discussed in detail in the *Business BASIC Subroutines, Utilities, and BASIC CLI* manual.

Sequential and Random Access

Files can be read and written in either random mode or sequential mode. In random mode, the POSITION FILE command may be used to move the file pointer, and any record may thus be accessed immediately after any other record.

In sequential mode, only the next record may be accessed; the POSITION FILE command does not work. Since random mode allows for faster access of files, it is usually used for files which will require constant updating of records contained within the file. For example, employee personnel records. Sequential mode is used for files which will not require constant updating to records within the file, but rather the addition of new records to the file. For example, a list of company products. Modes are determined when the file is opened.

To read to or write from a data file with sequential, random, or contiguous organization, you must open it in either random or sequential mode for access on a channel (see the explanation of a channel later in this chapter). This is done by using the OPEN FILE command described above. To open subfiles, linked files, or ISAM files you can use either the OPEN FILE command or the OPEN utility.

KEYWORD	PURPOSE
BLOCK READ	Input data from the common area (512 bytes)
BLOCK READ FILE	Input data from a file, in multiples of 512 bytes
BLOCK WRITE	Output data to the common area (512 bytes)
BLOCK WRITE FILE	Output data to a file in multiples of 512 bytes
CLOSE FILE	Close a specific file
CLOSE	Close all opened files
DATA	Specify values for variables in a READ statement
DELAY	Delay program execution
EOF	Check for end of file
GPOS	Determine the current file pointer position
INPUT	Input ASCII data from the terminal
INPUT USING	Input ASCII data from the terminal, allowing an error and a terminator trap
INPUT FILE	Input ASCII data from a file
INPUT FILE USING	Input ASCII data from a file allowing an error and terminator trap
OPEN FILE	Opens a file
PAGE	Set number of characters per output page line
POSITION FILE	Position the file pointer
PRINT	Output ASCII data to the terminal
PRINT USING	Output ASCII data according to a format to the terminal
PRINT FILE	Output ASCII data to a file, terminal, or device
PRINT FILE USING	Output ASCII data according to a format to a file, terminal, or device
READ	Read values from a DATA statement
READ FILE	Input binary data from a file
RESTORE	Reset DATA list pointer
TAB	Set number of characters per print zone
TINPUT	Input ASCII data from the terminal within a fixed amount of time
TINPUT USING	Input ASCII data from the terminal within a fixed amount of time, allowing an error and a terminator trap
WRITE FILE	Output binary data to a file

Table 5-1 Commands Used for Input/Output

The following discussion explains the OPEN utility as well as the concept of a channel and access modes. Later in this chapter more specific explanations are given of I/O to simple sequential and random files.

Types of OPEN

A Business BASIC program can open a data file in exclusive or shared mode. The mode is specified with the OPEN statement. In RDOS/DOS systems other users can READ exclusively opened files but cannot WRITE to them. In AOS/VS and AOS systems, other users cannot READ or WRITE to exclusively opened files. A file opened in shared mode may be used concurrently by any or all of the current users in the system.

Channels

In Business BASIC, channel is the term used to describe the line of communication from your working storage area in memory to a data file or device. Each file you read from or write to in a Business BASIC program must be assigned to a unique channel number. When the file is opened for access, a channel number is associated with that filename. The file is subsequently referred to by its channel number, rather than by its name.

Each user has sixteen channels available for file input/output, in the range of 0 through 15. In addition, channel number 16 refers to the terminal. Any file assigned to channel 16 will read data from the terminal, and will write data to the terminal.

Using the OPEN FILE Statement

In Business BASIC, "OPEN FILE" can be used as either a command or a statement. When a file is opened for access, the following things must be specified:

1. The file to be opened may be represented by either the actual filename, in quotes, or by a string variable which contains the actual filename.
2. The file must be associated with a particular channel number, from 0 to 15. Channel 16 is used to specify that input or output of data should be via the terminal. All subsequent references to that file will be by the channel number rather than by the filename.

   Any channel number which you have not already associated with another filename may be used. It is customary to use the next sequentially available channel number, although you are not restricted to doing this. The UCHANS utility will tell you which channels are currently in use.

   If you specify a channel number which is already associated with another file, an error condition will occur.
3. The mode of open will specify certain conditions about the file. These conditions are:

   - The type of open--exclusive or shared
   - The type of access--random or sequential
   - The type of I/O--input, output, or both
   - The rules on the file's existence--should the file be deleted if it already exists, create the file if it does not exist, or the file must exist.

   By choosing the correct mode, you can manipulate the file characteristics. Table 5-2 summarizes the 8 possible file access modes.

Mode	Type of Open	Type of I/O	Type of Access	Rule
0	Exclusive	Input/Output	Random	Create if file doesn't exist.
1	Exclusive	Output only	Sequential	Delete file, then create file.
2	Exclusive	Output only	Sequential	Append to existing file, or create file.
3	Shared	Input only	Sequential	File must exist.
4	Shared	Input only	Random	File must exist.
5	Shared	Input/output	Random	File must exist.
6	Exclusive	Input/output	Random	File must exist.
7	Exclusive use of mag or cassette tape	Input/output	MTDIO access	Tape must exist, tape file can be created, deleted, or appended to.

Table 5-2 File Access Modes

Closing a File

After a file has been opened for access, and after processing of that file (reading and writing records) has been completed, the file must be closed. Closing a file disassociates that filename from the channel to which it is assigned. You should close a file when you are finished using it. You should also close a file if you want to reopen it in a different mode.

"CLOSE" may be used as either a statement or a command, and has two forms. The first form simply contains the word CLOSE, and will close all files which are currently open to the user. The second form is "CLOSE FILE" followed by a channel number. It will close only the particular file which is associated with that channel number.

Once a file has been closed, then the channel number associated with that file is free, and can be used again to open another file. Also, the file itself can be reopened on a different channel and in a different mode.

Most programs that access files close the files when they are finished using them, or at the end of the program. If you forget to close a file, and try to open it, an error condition (stating the file is already open) will result. If this occurs you can issue a "CLOSE" or a "CLOSE FILE" command while in keyboard mode, and then continue with your program. In order to avoid this error, it is good practice to include a "CLOSE" statement at the beginning of your program, before any OPEN FILE statements are issued.

End of File

EOF (end of file) is a Business BASIC function which is used to detect whether the last READ FILE statement executed has read in the last data record, and the end of file has been reached. For more details on the EOF function see the Business BASIC Technical Concepts manual.

File I/O

To write to a sequential, random, or contiguous file, put the following steps in your program.

1. Dimension the string variable(s) and numeric arrays to be written.
2. Assign values to the variables to be written.
3. Open the file.
4. Write the variable or array to the file using WRITE FILE or PRINT FILE. The number of bytes written is determined as follows:
   - For numeric variables, by the variable's precision
   - For string variables, by the dimension of the variable. If PRINT FILE is used, the amount written will be less than the dimensioned length of the variable if the string contains either a <12> or a <13>, which represent NEWLINE and CR, respectively.
5. Use POSITION FILE to move the file pointer.

   If a file has random or contiguous organization, you may access individual records directly by their relative positions within the file. This is done by positioning the file pointer to the beginning of the record, or portion of a record, to be accessed. If records are fixed in length, the algorithm:

Record number * record length

is used to position the file pointer. The file pointer moves during a record access, and at the conclusion of the operation is positioned to the byte immediately following the last byte which was read or written.

It is also possible to check the current position of the file pointer using the GPOS (get position) function. This is particularly useful if a file contains variable length records.

6. Repeat steps 4 and 5.
7. Close the file using the CLOSE command.

   Steps 5 and 6 can be used only if the file has been opened for random access.

To read from a file, include the following steps in your program:

1. Dimension the string variables and arrays that will receive the information.
2. Open the file. The file pointer is at the first byte in the file.
3. Get the information from the file using either READ FILE or INPUT FILE. The number of bytes transferred is determined in step 1 above for strings and by precision for arrays.
4. Use the EOF function (described earlier in this section) to check if the end of the file has been read. (Attempting to read past the end of the file causes an error).
5. Use POSITION FILE to move the file pointer (see discussion at Step 5 above).
6. Repeat steps 3 through 5.
7. Close the file.

   Steps 4, 5, and 6 are optional. POSITION FILE may only be used if the file is open for random access.

File Access Summary

In order to read from, or write to, a data file, the file must first be opened for access and its filename associated with a channel number. Once the file has been opened, it is then possible to read records from the file, or to write records to the file (if the proper mode has been specified). The number of bytes of data which are read in or written out is determined by the size of the precision of the variable you supply. When reading or writing records, the file is referred to by its channel number rather than by its filename. Once the processing of a data file has been completed, the filename must be disassociated from its channel number by a CLOSE [FILE] statement.

Files with random or contiguous organization may also be processed sequentially. To do this, simply position the file pointer to byte 0 of the file, and then continue processing as if the file were organized sequentially. Remember that the file pointer moves with each read or write, and will be automatically positioned and ready to access the next record.

If a file is random or contiguous, you may access individual records directly by their relative positions within the file. This is done by positioning the file pointer to the beginning of the record, or portion of a record, to be accessed. If records are fixed in length, then the algorithm:

Record number * record length

is used to position the file pointer. See the explanation of this algorithm above.

Filename Conventions

The primary requirement of the file creation process is the assignment of a unique filename. It is the unique filename that you assign to each file that enables the system to access the file for all I/O activities.

Certain conventions are to be followed when assigning a filename. These conventions were originally imposed under RDOS (developed prior to AOS) and if your goal is compatibility between RDOS/DOS and AOS or AOS/VS, they should be followed when creating files under both operating systems. RDOS filename conventions for assigning file names are:

- The filename (including extension) must not exceed 13 characters in length.
- Characters forming the filename are usually alphanumeric. However the dollar sign ($) and a period (.) preceeding the filename-extension may be included.
- Filename extensions of up to two alphanumeric characters are allowed.

An example of a filename with extension appears in Figure 5-2.

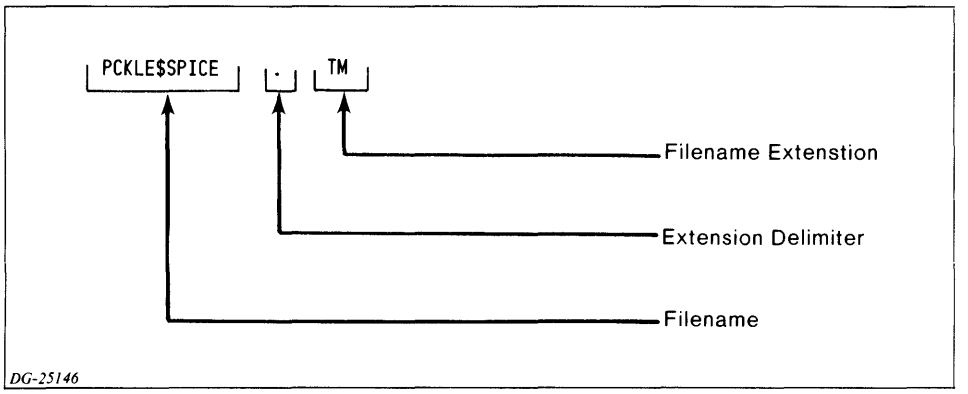


Figure 5-2 *Filename Format*

AOS and AOS/VS filenaming conventions include the following:

- Filenames have from 1 to 31 of the following ASCII characters: upper- or lowercase letters, numbers, period (.), dollar sign ($), question mark (?), and underscore (_).
- In filenames, upper- and lowercase letters are considered identical; for example, the system sees no difference between filenames "TEST" and "Test".

Business BASIC Filename Extensions

Certain filename extensions are appended to files to indicate file attributes. For example the ".TB" extension indicates that the file is a table file used with the File Maintenance(FM) utility. Similarly, the ".Sn" extension indicates that the file is one maintained by the Screen Maintenance(SM) utility.

Table 5-3 presents a summary of filename extensions that Business BASIC uses.

Business BASIC Extension	Meaning
.TB	A table file. Used with the FM utility.
.SL	A Business BASIC subroutine source or listing file in ASCII format.
.Sn	A screen file, used with the Screen Management utility, where n is the terminal type.
.SF	Indicates a utility source file listing which has comments. ".SF" files are contained in the $DOC directory which was supplied with your Business BASIC software.
.SP	Indicates a commented utility source file module which is accessed by one or more ".SF" files. ".SP" files are contained in the $DOC directory which was supplied with your Business BASIC software.

Table 5-3

The INFOS® II System

INFOS II software is a file management system that is sold as a separate product by Data General Corporation. It is available for use on AOS and AOS/VS systems. If INFOS® II software is installed on your system, you may access it from your Business BASIC programs by using the "DB" statements.

A brief description of INFOS® II is presented below. A detailed description is presented in the *INFOS® II System User's Manual* (093-000152).

The Business BASIC interface with the INFOS II system is designed to allow easy access to INFOS files. You cannot create an INFOS file with AOS Business BASIC. To create your index and database, you should use the AOS INFOS utility ICREATE. Once created, Business BASIC allows you to manipulate your INFOS file structure with all the functionality described in the *INFOS® II System User's Manual.*

Business BASIC's INFOS II interface statements are similar to interfaces of other Data General products (i.e., COBOL, IDEA, DGL, FORTRAN and PL/I). The Business BASIC INFOS II interface consists of a set of statements which behave like regular Business BASIC statements, though their format is somewhat different.

For each statement, there is a set of arguments which apply. These arguments are comparable to switches, altering the statements in various ways. Some arguments are required for particular statements, while others are optional or not applicable.

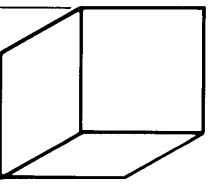
In order to access INFOS II indexes it is necessary to open them; this is done using the DBOPEN statement. To refer to an opened INFOS II file within your program, a string variable is used in a manner similar to a channel number in BASIC statements such as PRINT FILE, READ FILE, and WRITE FILE. This string variable is called the channel string, and is required for almost all INFOS II statements; it is initialized by the DBOPEN statement, and may then be used throughout the program to refer to the INFOS II file.

Assume you want to open an INFOS II file called "ACCOUNTS", and the name of the channel string used will be MASTER$, then the statement DBOPEN INFOS "ACCOUNTS",MASTER$ will perform that action for you. Once the file has been opened, you may perform any desired operation on the file through the use of the provided statements. The file will be referred to by its channel string, MASTER$.

The most common operations are to read and write records and keys; the DBREAD and DBWRITE statements are used to perform these actions. As an example, the statement DBREAD MASTER$,ACCESS=REL,MOTION=BACK, REC=MDATA$ will read from the INFOS file opened on a channel which is referred to by the channel string MASTER$; relative access would be used and the direction of motion is backward (i.e. read the previous record).

The data record that was read would be placed in the string MDATA$. Other common INFOS II operations include deleting keys and records (DBDELETE statement), reinstate logically deleted records (DBREINSTATE statement). You may also wish to perform more advanced features which INFOS II provides. These include defining subindexes, retrieving subindex definitions and linking subindexes. There are also two statements (DBSET and DBGET) which are not directly related to any one INFOS II function; they allow you to set default values for subsequent statements and retrieve values returned by previous statements respectively.

Chapter 6
Data Base Generator DBGEN, FM, and SM



This chapter describes the procedures for using the DBGEN, FM, and SM utilities. A running example is used throughout this chapter to illustrate the reasons for calling in each of these utilities. This example illustrates how to set up a simple tax application. It is designed so that you could use DBGEN, FM, and SM to recreate this example, and familiarize yourself with these utilities. Initial efforts to set up this application include the following:

- Structure a database designed to store all types of data (such as names and addresses).
- Provide a means of accessing each and every tax record for additions, later review, update or deletion.
- Design screens (forms) that can be used for data entry (new client) or retrieval (review a record) operations.

Database Structuring

In structuring a database to store records, you must first reflect on the categories of data to record. For reasons of simplicity, let's assume that the following four data categories are used for this application:

- Name
- Address
- Income
- Taxes

Business BASIC permits three approaches to structuring a database for record maintenance of the above type. They are:

- Invoking the Data Base Generator (DBGEN) utility to produce the file structure and screens of the database. Once the database has been generated, the File Maintenance (FM) utility is invoked for data entry, retrieval, or update. As a database structuring tool, DBGEN has the advantage of speed and ease-of-use. However it places certain rigid constraints on both the files and the structure which may be a disadvantage in some processing applications.
- Invoking a series of independent Business BASIC utilities (including the File Maintenance (FM), and Screen Maintenance, (SM) utilities) to structure the database. Though slower and more cumbersome, use of individual utilities permits users to precisely tailor the database and screen formats to their particular requirements.
- User-coded programs to set up and maintain a database and its indexes.

In the subsequent paragraphs we examine how the DBGEN utility can be used to structure a database and generate applicable screens for the tax record keeping application. Following this, we take a look at how the File Maintenance (FM) utility can be invoked to add, retrieve and change tax records.

Using DBGEN to Structure an ISAM Database

Before discussing the DBGEN interactive dialogue and the results it produces, it is useful to examine the overall makeup of this utility.

DBGEN Design Characteristics

The DBGEN utility (program) is an easy-to-use ISAM file builder. DBGEN automatically creates a random ISAM file with linked-available-record format. It consists of the DBGEN utility program and a CHAINed set of additional utility programs that the system enters from DBGEN. A description of DBGEN, the complete list of utilities CHAINed to by DBGEN, and the functions they implement are:

DBGEN	An interactive utility that promotes a multi-screen dialogue in which you define the parameters and characteristics of the file structure you want to create.
ICAL	An automatic utility that calculates the parameters you need for your index file. DBGEN automatically chains to ICAL. ICAL is a modification of INDEXCALC.
DBINIT	A modification of INITFILE. (INITFILE is described in the *Business BASIC Subroutines, Utilities, and BASIC CLI* manual). DBINIT is an automatic utility that initializes both the database and the indexes of the ISAM file. DBGEN passes the necessary information to DBINIT automatically.
DBOPEN	Opens the file to enable FM to write to it.
DBFM.T6	This is a modification of FM. It automatically displays the building of a Table File on your screen taking the necessary information from the files that DBGEN created.
DBPRINT	This is an interactive utility in which you respond to a prompt indicating whether you do or do not want a line printer copy of your Table File information. A yes (Y) response outputs a printed copy of the table file to the line printer. A no (N) response displays the table file information on your screen.

Collectively, all the utilities invoked throughout the DBGEN sequence implement the following processing activities:

1. Initiates an interactive dialogue that allows you to define your database structure via a multi-screen question and answer session. In this phase you define the number of records, the fields contained in these records, record length and any index files you require in your ISAM type database.
2. Builds index files from the index file parameters you specify during the interactive dialogue.
3. Initializes and opens all files (i.e. database and index files) in your ISAM structure.
4. Builds and prints a table file that contains the parameters and specifications for the database you have structured. These specifications are important in that they are essential to operation of the File Maintenance utility that you'll later invoke for file access and update.

There are three versions of DBGEN. They are:

- Double precision DBGEN
- Triple precision version of DBGEN that stores and retrieves only double precision numeric fields
- DBGENT, a triple precision version of DBGEN that stores and retrieves full triple precision numeric fields.

NOTE:

FMT should be used with files created by DBGENT, while FM should be used with files created by DBGEN.

FMPRINT only accesses double and triple precision DBGEN. It does not access DBGENT.

Entering/Exiting DBGEN

To execute DBGEN, type the following from the Business BASIC "*" prompt:

```
* RUN "DBGEN"
```

NOTE:

Do not SWAP to DBGEN via "!DBGEN" or "DBGEN".

The Data Base Generator then displays the first screen (Figure 6-1) in a multi-screen interactive sequence that continually prompts you for information about the database that is to be structured.

Exiting DBGEN at the conclusion of the interactive dialogue is automatic. That is, after the table file is displayed/printed, the asterisk (*) prompt reappears automatically to indicate that you have exited DBGEN.

If you wish to exit the DBGEN dialogue, before all prompts have been answered, press ESCape.

The DBGEN Interactive Dialogue

When you execute DBGEN, it invokes the first screen in a series of screens that prompt you for information about the database to be structured. Throughout the multi-screen dialogue you are continually prompted for information about the database structure. A summary of the interactive dialogue used by DBGEN is presented below.

NOTE:

Because the prompts of the DBGEN multi-screen dialogue are fairly self-explanatory, you can execute DBGEN and run it with minimum effort. The prompts are designed to lead you through the entire sequence without having to rely on additional reference material.

Referring now to our earlier example of the tax application, we can see that the application requires 4 record fields to accomodate the data, which consists of: name, address, income, and tax.

The first DBGEN screen displays the prompt shown below in Figure 6-1. You would respond by entering the numeral 4.

```
.....DEFINING FILE STRUCTURE.....

A RECORD FIELD IS A GROUP OF CHARACTERS OR NUMBERS DESCRIBING
 A DATA ITEM IN A DATA RECORD (E.G., NAME, ADDRESS, TELEPHONE ).

HOW MANY FIELDS DO YOU WANT IN YOUR DATA RECORDS?:  4
```

Figure 6-1 *DBGEN Interactive Dialogue, Screen 1*

After entering the numeral 4, DBGEN invokes the second screen in the interactive dialogue (Figure 6-2).

```
.....DEFINING FILE STRUCTURE.....

  THE SIZE OF A CHARACTER FIELD IS THE NUMBER OF CHARACTERS, SPACES AND
  DIGITS IT CONTAINS. THE MAXIMUM SIZE OF A NUMERIC FIELD IS 4 BYTES
  (2147483647). THE FIELDS MAY BE NUMERIC (N) OR STRING (S).
  DEFINING   4 FIELDS


  FIELD SIZE 1: 10 DESCRIPTION: NAME          (N OR S): S
FIELD SIZE 2: 15 DESCRIPTION: ADDRESS       (N OR S): S
FIELD SIZE 3:  7 DESCRIPTION: INCOME        (N OR S): N
FIELD SIZE 4:  6 DESCRIPTION: TAX           (N OR S): N
```

Figure 6-2 *DBGEN Screen 2*

During screen 2 of the dialogue, you are prompted for the following information about each field in your database:

- Field size
- Description
- String or numeric

Figure 6-2 shows the information you should enter in response to these prompts. From this illustration, we can see that a field length of 10 has been assigned to the first field (NAME). A field length of 15 has been assigned to the second field (ADDRESS). A length of 4 has been assigned to both the INCOME and TAX fields. Also note that the first two fields have been identified as strings and the last two as numeric fields. After the last entry is made, the third screen in the DBGEN interactive dialogue is invoked as shown below in Figure 6-3.

```
.....DEFINING INDEX STRUCTURE.....

AN INDEX CONSISTS OF KEYS POINTING TO COMPARABLE FIELDS IN ALL THE
DATA RECORDS. FOR EXAMPLE YOU CAN HAVE AN INDEX CONTAINING ALL THE NAME KEYS.

A NAME INDEX ALLOWS DUPLICATE KEYS (E.G., 2 SMITHS). SOME NUMERIC INDEXES
ALLOW DUPLICATE KEYS (E.G., CREDIT LIMIT).
OTHERS MUST BE UNIQUE (E.G., ACCOUNT NUMBER).

NUMBER OF INDEXES TO BE BUILT (MAX=3): 2

 FIELD NUMBER FOR INDEX 1 1 DESCRIPTION NAME      CORRECT (Y,N): Y
FIELD NUMBER FOR INDEX 2 4 DESCRIPTION TAX       CORRECT (Y,N): Y


DUPLICATE KEYS (Y OR N): Y
DUPLICATE KEYS (Y OR N): Y
```

Figure 6-3 *DBGEN Interactive Dialogue, Screen 3*

The interactive dialogue now enters the index file definition phase. Here in screen 3, the necessary prompts are presented for you to indicate which of the previously defined fields are to have related index files. In this example we are using two index files which are to be pointers to the records via the NAME and TAX fields. Duplicate keys are allowed for the NAME index and TAX index. After the last response is entered, DBGEN automatically displays a screen similar to the one shown below in Figure 6-4.

```
.....DEFINING FILE STRUCTURE.....

AN INDEX LABEL SHOULD REFLECT ITS FUNCTION. FOR EXAMPLE, THE LABEL OF
A NAME FIELD COULD BE NAMES
WARNING: TRUNCATES LABELS TO MAXIMUM OF 10 CHARACTERS.

NUMBER OF RECORDS: 1000









DATA BASE NAME TO BE: TAXES

 INDEX 1 POINTS TO NAME LABEL OF INDEX :: NAME
INDEX 2 POINTS TO TAX  LABEL OF INDEX :: TAX
```

Figure 6-4 *DBGEN Interactive Dialogue, Screen 4*

In Figure 6-4, the interactive dialogue requests the number of records for the entire data file. In this example we specify the number 1000 for the number of data records. This screen also requires that you assign a name to the database. In this case, we assign the name TAXES. The final input required by DBGEN is to enter labels for the index files. We enter NAME and TAX respectively and then press the NEW LINE key. Note that the index filename does not need to be the same as the name of the corresponding field.

DBGEN then displays the screen shown in Figure 6-5. This screen summarizes and displays the specifications that have been entered for this database.

```
      YOU HAVE DEFINED THE FOLLOWING STRUCTURE:

              DATABASE FILE.......TAXES
              CONSISTS OF.........1000 RECORDS OF 4 FIELDS EACH

FIELD   SIZE
1       10      DESCRIPTION: NAME       INDEX LABEL: NAME
2       15      DESCRIPTION: ADDRESS
3        4      DESCRIPTION: INCOME
4        4      DESCRIPTION: TAX        INDEX LABEL: TAX

CONFIRM (YES OR NO): Y
```

Figure 6-5 *DBGEN Screen 5*

The bottom line on this screen asks for confirmation of the displayed parameters. Entering a YES (or Y) response to this prompt initiates the chaining action wherein DBGEN CHAINs to the complete range of utilities mentioned earlier in this chapter (DBGEN Design Characteristics). Entering NO (or N) causes the system to redisplay screen 1 of DBGEN (to enable you to start over).

```
INDEX CALCULATION PHASE FOR:-

INDEX: NAME  INDEX: TAX
```

Figure 6-6 *DBGEN Screen 6*

The database structuring activity now proceeds automatically with the sequence first entering the index calculation phase followed by the database initialization phase. During these sequences, screens are displayed automatically with no interaction by users. The screens displayed during the index calculation and database initialization phases appear in Figures 6-6 through 6-8.

```
ADDING       NAME
ADDING       TAX
ADDING       TAXES
RECORD SIZE IN BYTES:    512
```

Figure 6-7 *DBGEN Database Initialization Phase, Screen 1*

```
END OF INIT PHASE

INDEX INITIALIZED        NAME
INDEX INITIALIZED        TAX

DATA FILE      TAXES

INCLUDES       TAXES.TB
```

Figure 6-8 *DBGEN Database Initialization Phase, Screen 2*

The index calculation and database intialization phases are quite lengthy, requiring a few minutes to complete. During this time there is no interaction with the utilities. Everything is automatic and users at the terminal simply sit back and watch. After the database initialization phase, utility DBFM.T6 automatically creates a table file for the TAXES database. Since DBFM.T6 is automatic, users do not enter any information in this phase. When the table file creation phase is complete, DBGEN displays the following prompt:

OUTPUT TO LINE PRINTER:

If you want a printed copy of the table file showing the design specifications of the database structure, enter Y. If not, enter N to display the information on the terminal screen. The output which is printed for this example is shown in the following table file printout.

```
FILENAME:      TAXES LINKED

TABLE FILE: TAXES.TB
RECORD LENGTH: 35      MAXIMUM NUMBER OF RECORDS: 1000
DBGEN FILE MAINTENANCE
DBGEN TAX RECORD (TYPE 1)
REC NAME   SEQ   DESCRIPTION            SIZE   TYPE   POS.  FORMAT   EDIT
INDEX FILE: NAME      KEY LENGTH: 10
16 NAME    1 NAME                       10     S      2     A10      U

INDEX FILE: TAX       KEY LENGTH: 4
19 TAX     1 TAX                        4      D      31    F11      N

16 NAME    1 NAME                       10     S      2     A10      U
17 ADDRESS 2 ADDRESS                    15     S      12    A15      U
18 INCOME  3 INCOME                     4      D      27    F11      N
19 TAX     4 TAX                        4      D      31    F11      N
```

The table file printout above specifies the following information:

- A record length of 35. That is the total of the field lengths specified in screen 2 plus two bytes for header information. (Note, however, that when you specify a field as numeric, it is automatically assigned a 4-byte length for double precision and a 6-byte length for triple precision, rather than the length specified.)
- The maximum number of data records, in this case 1000, which was specified in screen 4 of the DBGEN interactive dialogue.
- The labels for the index files and the record fields and also the attributes for the record fields.

At this point, DBGEN has done its job of building an ISAM structured database. This means that we are now ready to enter data records in the data file using File Maintenance (FM) as explained in the subsequent paragraphs.

File Maintenance (FM) Utility

File Maintenance (FM) is a major file access and update tool within Business BASIC. Use of the File Maintenance (FM) utility for access/update purposes assumes the pre-existance of a structured database with an attendant table file (which specifies the database parameters). FM uses the database parameters to generate a formatted screen that displays the record fields to be used for data entry, retrieval and update. FM has its own set of function keys (and related template) to accomplish file manipulation.

NOTE:

FM is equipped with additional features that are aimed at database structuring rather than simple file access/update. Such FM features are covered in detail in the *Business BASIC Technical Concepts* manual. Here we look simply at the file manipulation capabilities of FM as an access/update tool for the database we have already structured with DBGEN.

Entering and Exiting FM

To execute FM, type the following from the Business BASIC "*" prompt:

```
* RUN "FM"
```

The File Maintenance utility responds by displaying the following:

```
DATA FILE MAINTENANCE REV 3.52 10/28/77
FILENAME:
```

Enter the filename of an existing file (which has an accompanying table file with the .TB extension) in your directory. To continue with the previous file which we generated using DBGEN, enter:

```
TAXES
```

FM now responds by displaying the first of a series of "record" screens that have a bracketed ([ ]) prompt field at the upper right corner of the screen. Once the cursor is positioned to the displayed brackets you may exit FM any time as follows: Press the STOP function key as indicated on the FM function key template.

File Maintenance (FM) Operational Modes

When the first record screen appears at your terminal it means that FM has opened your database for I/O access and you can now exercise any of the various FM operational modes. The operational modes are implicit in the names of the function keys as shown on the function key template A brief summary of FM operational modes is presented below:

- ADD a record. To enter this mode, the ADD function key is used. This mode permits the addition of a new record in numerical sequence or to add a record where a pre-existing record has been deleted.
- FIND a record. To enter this mode, the FIND function key is used. The dialogue for this mode requires that you provide either a record number or an index key so that FM can locate the record you specify.
- FIND NEXT record. To enter this mode, the FIND NEXT key is used to access the next record in numerical sequence. The FIND function must be used prior to this function.
- CHANGE a record (after exercising the FIND or FIND NEXT function). To enter this mode, the CHANGE key is used. This mode allows you to change the contents of character fields within a record.
- DELETE a record (after exercising the FIND or FIND NEXT function). To enter this mode, the DELETE key is used. This mode allows you to delete an entire record meaning that the space is freed-up for re-use via the ADD function.
- FORMAT CHANGE and PAGE CHANGE modes. These are more complex modes that are described in the *Business BASIC Technical Concepts* Manual.

FM Function Keys

Entering FM activates the terminal function keys in a special way to enter the operational modes summarized above. When the cursor is in the command brackets, you must enter a command. If you have a DASHER® display terminal, you can use the File Maintenance function keys listed in Table 6-1. (See the Preface of this manual for information on ordering FM templates). Other terminals (as well as DASHER® displays) can run the roll mode version (FM.RM). If FM doesn't recognize your terminal, it will execute the roll-mode version.

The FM templates for use with the DASHER® D2 and DASHER® D200 terminals are shown in Figures 6-9 and 6-10 respectively.

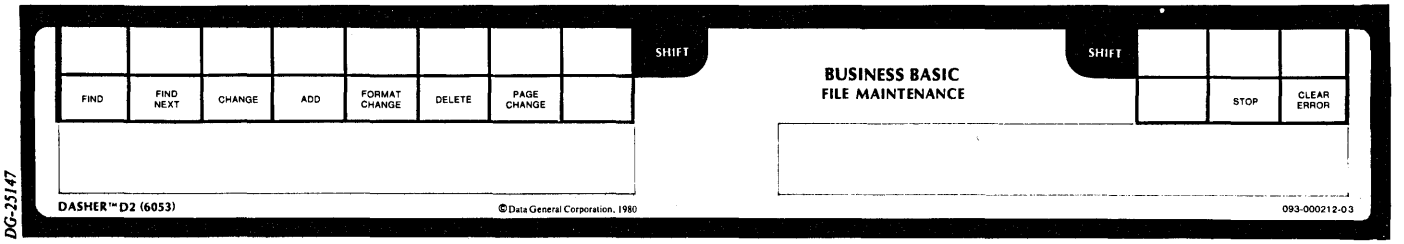


Figure 6-9 FM Template for DASHER® D2 Terminal

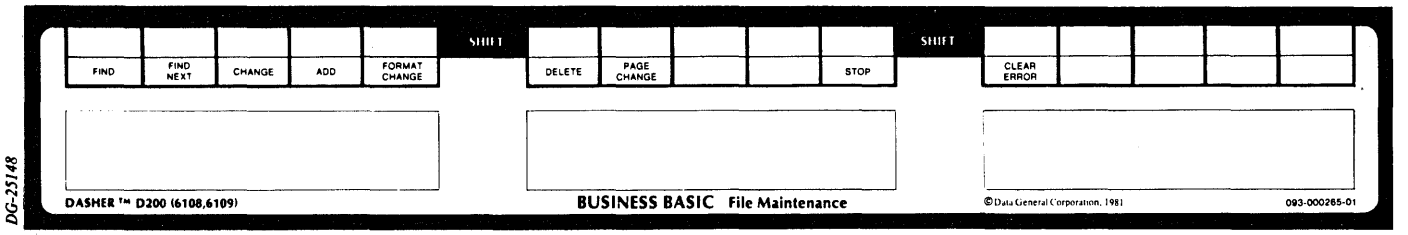


Figure 6-10 FM Template for DASHER® D200, D400, D450, G300 Terminals

Key	Function/ Command	Roll-Mode Command	Description
1	FIND	FIND	Find a record.
2	FIND NEXT	NEXT	Find next record.
3	CHANGE	CHANGE	Change a record.
4	ADD	ADD	Add a new record.
5	FORMAT CHANGE	FORMAT	Change record format.
6	DELETE	DELETE	Delete a record.
7	PAGE CHANGE	PAGE	Change page.
10	STOP	STOP	Stop FM.
11	CLEAR ERROR	not applicable	Clear an error.
n/a	not applicable	DISPLAY	Display a record.
n/a	not applicable	AUTO	Automatic display.
12	PREVIOUS FIELD	not applicable	Previous field.
NEW LINE or CR	NEXT FIELD	not applicable	Next field. Return to command brackets.

Table 6-1

File Access And Update With FM

To illustrate the file access/update process with FM, we return to the tax example. The database for this example is the one we structured earlier, using the DBGEN utility. We will use FM to enter tax data records of clients. To execute FM, type

```
* RUN "FM"
```

FM then displays the dialog in Figure 6-11.

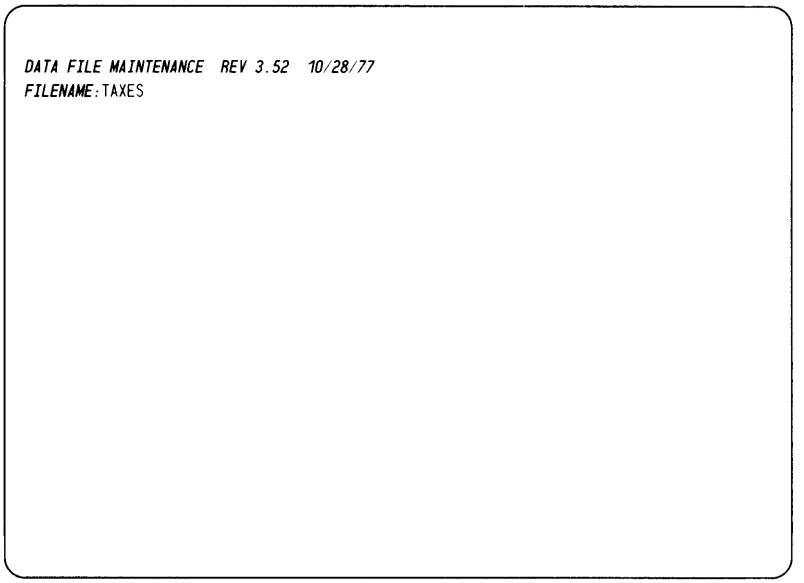


Figure 6-11

Entries

FILENAME: For this example, we enter the filename "TAXES". FM then displays the first screen in the database to be used for data entry. The format of this screen is as shown below:

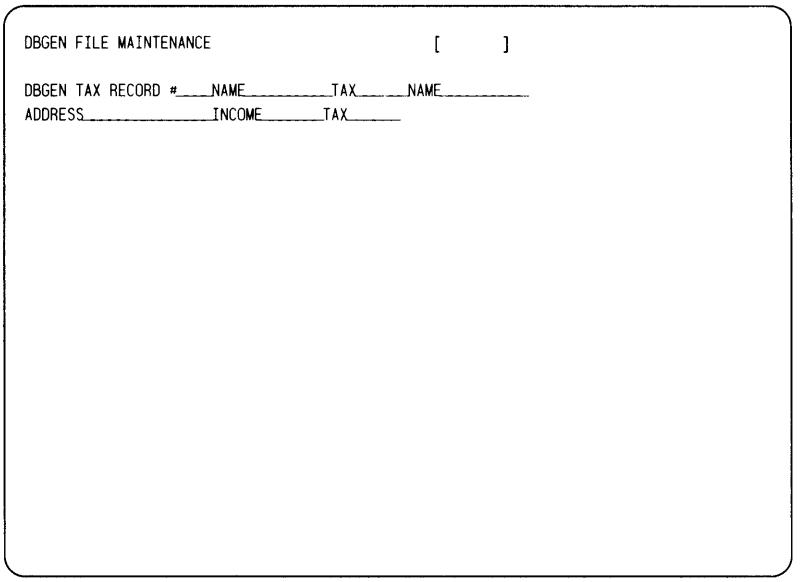


Figure 6-12

The general sequence of presentation in the screen shown in Figure 6-13 is:

- Record number
- Index file labels (NAME and TAX)
- Record field labels (NAME, ADDRESS,INCOME and TAX)

Let's now assume that in this database accessing session you intend to enter tax and related data for five clients: Cochran, Brown, Smith, Tims and Adams. Figures 6-13 through 6-17 show the particular tax record data that must be entered for each of these five clients.

NOTE:

You use the ADD function key to invoke and enter data to each successive record screen. To advance to each successive field within a record screen you use the NEW LINE key on AOS and AOS/VS systems and the CR key on RDOS/DOS systems. When you advance the cursor past the last field in the record, the cursor returns to the brackets permitting selection of the ADD function or any other function.

```
DBGEN FILE MAINTENANCE                          [ADD    ]

DBGEN TAX RECORD # 1  NAME  COCHRAN    TAX  1234
NAME COCHRAN     ADDRESS  MARLBORO     INCOME  14567  TAX 1234
```

Figure 6-13

```
##
DBGEN FILE MAINTENANCE                          [ADD    ]

DBGEN TAX RECORD # 2  NAME BROWN     TAX 4567
NAME BROWN      ADDRESS HUDSON      INCOME 18000 TAX 4567
##
```

Figure 6-14

```
DBGEN FILE MAINTENANCE                          [ADD    ]

DBGEN TAX RECORD # 3  NAME SMITH     TAX 4314
NAME SMITH    ADDRESS BOSTON      INCOME 24567 TAX 4314
```

Figure 6-15

```
DBGEN FILE MAINTENANCE                          [ADD    ]

DBGEN TAX RECORD # 4  NAME TIMS      TAX 4137
NAME TIMS     ADDRESS W. ROXBURY INCOME 22000 TAX 4137
```

Figure 6-16

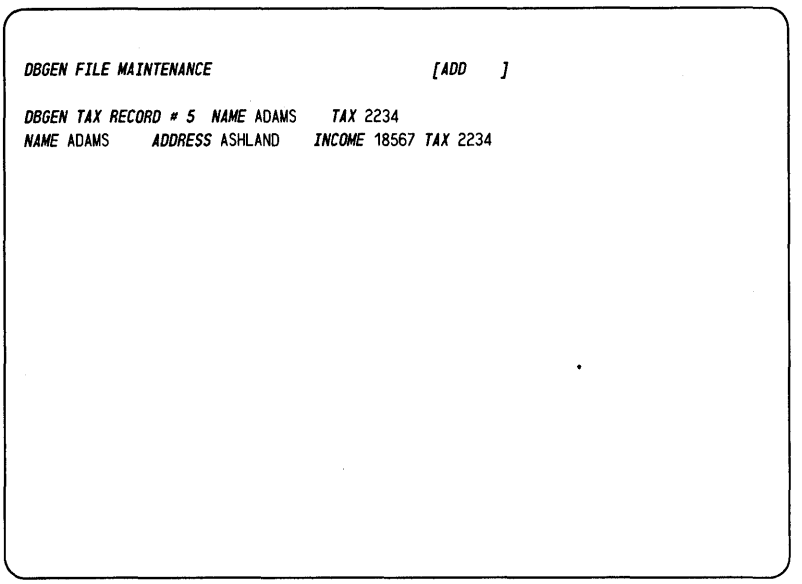


Figure 6-17

To find a record by referencing one of its indexes, press the FIND function key; the cursor moves to the record # field. Now, to access the record by means of the NAME index, first press the NEW LINE key to advance the cursor to the NAME index field. You must then type in the name of the record you are seeking. If you enter ADAMS, the ADAMS tax record is displayed on the screen.

Use of the CHANGE or DELETE function keys must always be preceded by use of the FIND or FIND NEXT functions. To change the contents of a field, press the CHANGE key. This action causes the cursor to be positioned at the beginning of the second field. The keys are now used to enter any desired changes in the content of a given field. The NEW LINE key is used to advance the cursor from field to field.

To delete a record, you must first locate the record via the FIND function. You must then press the DELETE key and the record is erased from the database. Significantly, you can now ADD a record at this record position if desired.

To clear an error, press function key 11 (CLEAR ERROR) and restart the function.

Screen Maintenance (SM) Utility

The essential purpose of the Screen Maintenance (SM) utility is to allow you to custom tailor screen formats to your company's particular business needs. The DASHER terminals are capable of displaying 80 columns by 24 rows of ASCII characters. And it is the SM utility that allows you to layout this 1,920 character field as you like. Since FM, as illustrated in Figure 6-14, has layed out a screen for us (in the labels for RECORD#, NAME, etc., and blank fields for data entry), the question may arise as to why a screen maintenance utility is necessary. The answer is that SM provides users far greater flexibility in formatting screens than the rigid display arrangement always employed by FM.

Figure 6-19 illustrates one of the ways in which you might want to format a data entry screen for use in the TAXES application which we built with the DBGEN and FM utilities.

Some of the more important features built into SM are:

- Its own function key template that permits cursor manipulation and positioning to implement screen formatting activities.

- A group of special characters that can be used to assign screen field attributes.
- Subroutines FORM.SL, PROTFORM.SL, and UNFORM.SL that can be called to implement screen formatting activities. Detailed coverage of these utilities is presented in the *Business BASIC Subroutines, Utilities, and BASIC CLI* manual.

Take a look at the screen shown in Figure 6-18. Notice that on this illustration the data entry/retrieval fields (those shown as a a string of "L" characters) are shown as underlined while the labels for these fields (Name, Address ETC.) are not. This is to differentiate between high intensity (underlined) and low intensity (non-underlined) fields. High intensity fields are considered unprotected in the sense that you can enter data into them. Low intensity fields (NAME, ADDRESS, INCOME, and TAX) are considered as protected fields in the sense that they serve as fixed labels for the data entry fields. No data may be entered over these fields in the data entry process. The advantage of protected fields is that they provide a form that positions you to the correct locations. Thus, after you've created a screen and you are ready to enter data into it, the cursor automatically positions itself to the first input field. Then, after you type NEW LINE, it proceeds to the next input field, and so on.

Table 6-2 lists all of the special characters which are used for formatting input/output in Screen Maintenance.

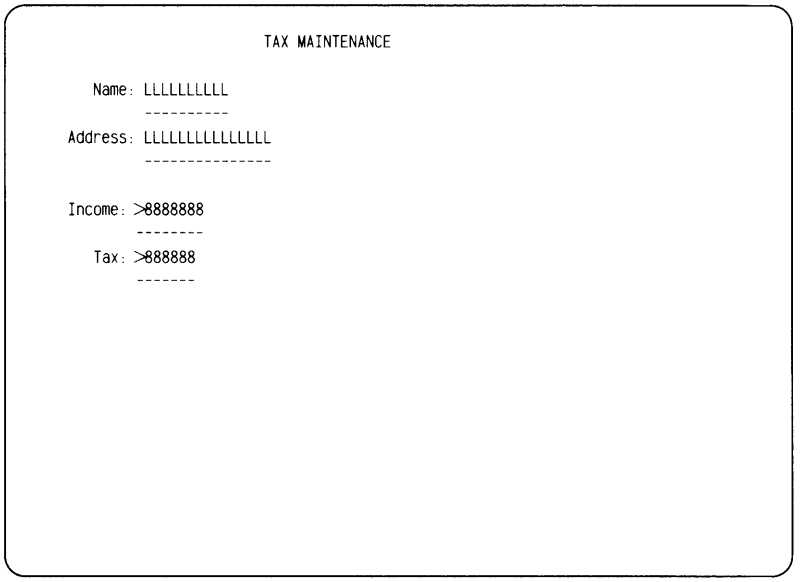


Figure 6-18 *Formatted Screen for TAXES Example*

Character	Field	Data type
8	unprotected	numeric (fill)
9	protected	numeric (fill)
-	both	at the beginning of a numeric field, allows negative values.
.	both	fixed decimal location in a numeric field
:	both	relative location for a floating decimal point.
X	unprotected	alphanumeric field (fill).
Y	protected	alphanumeric field (fill).
L	unprotected	alphanumeric field following lowercase characters (fill).
M	protected	alphanumeric field following lowercase characters (fill).
C	unprotected	crammed field (fill).
D	protected	crammed field (fill).
>	both	right-justified field, put only in first column field.

Table 6-2 Special Characters for Formatted Input/Output

Special Format Characters

When you fill a protected field with the character 9, it assures that only numeric data can be output to the field. When you fill an unprotected field with the character 8, it assures that only numeric data is allowed as input to the field. A negative character in the first position of either field will allow negative input/output, and a decimal point in a location will allow a decimal point to be input at that location and will output a decimal point in that location. A floating decimal (:) allows you to type a decimal point anywhere within the input (unprotected) field. A floating decimal point in an output (protected or unprotected) field allows you to output a decimal anywhere in the field.

Your data type is alphanumeric if it is not strictly numeric, decimal, or signed numeric. X/Y allows an alphanumeric field, L/M allows lowercase alphabetic characters (special case of X/Y), and C/D allows a crammed field only -- input will be crammed after it is typed, and output will be uncrammed before it is printed.

The right-justify character (>) justifies any output to the right of the field. Use these formats if you are going to use the FORM.SL subroutine. FORM.SL will return the variable E with a value of 128 if the input is not successful, so you can issue an error message (using UNFORM.SL) to have the user rekey the screen input. Each of these utilities are described in detail in the *Business BASIC Subroutines, Utilities, and CLI* manual.

Entering and Exiting SM

To invoke the Screen Maintenance (SM) utility you must already be in Business BASIC and should have the SM function key template in place. Invoke SM by typing:

```
* RUN "SM"
```

from the Business BASIC "*" prompt.

The Screen Maintenance utility responds by displaying the following headline and prompt:

DG SCREEN MAINTENANCE REV:x.xx mm/dd/yy

SCREEN FILENAME: You must reply by typing the filename of an existing screen file or a new screen file that you wish to create. The screen filename can be any name, but by convention, we use a filename with a .Sn extension, where n is the terminal type (6 for 6052, 6053, and D200 terminals). For example,

```
SCREEN5.S6
```

SM now responds by displaying the double angle bracket (>>) prompt. After this you can begin the process of formatting your screen with the SM function keys. Once you've entered SM, you'll find that certain keys such as "NEW LINE", "CR", and others are disabled. However the character,digit and cursor movement keys remain enabled which together with the SM function keys allow you to format the screen as you desire.

Once the double angle bracket prompt is displayed, you may exit SM anytime through the following keyboard sequence:

Press the ENTER COMMAND function key as indicated on the SM function key template.

Type STOP adjacent to the double angle bracket (>>) prompt.

Press the PROCESS COMMAND function key.

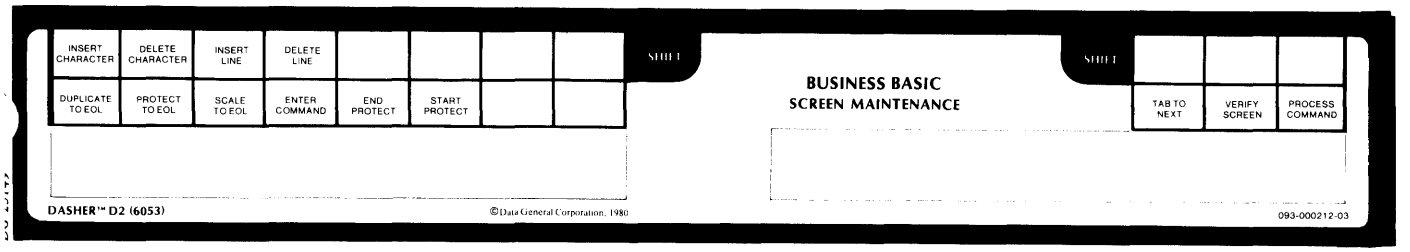


Figure 6-19 *SM Template for DASHER® D2 Terminal*

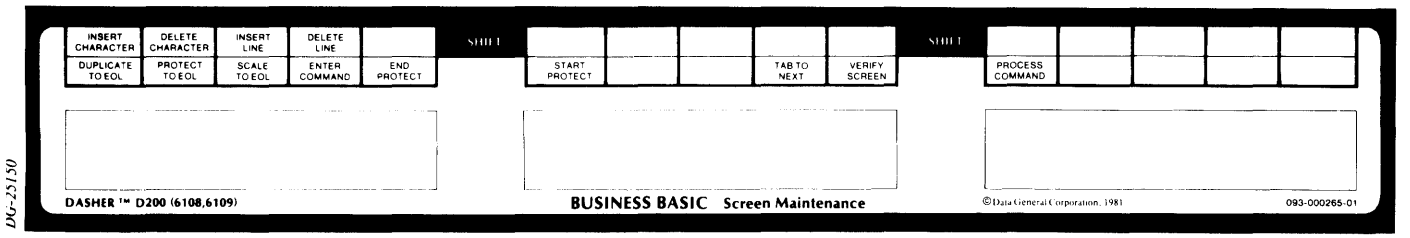


Figure 6-20 *SM Template for DASHER® D200, D400, D450, G300 Terminals*

The asterisk(*) prompt of Business BASIC now reappears indicating that you have exited SM.

The SM Function Key Template

The SM function key templates for the DASHER D2 and DASHER D200 terminals are shown in Figures 6-19 and 6-20 respectively.

A summary of the screen formatting functions implemented by each of the SM function keys now follows.

DUP TO EOL (F1): This duplicates the previous line to the end of the current line. The current line will be exactly like the previous line.

INSERT CHAR (Shift-F1): This key creates a space before the cursor allowing you to insert a character.

PROTECT TO EOL (F2): After a "Start Protect," this function key protects (low intensity) all positions from the cursor to the end of the line.

DELETE CHAR (Shift-F2): This deletes a character at the cursor.

SCALE TO EOL (F3): This fills the line from the cursor to the end with repeating sequences of "1234567890123...". This is useful as a scale in case you have difficulty judging the number of spaces you've moved the cursor. The scale gives you the correct number. The cursor moves back to the end of the line so that you can type over the scale. The scale remains on the screen while you are creating it. However, it is not there later when you add the data.

INSERT LINE (Shift-F3): This inserts a new blank line before the current line.

ENTER CMND (F4): This key positions the cursor to the command line so that you can type one of the SM commands.

DELETE LINE (Shift-F4): This deletes the current line.

END PROTECT (F5): This key ends a protected field and starts an unprotected one. After using this key, you should type into the unprotected field the special character for the data type you expect to input. The special characters are in Table 6-1.

START PROTECT (F6): This key starts a protected field or ends an unprotected one. You can insert any type of characters in a protected field as prompts for input. Business BASIC protects these prompts at runtime. You can't input any data into a protected field.

TAB TO NEXT (F9): This moves the cursor to the next unprotected field.

VERIFY SCREEN (F10): Displays the current version of your screen. You can press this key to see what you have on your screen. After a DISPLAY command, this key displays only protected fields.

PROCESS CMND (F11): After you type a command on the command line such as READ, WRITE, PRINT, DISPLAY, or STOP, you must press this key to process the command.

Using SM-Defined Screens

There are two ways in which you can use the screens that you design with SM:

- By specifying the screen(s) you have designed with SM in a special phase of the FM utility
- By writing a program which accesses the screen(s) you have designed with SM

Using SM Screens with FM

There is a phase of the File Maintenance (FM) database structuring sequence (described in the *Business BASIC Technical Concepts* manual) which allows you to impose your SM designed screen as the display vehicle for data entry/retrieval operations to/from the database. This means that whenever FM is invoked at a later time for file access/update, the screen you have designed with SM appears at the CRT display. Although the screen becomes a part of FM, it still remains in your directory for access and update via SM.

Using SM Screens in Programs

You may write a Business BASIC program to use one or more screens, process input from the terminal operator, display values in different fields, display error messages for incorrect data types by the operator, or make a field blink. You perform these actions with the four screen usage subroutines:

UNFORM.SL produces unformatted screen input and output including displaying a screen from the screen file, positioning the cursor to a field, reading data typed by the operator, outputting a value to a screen field, locking the keyboard and displaying an error message.

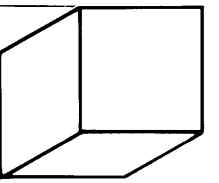
FORM.SL produces formatted screen input and output converting a data value to the specifications you supplied when defining the screen.

PROTFORM.SL converts a high intensity field to low intensity and vice versa.

BLNKFORM.SL starts and stops the blinking of a screen field.

Instructions for how to use these four subroutines, as well as SM, are presented in the Business BASIC Subroutines, Utilities, and BASIC CLI manual.

Related Documents



Basic BASIC **069-000003**
This manual introduces the BASIC computer language and covers its elementary commands and statements. Its is written for the novice who has no previous knowledge of BASIC.

Business BASIC Technical Concepts
(AOS/VS, AOS, RDOS, DOS) **093-705004**
Presents technical information on file structures, INFOS interface, File Maintenance, and Screen Maintenance. It also contains a glossary, and error message listing. It is intended to be used as a guide for programmers.

Business BASIC Commands, Statements, and Functions
(AOS/VS, AOS, RDOS, DOS) **093-705005**
An alphabetical directory of Business BASIC statements, commands, and functions. It is intended to be used as a reference manual for programmers.

Business BASIC Subroutines, Utilities, and BASIC CLI
(AOS/VS, AOS, RDOS, DOS) **093-705006**
An alphabetical directory of Business BASIC subroutines, utilities, and the BASIC CLI commands. It is intended to be used as a reference manual for programmers.

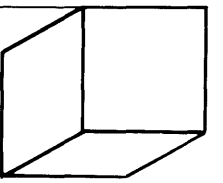
Business BASIC System Management
(AOS/VS, AOS, RDOS, DOS) **093-705007**
Describes how to load and generate Business BASIC on AOS/VS, AOS, RDOS, and DOS. It is intended for the system manager or system operator.

Business BASIC (AOS/RDOS/DOS) Reference Card **069-705010**
A pocket-sized reference card for Business BASIC programmers on AOS/VS, AOS, RDOS, or DOS.

DASHER® D2 File Maintenance and Screen Maintenance Template **093-000212**

DASHER D200 File Maintenance and Screen Maintenance Template **093-000265**

Index



reader comment form

A Guide to Using Business BASIC

069-000028-01

Your comments will help us improve the quality of this publication. They will be carefully reviewed by the writers. Please refer to page numbers if appropriate.

DID YOU FIND THE MATERIAL:

	YES	NO		YES	NO
• Useful?	☐	☐	• Well illustrated?	☐	☐
• Complete?	☐	☐	• Well written?	☐	☐
• Accurate?	☐	☐	• Easy to read?	☐	☐
• Well organized?	☐	☐	• Easy to understand?	☐	☐

COMMENTS:

_____

_____

_____

_____

_____

_____

_____

_____

HOW DID YOU USE THIS PUBLICATION?

- ☐ As an introduction to the subject
- ☐ For information about operating procedures
- ☐ To instruct in a class
- ☐ As a student in a class
- ☐ As a reference manual
- ☐ Other *(please explain)*:

Name _____ **Title** _____

Firm _____ **Date** _____

Street _____ **State** _____

City _____ **Zip** _____

First fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 WESTBORO, MASS 01580

POSTAGE WILL BE PAID BY ADDRESSEE:

DataGeneral

ATTN: Small Business Systems Documentation
MS F213
Westboro, Ma. 01580
USA

Second fold

CUT ALONG DOTTED LINE

THIS IS THE SPINE FOR YOUR NOTEBOOK COVER

CUT ON DOTTED LINE

order number 069-705023
for matching cover insert.

order number 069-705022
for 1 inch black ring binder
to hold all inserts.