SOFTWARE MANUAL

AlphaBASIC

USER'S MANUAL

DWM-00100-01

REV. B00

alpha micro

SOFTWARE MANUAL

AlphaBASIC

USER'S MANUAL

DWM-00100-01

REV. B01

alpha micro

NOTE: This printing of the manual contains the contents of Change Page Packet #1 for the "AlphaBASIC User's Manual", (DSS-10000-04), which may be ordered separately from Alpha Micro.

First Printing: 1977
Second Printing: October 1980
Third Printing: 30 October 1980

This book reflects AMOS Versions 4.4 and later

ALPHA MICROSYSTEMS
17881 Sky Park North
Irvine, CA 92714

Table of Contents

PREFACE

AlphaBASIC is a particularly powerful version of BASIC that has been expanded in several important areas. The following chapters describe the AlphaBASIC features and operations.

We assume that you are already familiar with the BASIC programming language, and that you are interested in getting to know AlphaBASIC. Therefore, this book emphasizes features of AlphaBASIC that differ from those of conventional BASICs, without going into much detail on standard BASIC statements and commands.

This book is not a BASIC tutorial, but is a technical manual intended for the experienced BASIC programmer. We encourage you to contact your local Alpha Micro dealer for help in answering specific questions you may have about AlphaBASIC.

BIBLIOGRAPHY

If you are not familiar with BASIC, you may be interested in taking a look at one or more of the books listed below. We have found these books to be helpful to the beginning BASIC programmer.

Albrecht, R. L., et al.
BASIC, 2nd Edition
John Wiley & Sons, 1978

Brown, J. R.
Instant BASIC
Dilithium Press, 1977

Cassel, D.
BASIC Made Easy: A Guide to Programming Microcomputers and Minicomputers
Reston Publishing Co., 1980

Dwyer, T. and Critchfield, M.
A Bit of BASIC
Addison-Wesley, 1980

Dwyer, T. and Critchfield, M.
BASIC and the Personal Computer
Addison-Wesley, 1978

Hirsch, S. C.
BASIC Programming: Self Taught
Reston Publishing Co., 1980

Kemeny, J. G. and Kurtz, T. E.
BASIC Programming, 3rd Edition
John Wiley & Sons, 1980

CONVENTIONS USED IN THIS MANUAL:

To make our examples concise and easy to understand, we've adopted a number of graphics conventions throughout our manuals:

{} Optional elements of a BASIC statement or command. When these symbols appear in a sample statement or command, they designate elements that you may omit.

_____ Underlined characters indicate those characters that AMOS prints on your terminal display. For example, throughout this document you see an underlined dot, .., which indicates the prompt symbol that the operating system prints on your terminal when you are at AMOS command level.

(RET) Carriage return symbol. The (RET) symbol marks the place in your keyboard entry to type a RETURN (i.e., hit the key labeled RETURN). For example: ".BASIC (RET) " tells you "After an AMOS prompt, type BASIC and a RETURN."

^ Indicates a Control-character. If you type a Control-C in the compiler mode of AlphaBASIC, for instance, you see a ^C on your terminal display. (Refer to the AMOS User's Guide, (DWM-00100-35), for more information on Control-characters.)

CHAPTER 1

INTRODUCTION TO ALPHABASIC

The acronym BASIC stands for Beginners' All-purpose Symbolic Instruction Code. BASIC is a higher-level programming language created to be a versatile tool for learning computer programming, and also to provide a relatively simple language for a wide variety of applications. But today, BASIC is more than a learning tool or a beginner's tool for higher-level programming. It can be said that most programming on small, interactive systems is done in BASIC. This is in part because of the inherent similarity of BASIC to the English language.

Over the years since its inception, BASIC has been added to and modified as new concepts of programming have emerged. Some implementations of BASIC are more extensive than others; the use of these extended versions allows the programmer a wider range of applications, greater ease in programming, or greater efficiency and speed.

AlphaBASIC is just such an extension of the BASIC language, with several features not found in other implementations. These features not only enhance the performance of traditional uses of the language but also make business applications easier to program. For instance, programmers familiar with COBOL's powerful hierarchial data structures will appreciate AlphaBASIC features which make data manipulation and assembly language subroutine linking similarly convenient. Floating point hardware in the processor is fully supported, greatly increasing the speed of mathematical computations.

AlphaBASIC runs in one of two modes: interactive or compiler mode. Interactive mode operates much like a traditional interactive interpreter; that is, you create, alter and test your program which resides totally in memory. This mode is convenient for the creation and debugging of new programs or the dynamic alteration of existing programs. Compiler mode is more useful for programs which are to be put into production use, or for testing programs which are too large to fit in memory in the interactive mode. In compiler mode, you compile the program at monitor level and store the compiled object code on the disk. During the actual running of the compiled program, only the object code and a minimal run-time execution package need to be in memory, thereby conserving space. The compiler and the run-time package are both written as re-entrant programs. This means that in a timesharing environment, any or all users who are running or

debugging programs may optionally share one copy in system memory of the compiler and the run-time package. Once created by the compiler, the object programs (also known as compiled programs) are also totally re-entrant and sharable, thereby further reducing memory requirements if several users desire to run the same application program.

AlphaBASIC supports floating point, string, binary and unformatted data formats. All data formats may be simple variables or array structures. In addition, the unique memory mapping system allows you to specify the ordering of variables in prearranged groupings for more efficient processing. This system is similar to the data formatting capabilities of the COBOL language and lends itself well to business applications where the manipulation of formatted data structures is of prime concern.

Variable names are not limited to the single character and single digit format of many BASICs, but may be any number of alphanumeric characters in length, as long as the first character is alphabetic. This is another feature which makes AlphaBASIC well suited for business applications. Since the source code is compiled and need not be in memory when the program is eventually run, the length of the variable name is not a significant concern. Label names may also be used to identify points in the program for GOTO and GOSUB branches. Label names are alphanumeric and help to clarify the program structure (for example, EXIT'ERROR: or EVALUATE'ANSWER:).

CHAPTER 2

INTERACTIVE AND COMPILER MODES

The major purpose of this chapter is to explore the differences between the two modes in which you can use BASIC. We will also discuss how to create, compile, and run your programs.

The AlphaBASIC system consists of three programs: RUN.PRG (the monitor level BASIC run-time package), COMPIL.PRG (the monitor level disk-based BASIC compiler), and BASIC.PRG (which combines an interactive compiler and run-time package to simulate a BASIC interpreter). You use RUN and COMPIL from AMOS command level to run and compile AlphaBASIC programs outside of BASIC. You use the BASIC.PRG program when you want to use AlphaBASIC as an interactive interpreter.

Your choice of interactive or compiler mode depends on several factors: your personal preference, the amount of memory you have in your partition, the stage of development your program is in, and the physical form of your program.

Interactive mode simulates a BASIC interpreter by allowing you to deal directly with BASIC. This is the mode that most BASIC users are probably familiar with. Interactive mode permits direct editing of the source program in memory and immediate feedback as each program line is edited. In this mode you are "in" BASIC, and can use AlphaBASIC's unique program debugging features. To execute a program in this mode, you must first load in or create in memory your uncompiled source program. After you have finished compiling and executing the program, you are still in BASIC, and are not returned to monitor level until you use the BASIC command BYE.

Compiler mode allows you to compile programs from the monitor level without ever entering BASIC. You first create the source program (a .BAS file) using one of the system text editors. Then, from the AMOS monitor level you compile the program. The compiler automatically saves this compiled version of your source program (called the "object program") on the disk as a .RUN file; the file is available for execution then or later using the AlphaBASIC run-time package, RUN.PRG. After you execute the object program from the monitor level, the run-time package returns you to monitor level.

Whether you use interactive or compiler mode, the resulting object program is re-entrant, and may be loaded into system memory for use by multiple users.

2.1 INTERACTIVE MODE

Perhaps the major advantage of interactive mode is that it allows you to "talk to" BASIC while you are creating or editing your program. You are free to enter entire programs which will be executed when you use the RUN command, or you can enter single statements outside of a program for direct execution. You can interrupt a program and, since you are still in BASIC, can display and change variable values and then resume program operation. You will probably be most interested in using interactive mode if the interactive nature of the compiler is of particular use to you (for example, if you are new to AlphaBASIC and want to try out various statements and small programs, or if a program is in an early development stage and you want to make use of interactive mode's debugging features).

One disadvantage in using interactive mode has to do with memory requirements. Your source program, your object program, and BASIC.PRG all reside in memory at the same time. In addition, BASIC loads into memory the BASIC run-time package, RUN.PRG. The BASIC.PRG file itself is fairly large, since it contains a compiler as well as the code that allows it to simulate an interactive interpreter. This all means that using BASIC in interactive mode uses up more memory than compiling and executing a program outside of BASIC.

Because BASIC.PRG is re-entrant, you may place it in system memory to save room in user partitions. (If you do so, you may also want to place RUN.PRG in system memory.) However, since BASIC is a fairly large program, you probably will only want to put it in system memory if most of the users on your system do a great deal of BASIC program development.

To use interactive mode, at monitor level type BASIC followed by a RETURN:

```
.BASIC (RET)
```

When BASIC is ready to communicate with you, you see the prompt:

```
READY
```

You are now inside BASIC.

2.1.1 Loading, Creating, and Saving BASIC Programs

To load a source program into memory, use the LOAD command. For example:

```
LOAD NEWPRG[23,4] (RET)
```

BASIC will load in the specified .BAS file. (This program file could have been created using one of the system text editors, or might have been saved from a previous interactive mode session.) Or, instead of loading in an existing program, you can start creating a new source program by simply typing in the program. Editing the program takes place in the conventional manner, by typing each line with its line number first. BASIC keeps lines in sequence automatically, so you may enter them in any numeric order. To edit a program line, you must re-type the entire line. As you enter program lines, BASIC scans that line looking for syntax errors. If you enter a line incorrectly, BASIC will tell you so. For example:

```
READY

10 HELP WHAT AM I DOING? RET
syntax error
```

If you want to save a source program, use the SAVE command. For example:

```
SAVE NEWPRG RET
```

The command above saves the source program NEWPRG.BAS as a disk file in the account you are logged into. You can save the compiled version of that program by specifying the .RUN extension:

```
SAVE NEWPRG.RUN RET
```

If you have not previously compiled the source program, or if you have changed the program since the last time you compiled it, BASIC automatically compiles it for you when you save a .RUN file to ensure that you are saving the most current version.

If you try to save a .RUN file when there is no source program in memory, BASIC reports:

```
No source program in text buffer
```

Since there is no way to convert an object file back to a source program file, you will want to save both the .BAS and .RUN versions of your program. (For information on the SAVE and LOAD commands, see Sections 9.8 and 9.11.)

2.1.2 Direct Statements

Program statements that do not begin with a line number are considered direct statements, and BASIC executes them immediately. For example:

```
READY
A=5 RET
PRINT A+4 RET
 9
```

Although it looks as if it is being interpreted, a direct statement is actually compiled, then it is applied against the current set of defined variables. You can define variables and change variable values using direct statements.

Certain statements are meaningless as direct statements, and so are not allowed (for example, RESUME, GOSUB, etc.).

BASIC allows multi-statement lines as direct statements. (Multi-statement lines are lines which contain more than one statement; the statements are separated by colons.) As you enter direct statements, BASIC checks them to see that they are in proper form and that they are legal for use as direct statements. You see an error message if you enter a statement incorrectly, or if it is not a legal direct statement.

2.1.3 Compiling and Running a Program

Although interactive mode simulates an interactive interpreter, in operation BASIC.PRG is a full compiler. As you enter a direct statement, BASIC compiles it and gives you immediate feedback. Whenever you change the source program in memory, BASIC sets a switch that indicates that the program must be re-compiled before it is executable again.

The source program that you have loaded in from the disk or created while in BASIC resides in memory. Before you can execute that program, it must be compiled. Running in interactive mode always involves the compilation and running of a source program which is in memory, and never includes running a saved disk object program directly. Also, in interactive mode, you may compile only the program currently in memory. NOTE: To erase anything in memory in preparation for loading in a new program or creating a new program, use the NEW command. If you do not erase the program in memory, BASIC will merge the new program into whatever is in memory. If any line numbers from the new program duplicate line numbers of an old program in memory, the new lines will replace the old lines in memory.

To execute the program, use the RUN command. NOTE: If you try to use any of the execution commmands (e.g., RUN or CONT), and if the program has been changed since the last time it was compiled, BASIC automatically re-compiles it for you before executing the program. Therefore, if you need to compile the source program and then run it, you may simply use the RUN command, and BASIC will compile and execute the program for you. For example:

```
READY

10 REM This is a small program (RET)
20 FOR I = 1 TO 5 (RET)
30 PRINT "Little tasks make large return." (RET)
40 NEXT I (RET)

RUN (RET)
COMPILE
Compile time was 0.13 seconds
Little tasks make large return.
Little tasks make large return.
Little tasks make large return.
Little tasks make large return.
Little tasks make large return.
Runtime was 0.40 seconds

READY
```

To just compile the program, but not run it, use the COMPILE command. For example:

```
COMPILE (RET)
Compile time was 0.13 seconds
```

Once the program is compiled, the object code resides in memory along with the source program. You can write it out to disk as a .RUN file by using the SAVE command and specifying the .RUN extension.

2.1.3.1 Compiler Options - You may specify the /O compiler option to the interactive compiler. The /O option tells BASIC to strip out any references to line numbers in your compiled object code. It does not change your source program. By removing line number references from your object program, you ensure that your compiled program will be smaller and will run faster. However, if an error occurs while executing the program, the resulting error message will not show the number of the line where the error occurred.

2.1.4 Debugging Features

A unique feature which is very useful for debugging programs is the single-step command. Every time you type a line-feed alone on a line, BASIC lists and executes the next program statement. At that point you can inspect variables or alter their values before you continue program execution (via the CONT command or typing another line-feed). Note that any change in the source program results in BASIC re-compiling the program before the next single-step command is actually carried out. See Chapter 9, "Interactive Command Summary," for more information on single-stepping programs, and on setting and clearing breakpoints.

2.2 COMPILER MODE

Compiler mode consists of using the disk-based compiler, COMPIL, and the run-time package, RUN, at monitor level to compile and execute programs without entering BASIC.

Although you do not have the interactive features of AlphaBASIC available to you in compiler mode, you do have the advantages of being able to compile source programs that are too large to fit into memory, and of reducing the amount of memory you need to compile and execute programs. Remember that interactive mode keeps the BASIC interactive compiler, the run-time package, and your source program all in memory at the same time. When you compile a program in interactive mode, the object code also resides in memory.

On the other hand, in compiler mode, only COMPIL and RUN need reside in memory. Your source program is read in a line at a time from the disk and the statements, except comments, are compiled into object code. When you execute a program from the monitor, only the run-time package and your object file need be in memory.

2.2.1 Creating a Program

There are two ways to create a source program for use in compiler mode: you can either use AlphaBASIC in interactive mode to type in the program, save that program on disk, and then exit BASIC; or, you can use one of the system text editors, EDIT or VUE. The usual way to create a program that is going to be compiled with COMPIL is to use VUE to create the .BAS file. VUE is a screen-oriented text editor that allows you to see your program on the terminal screen as you type it in. You can move the cursor around on the screen and change or delete text at the current cursor position.

2.2.1.1 Program Form - The form your program may take differs somewhat between compiler mode and interactive mode. If you create and save your source program in interactive mode, that program must, of course, contain line numbers. (Otherwise, BASIC would interpret each statement as a direct statement when you tried to type the program in.)

COMPIL, however, does not require that a program contain line numbers. In fact, COMPIL ignores any line numbers. That means that if you create your program using VUE, you do not need to include line numbers in that program. In addition, you may indent your program lines in any fashion you desire. By omitting line numbers, including line labels, and using indentation judiciously, you can give your source program a much more structured look than is usually possible with BASIC programs. (A "label" is a special name defined by you that identifies a location within a program.)

COMPIL also allows the use of continuation lines within a source program. Specify a continuation line by making an ampersand (&) the last character on that line. For example:

```
IF Answer = RIGHT'NUMBER THEN &
        PRINT "Very Good!" &
    ELSE &
        PRINT "Try again."
```

If you use LOAD in interactive mode to load a program that uses continuation lines, BASIC concatenates contiguous continuation lines into one line. (The maximum line length, including any continuation lines, tabs, or blanks, is 500 characters.) Then, if you save that program back out to the disk, any continuation lines are gone.

Below is a small example of a valid program that uses continuation lines, indentation, and labels, and has no line numbers:

```
        ! Program to print name in reverse.

INIT:   STRSIZ 20

STATT:  INPUT LINE "Enter your name: ",NAME$
        IF LEN(NAME$) = 0 THEN &
            GOTO START

LOOP:   COUNTER = LEN(NAME$)
        FOR I = 1 TO COUNTER
            PRINT NAME$[COUNTER;1]
            COUNTER = COUNTER - 1
        NEXT
        INPUT "Do you want instant replay? (Y or N)",QUERY$
        IF (QUERY$ = "Y") OR (QUERY$ = "y") THEN &
                GOTO LOOP &
            ELSE &
                PRINT "All done."
        END
```

NOTE: Since COMPIL ignores line numbers, it does no checking for duplicate line numbers or lines out of numeric sequence. If your program contains these kinds of errors, it will compile using COMPIL. However, if you use interactive mode and load the program in, BASIC (which requires line numbers) will be unable to handle the program correctly, and errors will result. (For example, in the case of duplicate line numbers, BASIC will merely take the last line in the file bearing the duplicate number.)

2.2.2 Compiling a Program

To compile a program in compiler mode, at AMOS command level enter COMPIL followed by the specification of the file you want to compile. You may supply a full file specification, including account and device specifications. (The default extension is .BAS. The default account and device are the ones you are logged into.) After you enter the file specification, type a RETURN. For example:

```
.COMPIL REVRSE (RET)
```

Now you see a number of statistics on your terminal as COMPIL compiles your program. A typical display might look something like this:

```
.COMPIL ACMSLS (RET)
Phase 1 - Initial work memory is 2310 bytes
Phase 2 - Adjust object file and process errors
Illegal MAP level - 350 MAP FILL S,16
Syntax error - 980 SLSMTD = SLSMTD;SSLAMT
Memory usage:
        Total work space - 4712 bytes
        Label symbol tree - 322 bytes
        Variable symbol tree - 1186 bytes
        Data statement pool - 0 bytes
        Variable indexing area - 274 bytes
        Compiler work stack - 140 bytes
        Excess available memory - 11918 bytes
```

Note that COMPIL tells you if any error exists within the source program when it processes your file (lines 4 and 5 of the display above). The "Excess available memory" message is useful for letting you know how close you are to running out of memory. If you do run out of memory during a compilation, you see the message: (Out of memory - compilation aborted), and COMPIL returns your terminal to AMOS command level.

When COMPIL has finished processing your file, it returns you to monitor level and writes the object program to the disk as a file bearing the name of your source program and a .RUN extension.

2.2.2.1 Compiler Options - Two compiler options are available for use with COMPIL: /O and /T. To choose an option, include the symbol "/" at the end of the file specification that you supply to COMPIL, followed by the appropriate option request code. For example:

```
.COMPIL NEWFIX.BAS/O (RET)
```

The compiler mode /O option is the same as the /O option for the interactive mode (see Section 2.1.3.1, above). The /O option code tells COMPIL to strip out any line number references in your compiled object code file. This makes your object code file smaller and makes the program run faster, but if an error message occurs, the message will not contain the number of the line at which the error occurred.

The compiler mode /T option is primarily for debugging purposes. It tells COMPIL to display each line of your source program as it scans that line. If a problem occurs during compilation, you can use the /T option to determine the line in which the problem occurs. You can also use /T to gauge the speed with which certain statements compile.

2.2.3 Running a Program

To run a program in compiler mode, at AMOS command level enter RUN followed by the name of the .RUN program you want to execute. Then type a RETURN. For example:

```
.RUN LOOP (RET)
```

You may supply a full file specification, including device and account specifications. The monitor looks for the run-time package, RUN.PRG, in memory; if it is not found in system or user memory, AMOS loads RUN into memory from the disk. RUN initializes memory and then looks for your program in memory; if it is not there, RUN loads the specified .RUN file from the disk. Now RUN executes your program. When RUN finishes executing the program, or if you type a Control-C to interrupt the program, RUN returns you to AMOS command level.

Note that the RUN command serves two different functions, depending on whether you are in compiler or interactive mode. In compiler mode, RUN is a monitor command used to execute a compiled BASIC program that has previously been saved on the disk or loaded into memory. The command:

```
.RUN PAYROL (RET)
```

will run PAYROL.RUN and then exit back to AMOS command level without ever entering BASIC. In interactive mode, the RUN command is a BASIC command that compiles and executes the current source program that you are editing and testing; when it finishes, you are still in BASIC.

NOTE: Be careful that you do not try to use the monitor command RUN on a file with a .BAS extension.

CHAPTER 3

GENERAL INFORMATION

This chapter gives general information about the form that your AlphaBASIC program may take. For example, we discuss multiple statement lines, EXPAND and NOEXPAND modes, program labels, and line numbers.

3.1 MULTIPLE STATEMENT LINES

The system supports multiple statement lines by using colons to separate the statements. For example:

```
10 FOR I=1 TO 10 : PRINT "THIS IS A LOOP" : NEXT I
```

The normal rules apply; for instance, a DATA statement cannot contain other statements on the same line and no other statements may follow a "comment" (designated by the REM or ! keywords). Direct statements may also be multiple statement lines.

You should always use spaces around the colons since BASIC will otherwise try to treat two commands (e.g., PRINT:PRINT) as a label and a single command. (The one situation where you do not have to use spaces around the colon that separates two statements is when you are in NOEXPAND mode. See Section 3.8 for information on EXPAND and NOEXPAND modes.)

3.2 CONTINUATION LINES

COMPIL allows the use of continuation lines within the source program. That is, statements may be continued on the next line by using the ampersand (&) symbol as the last character on the line. Since any statement line may be indented as you please in the compiler mode, considered use of continuation lines and indentation, plus optionally eliminating line numbers (as discussed in the next section) enables you to give your source program a much more structured look than allowed by more conventional BASICs or AlphaBASIC in the interactive mode. For example:

```
IF (TIME/60/60/CLKFRQ)*10000 > 120000 &
AND (TIME/60/60/CLKFRQ)*10000 < 130000 &
        THEN &
                PRINT "IT IS LUNCHTIME" &
        ELSE &
                PRINT "GO BACK TO WORK"
PRINT (TIME/60/60/CLKFRQ)*10000
```

The maximum size of any line, including blanks, tabs and any continuation lines, is 500 characters.

3.3 LINE NUMBERS

Program line numbers range from 1 to 65534. Programs used in interactive mode must contain line numbers. Programs to be compiled in compiler mode do not need to have line numbers, because COMPIL ignores line numbers. Therefore, if you create your program using VUE and are going to use COMPIL, you may omit the line numbers from the program. Unnumbered lines may enhance the structured look of your source program as shown in Section 3.2. NOTE: If you include line numbers, that does mean that if an error occurs, BASIC will be able to tell you which line the error occurred in.

3.4 COMMENTS (REM AND "!")

AlphaBASIC supports the ability to insert comments into the source program using two methods. The keyword "REM" may appear alone on a line followed by the comment, or may be inserted on the same line as a statement, to comment on the purpose of the statement. You may follow the REM (or "remarks") keyword with anything you want. For example:

```
70 REM ANYTHING YOU WISH TO SAY
100 PRINT A : REM VARIABLE A MEANS "ALLOWANCE"
```

Note that line 100 above is a legal multi-statement line; however, no statement may follow a REM statement on a line. When the program is compiled, everything in the line following the REM statement is ignored.

The comment symbol "!" is an abbreviation of the REM statement, and is used the same way. Like the REM statement, anything following the ! symbol on the line is ignored. For instance:

```
40 PRINT "TRY ANOTHER TIME"             !IF THEY MISS BETWEEN
50 GOTO AGAIN                           !ONE AND THREE TIMES.
```

3.5 INTERACTIVE MODE DIRECT STATEMENTS

AlphaBASIC immediately executes any line you enter if that line does not start with a line number. Such lines may be of two types: BASIC system commands and direct statements. A BASIC system command performs a system function (for example, the LIST command tells AlphaBASIC to display the program currently in memory). BASIC system commands may never be part of a program line. Direct statements, on the other hand, are normal program statements that may also appear within a program line. (For example, the PRINT statement tells AlphaBASIC to display a specified numeric or string value and may appear either as a direct statement or as part of a program). Some statements are not allowed as direct statements (for example, the GOSUB statement).

3.6 PROGRAM LABELS

AlphaBASIC allows the use of program labels to identify locations in a program. A program label is composed of one or more alphanumeric characters which are not separated by a space or other delimiter. The first character must be an upper case alphabetic character A-Z or a lower case alphabetic character a-z. Apostrophes may be used within labels in place of spaces for clarity, since apostrophes are not recognized as delimiters. A label, when used, must be the first item on a line and must be terminated by a colon (:). It is important to remember that you may not place a space between the label and its colon; to do so will cause BASIC to think that you have entered a multi-statement line rather than a labeled line. A label may be followed by a program statement on the same line, or it may be the only item on the line. The use of labels is similar to the use of line numbers with GOTO and GOSUB statements, and makes the program easier to document. Here are some examples of labels (using apostrophes):

```
10  START'PROGRAM: INPUT "Enter two numbers to get sum: ",A,B
20                 PRINT A; "+"; B; "=" A+B
30                 IF A+B <> 0 GOTO SUM'NOT'ZERO
40                 PRINT "Sum is zero"
50                 GOTO START'PROGRAM
60  SUM'NOT'ZERO:
70                 PRINT "Sum is not zero"
80                 GOTO START'PROGRAM
90                 END
```

where Start'Program: and Sum'Not'Zero: are labels. Note that a reference to a label, as seen in lines 30, 50 and 80, is neither the first item on a line nor is it terminated by a colon. The reference must be identical to the actual label in its case (upper and/or lower) and in the placement of apostrophes.

3.7 MEMORY ALLOCATION

The compiler system allocates memory dynamically as you edit your program, and also during its compilation and execution. Checks are made to tell you if you have run out of memory. If you do, you get an error message. If you run out of memory while COMPIL is compiling your program, compilation is aborted and you are returned to AMOS command level.

3.8 EXPAND AND NOEXPAND MODES

AlphaBASIC normally scans the source text of the program in EXPAND mode, which dictates that reserved words (verbs, functions, commands, etc.) be terminated by a space or a character that is illegal in variable names. This allows labels and variables to begin with reserved words. In other words, the variable name PRINTMASTER is not interpreted as PRINT MASTER in expanded mode. In the EXPAND mode, the statement FOR A=1 TO 10 cannot be written as FORA=1TO10. These are the two commands which you may apply to switch back and forth between the normal EXPAND mode and the NOEXPAND mode:

EXPAND	sets syntax scanner to expanded mode
NOEXPAND	sets syntax scanner to non-expanded mode

The default mode is EXPAND mode. Note that the object code which is generated as a result of a compilation is not affected in size, execution speed or anything else by the mode in which it is compiled.

NOEXPAND is usually used only when running programs written on other systems.

3.9 LOWER CASE CHARACTERS

AlphaBASIC supports lower case letters (a-z) and upper case letters (A-Z) in both the input source program and in the run-time execution of programs. The line editor built into the interactive system accepts and stores source input text in lower case or upper case characters. Lower case letters, when used within variable names and labels, are unique and separate from the corresponding upper case letters. In other words, the variable "a" is separate from the variable "A" and the variable "Tom" is separate from the variables "TOM" and "tom". Lower case letters may be used as the first character of a variable name or program label just as upper case letters may be.

Reserved words are treated somewhat differently from the above system. When a reserved word is expected, the syntax parser temporarily translates all lower case letters to upper case and then checks for a reserved word match. If the word is not a reserved word, the translation is not retained and the lower case letters are used for variable name matches. The following statements are all considered to be identical:

```
FOR A = 1 TO 100 STEP 2
For A = 1 To 100 Step 2
For A = 1 to 100 step 2
for A = 1 to 100 step 2
```

The entire string processing system supports lower case characters. That is, lower case letters used within string literals (inside quotes) are retained and printed as lower case. Lower case letters which are entered into string variables by means of the INPUT statement are also retained as lower case letters.

Note that all lower case characters are considered greater than any upper case character due to their position in the ASCII collating sequence. To assist in processing and comparing input which contains lower case letters, the UCS(X) function has been implemented. This function returns a string which is identical to the argument string (X), with all characters translated to upper case. The inverse function LCS(X) returns a string with all characters translated to lower case.

3.10 LIBRARY SEARCHING

Whenever a program (called via RUN or CHAIN) or a subroutine (called via XCALL) is requested, BASIC follows a specific pattern in looking for the requested module. If you specify an account, then BASIC uses the current default device and the specified account. If you do not specify an account, the search sequence is as follows (where [P,pn] designates the Project-programmer number that specifies your account):

```
System memory
User memory
Default disk:[User P,pn]
Default disk:[User P,0]
DSK0:[7,6]
```

Note that earlier versions of BASIC (pre-4.2) used a different search algorithm that was the reverse of the one outlined above.

CHAPTER 4

ALPHABASIC VARIABLES

4.1 VARIABLE NAMES

An AlphaBASIC variable name may contain any number of alphanumeric characters, and is not limited to a single letter or to a letter and a digit, as in most BASIC implementations. The first character of the name must be alphabetic (from A to Z and a to z), and the variable name may begin with any reserved word unless NOEXPAND mode is set (see Section 3.8, "EXPAND and NOEXPAND Modes"). (For a list of AlphaBASIC reserved words, see Appendix C, "Reserved Words.") Apostrophes may also be used in variable names to improve clarity. You may use both upper and lower case characters in your variable names. Note that although AlphaBASIC folds reserved words to upper case, it does not translate variable names (e.g., the variable name REC'SIZE is considered unique and separate from the variable name Rec'Size). (See Section 3.9 for a discussion of how AlphaBASIC handles upper and lower case characters.)

Normal (unmapped) variables are considered floating point variables unless their names are terminated by a dollar sign, in which case they are considered string variables. Variables defined via a MAP statement (called mapped variables) are defined by an explicit type code and therefore do not follow the standard convention of using a dollar sign for string variables; they may take on any kind of data format, regardless of the name terminator. (Mapped variables are a special form of AlphaBASIC variable that enable you to perform sophisticated data I/O. For information on mapped variables, see Chapter 8, "Memory Mapping System.")

Integer variables are specified by appending a percent sign to the variable name. (NOTE: The integer variable was added for compatibility reasons. However, AlphaBASIC does not perform integer arithmetic. Following a variable name with a % symbol is equivalent to using the integer function on that variable. For example, COUNTER% is the same as INT(COUNTER).)

Subscripting of array variables follows the standard conventions of other BASICs by enclosing the subscripts within parentheses.

The following are examples of legal variables:

```
A
A$
NUMBER
STRING$
MASTER'INVENTORY'RECORD
HEADER1
MOM'ALWAYS'LIKED'YOU'BEST
Z1234567
NEW'ARRAY(3,3)
```

4.2 NUMERIC VARIABLES

The normal mode of processing numeric variables (as opposed to string variables) is in 11-digit accuracy, which might be termed "single-and-one-half" precision compared to normally accepted standards. This is due to the hardware floating point instructions which are implemented in the Alpha Micro computer. Integer and binary variables are also considered numeric variables, but are always converted to floating point format prior to performing mathematical operations on them. All printing of numeric variables is done under normal BASIC format, with the significance being variable under user control from 1 to 11 digits. The SIGNIFICANCE statement is used to set up this value. (See Section 10.24, "SIGNIFICANCE.")

4.3 STRING VARIABLES

AlphaBASIC supports string variables in both single and array form. The memory that is allocated for each string variable is the number of bytes representing the maximum size that the string is allowed to expand to. Each string is variable in size within this maximum limit and a null byte is stored at the end of each string to indicate its current actual size if the string is shorter than the maximum. At the start of each compilation, the default size to be used for strings is 10 characters maximum. The STRSIZ statement may be used within the program to alter the value to be used for all new string variables which follow.

String variables may be concatenated by use of the plus sign between two strings. String variables may be assigned values by enclosing string literals in quotes. String functions such as LEFT$, RIGHT$, MID$, etc. are implemented to assist in manipulating portions of strings or substrings. In addition, a powerful substring modifying system may be used to operate on portions of strings within expressions. Chapter 7, "Substring Modifiers," is devoted to this unique option of AlphaBASIC.

Unformatted, mapped variables are also considered string variables when they are used in expressions or printed. (See Chapter 8, "Memory Mapping System," for information on mapped, unformatted variables.) (NOTE: Of course, an unformatted variable may contain non-string data. If this the case, then using the PRINT statement to display either that variable or an expression containing it will result in a very odd display, since the data is not in a printable form.)

4.4 ARRAY VARIABLES

Arrays may be designated by numeric or string variables and are allocated dynamically during execution when the DIM statement is encountered in the program. During execution, if no DIM statement has been encountered when the first reference to the array is made, a default array size of 10 elements for each subscript level is used. This means that all DIM statements must be executed in the program prior to any actual references to the array.

Arrays may be any number of levels deep but practicality dictates some reasonable limit of 20 or so. Each level is referenced by a subscript value starting with element 1 and extending to element N. Once an array has been dimensioned by a DIM statement, it may not be redimensioned by a subsequent DIM statement in the same program. At no time may the number of subscripts vary in any of the references to any element in the array. The number of subscripts in each element reference must also match the number of subscripts in the corresponding DIM statement which defined the array size. (See Section 10.4, "DIM," for more information on the DIM statement.)

CHAPTER 5

ALPHABASIC EXPRESSIONS

5.1 ARITHMETIC EXPRESSIONS

An expression can contain variables, constant values, operator symbols, functions, or any combination of the above. For example:

```
(1+(FIX(TOTAL'RECS*REC'SIZE)/512))
```

Parentheses are used to designate hierarchy within expression terms; the normal mathematical hierarchy prevails in the absence of parentheses. AlphaBASIC recognizes the following mathematical operators:

+	unary plus or addition	=	equal
-	unary minus or subtraction	<	less that
*	multiplication	>	greater that
/	division	<>	unequal
^	raise to power	><	unequal
**	raise to power	#	unequal
"	string literal	<=	less than or equal
NOT	logical NOT	=<	less than or equal
AND	logical AND	>=	greater than or equal
OR	logical OR	=>	greater that or equal
XOR	logical XOR	USING	expression formatting
EQV	logical equivalence		
MIN	minimum value		
MAX	maximum value		

5.2 OPERATOR PRECEDENCE

The precedence of operators determines the sequence in which mathematical operations are performed when evaluating an expression that does not have overriding parentheses to dictate hierarchies. AlphaBASIC uses the following operator precedence:

exponentiation
unary plus and minus
multiplication and division
addition and subtraction
relational operations (comparisons)
logical NOT
logical AND, OR, XOR, EQV, MIN, MAX
USING

NOTE: The USING operator allows you to format numeric or string data using a format string. For information on USING, see Chapter 13, "Formatting Output (PRINT USING and Extended Tabs)."

5.3 MODE INDEPENDENCE

Expressions may contain any mixture of variable types and constants in any arrangement. AlphaBASIC performs automatic string and numeric conversions as necessary, to ensure that the result is in the proper format. For example, if two strings are multiplied together they are first automatically converted to numeric format before the multiplication takes place. If the result is then to become a string, it is reconverted back to string format before the assignment is performed. In other words, the statement A$ = B$ * "345" is perfectly legal and will work correctly. This is a powerful feature which can save much programming effort when used correctly.

There is a seemingly ambiguous situation which arises from this mode independence. The plus symbol (+) is used both as an addition operator for numeric operations and as a concatenation operator for string operations. The value of 34+5 is equal to 39 but the value of "34"+"5" is equal to the string "345". The operation of the plus symbol is unambiguous in its operation but may take a little thought to figure out its exact usage in a given situation. A few examples might help.

If the first operand is numeric and the second is string, we convert the second to numeric form and perform addition.

34 + "5" equals 39

If the first operand is string and the second operand is numeric, we convert the second to string and perform concatenation.

"34" + 5 equals "345"

NOTE: The above two examples apply only when we are not "expecting" a particular type of variable or term. This generally occurs only in a PRINT expression such as PRINT "34" + 5. At other times, we are expecting a specific type of variable; the conversion of the first variable is then performed prior to inspecting the operator (plus sign). The operation of the plus sign is implicitly specified by the result of the first variable. Take the following example:

```
5 * "34" + 4
```

The multiplication operator (*) forces us to expect a numeric term to follow. The "34" string is therefore immediately converted to numeric 34 and multiplied by the 5. The plus sign then performs numeric addition instead of concatenation. The result is in numeric format and is converted to string format if its destination is a string.

The following are a few examples as they would be seen if you were to use them in an actual program:

```
10 A = 34 + 5
20 B = 34 + "5"
30 C = "34" + 5
40 D = "34" + "5"
50 A$ = 34 + 5
60 B$ = 34 + "5"
70 C$ = "34" + 5
80 D$ = "34" + "5"
90 PRINT A,B,C,D
100 PRINT A$,B$,C$,D$

READY
RUN (RET)
 39            39            39            39
 345           345           345           345
```

You can see that conversion is affected by the type of variable being used.

You might like to try a few examples of your own on your system to see what the results are. Remember, any potentially ambiguous expression may always be forced to one or the other type by use of the STR and VAL functions.

For more examples of mode independence, see the sample programs in Chapter 7, "Substring Modifiers."

CHAPTER 6

DATA FORMATS

This chapter discusses the various forms which your data may take. Note that if you do not use MAP statements to define your data, your variables may only take on floating point numeric values or string values. If you use MAP statements, however, you have a great deal more versatility in the format of your data, and can define binary and unformatted data as well. MAP statements also give you a way to define powerful hierarchial data structures that allow sophisticated data manipulation. (For information on using MAP statements, see Chapter 8, "Memory Mapping System." That chapter also discusses how BASIC assigns memory locations to data.)

6.1 FLOATING POINT FORMAT

All numeric variables are assigned floating point format unless specified otherwise in the program. The standard precision in use by the Alpha Micro system can be called "single-and-one-half," since it lies midway between what are known as single precision and double precision formats. The reason for this is that the hardware floating point instructions all work in this format. Floating point numbers occupy six bytes of storage and are in the format dictated by the hardware instructions. Of the 48 bits in use for each 6-byte variable, the high order bit is the sign of the mantissa. The next 8 bits represent the signed exponent in excess-128 notation, giving a range of approximately 2.9*10^-39 thru 1.7*10^38. The remaining 39 bits contain the mantissa, which is normalized with an implied high-order bit of one. This gives an effective 40-bit mantissa which results in an accuracy of 11 significant digits.

6.2 STRING FORMAT

The string format is used for the storage of alphanumeric text data. String variables require one byte of storage for each character and may be fixed in position using the memory mapping system. If a string is shorter than the maximum length, a null byte is stored following the last character to terminate the string.

NOTE: When AlphaBASIC compares a string of spaces and a null (empty) string, it sees them as equal. This is by design and demonstrates how AlphaBASIC compares strings. If two strings are of equal length, AlphaBASIC compares the strings on a character-to-character basis. If they are of different lengths, AlphaBASIC pads the shorter of the two with spaces until the strings are of equal length, and the comparison proceeds. For example, the string "PAST DUE" is equal to the string "PAST DUE ".

As you can see, using this algorithm causes a null string to be treated as a string of spaces during comparison. The proper way to check for a null string is to use the LEN$ function, rather than to see if it is equal to "". (If LEN$(string-variable) returns a zero, the string is null.)

6.3 BINARY FORMAT

2-BYTE Binary = 65535 DECIMAL

Binary variables are specified via MAP statements, and are similar to integer variables in other implementations of BASIC. A binary variable may be from 1 to 5 bytes in length and may be signed when all 5 bytes are specified. When less than 5 bytes are specified in a MAP statement as the length, the binary value may be loaded as a negative number, but it is always returned as a positive number of full magnitude. The upper bit (preloaded as the sign) takes on its specific value in the equivalent positive binary variable. For instance, a 1-byte binary may be loaded with positive numbers from 0 thru 255 (decimal), or negative numbers from -1 thru -128, but the negative numbers are returned as the positive values of 128 thru 255 respectively. Only 5-byte binary variables return the original sign and value when loaded with a negative number.

Binary variables may be used in expressions but they are slower than floating point variables because they are always converted first to floating point format before any mathematical operations are performed on them. Binary variables are useful in integer and logical (Boolean) operations or for storing values in small amounts of memory (floating point numbers always take 6 bytes of memory regardless of their values). All logical operations performed within expressions (AND, OR, XOR, NOT etc.) cause the values to be converted first to signed 5-byte binary format before the logical operation is performed. The value -1 represents a 40-bit mask of all ones. Any relational comparison between two expressions or variables returns a -1 if true, or a 0 if false.

6.4 INTEGER FORMAT

Integer variables and constants are specified by appending a percent sign (%) to the variable name, which is the standard convention in use by other BASICs. AlphaBASIC generates floating point variables and performs automatic integer truncation for all integer variables specified in this manner. Integer constants are generated as their equivalent floating point values and are included only for compatibility with existing program structures. Since integer variables are effectively floating point variables with an additional INT conversion performed, they are actually slower than regular floating point variables. This is the opposite of most other BASICs, which usually store integer variables as 2-byte signed values and perform special integer arithmetic on them. True integer variables may be defined by using the MAP statement and the "B" binary type code. See Section 8.3, "Type Code," for a description of the "B" type code.

6.5 UNFORMATTED

An unformatted numeric variable, specified via a MAP statement, defines a fixed size area of storage used to contain absolute unformatted data which may be in any of the above formats. This format is normally used in the mapping system to define contiguous storage which is subdivided into multiple variables of different formats. No conversion ever takes place when moving data to and from this format. Unformatted variables are treated as string variables when used in expressions.

CHAPTER 7

SUBSTRING MODIFIERS

AlphaBASIC supports a unique method of manipulating substrings. A substring is a portion of an existing string, and may be as small as a single character or as large as the entire string. Substring modifiers allow the substring to be defined in terms of character positions within the string, relative to either the left or right end of the string. The length of the substring is defined either in terms of its beginning and ending positions or in terms of its beginning position and its length. Substrings are defined by referencing the desired string followed by the substring modifier. The substring modifier is two numeric arguments enclosed within square brackets.

7.1 SUBSTRING MODIFIER FORMATS AND FEATURES

The substring modifier takes on two distinct formats:

```
[beginning-position,ending-position]
[beginning-position;substring-length]
```

The first format defines the substring in terms of its beginning and ending positions within the string and uses a comma to separate the two arguments. The second format defines the substring in terms of its beginning position within the string and its length, using a semicolon to separate the arguments. The second format basically performs the same function as the MID$ function.

The beginning and ending positions are defined as character positions within the string relative to either the left or right end. A positive value represents the character position relative to the left end of the string, with character position 1 representing the first (leftmost) position. A negative value represents the character position relative to the right end of the string, with character position -1 representing the last (rightmost) position. For example, assume the following string has the letters ABCDEF in it. The positions are defined in terms of positions 1 through 6 (left-relative) or positions -1 through -6 (right-relative).

```
 A  B  C  D  E  F    (6 characters within main string)
 1  2  3  4  5  6    (left-relative position values)
-6 -5 -4 -3 -2 -1    (right-relative position values)
```

Allowing negative values for right-relative positions provides the ability to pick out digits within a numeric string without having to calculate the total size of the string first and then working from the left. (Remember that the mode independence of AlphaBASIC allows you to apply string operations to numeric data.)

The substring-length argument used by the second format may also take on negative values for a more flexible format. Normally the length is a positive value which represents the number of characters counting the beginning position and incrementing the index to the right. A negative length causes the index to move to the left and returns a substring whose last character is the one marked by the beginning-position argument. Perhaps a few examples may clarify the use of substring modifiers. Assume the main string is A$ and it contains the above example of ABCDEF. The following substrings are returned:

```
A$[2,4]          equals    BCD
A$[2;4]          equals    BCDE
A$[3,3]          equals    C
A$[3;3]          equals    CDE
A$[-3,-2]        equals    DE
A$[3,-2]         equals    CDE
A$[3;-2]         equals    BC
A$[-3;-2]        equals    CD
A$[4;1]          equals    D
A$[4;-1]         equals    D
```

For example, A$[3,-2] tells AlphaBASIC to return the substring that begins at character position 3 (from the left) and ends with character position 2 (from the right); that is, to return all characters between C and E, inclusive. A$[3;-2], however, tells AlphaBASIC to return the substring that begins with character position 3 (from the left) and extends 2 character positions (toward the left); that is, to return all characters starting with C and working backward two positions to B, inclusive.

Any position values or length values which would cause the substring to overflow out of either end of the main string are truncated at the string end.

```
A$[3,10]         equals    CDEF
A$[-14,34]       equals    ABCDEF
```

The main string to which the substring modifier is applied is actually any expression and does not need to be a defined single string variable. For example:

```
Q$ = (A$+B$+C$)[2;10]
Q$ = ("ABLE"+A$+"QQ34")[4,10]
```

The mode independence feature allows substring modifiers to be applied to numeric expressions. (See Chapter 5, "AlphaBASIC expressions," for information on mode independence.) A string is returned, but if the destination is a numeric variable, another conversion is made on the substring to return a numeric value.

```
10 INPUT "Enter number: ",NUMBER : INPUT "Enter another: ",NUMBER2
20 SUM = NUMBER+NUMBER2
30 PRINT NUMBER;" + ";NUMBER2;" =";SUM
! Strip off rightmost digit and test it for divisibility.
40 IF SUM[-1;1] = 0 THEN PRINT "Divisible by 5 and 2"
50 IF SUM[-1;1] = 5 THEN PRINT "Divisible by 5"
```

Be sure you understand the concept of mode independence before you begin to use substring modifiers or you may get answers you don't expect. For example, lines 30 and 40 in the small program below return different answers, even though the subscripting is performed exactly the same in both cases. This is because the mode independence feature examines the data type of the destination variable before allowing any operations to be performed. When it scans line 30, BASIC knows that a string result is expected (because STRING$ is a string variable), and so reads the "+" symbol as a string concatenation operator. In line 40, BASIC knows that a numeric result is expected (because NUMERIC is a numeric variable), and so reads the "+" symbol as an addition operator.

```
10 VALUE1$="123"
20 VALUE2$="456"
30 STRING$ = (VALUE1$ + VALUE2$)[1;3]
40 NUMERIC = (VALUE1$ + VALUE2$)[1;3]
50 PRINT "NUMERIC =";NUMERIC,"STRING$ = ";STRING$
```

The program above prints:

```
NUMERIC = 579          STRING$ = 123
```

You may apply substring modifiers to subscripted variables or expressions containing subscripted variables. Be careful not to confuse substring modifiers with subscripted variables. For example:

```
A$(2,3)          designates a location in array A$
A$[2,3]          designates a substring of string A$
A$(2,3)[4,5]     designates a substring of the string
                 in location A$(2,3)
```

These are valid uses of the substring modifiers:

```
Q$ = A$(3,4)[2,5]
Q$ = (A$(1)+B$(3))[-5,3]
```

Substring modifiers return a string value. These may be used as part of string expressions. For example:

```
Q$ = A$ + B$[2;5] + (A$[2,2] + C$)[-5;-3]
```

You may apply substring modifiers to the left side of an assignment in order to alter a substring within a string variable. Only that portion of the string defined by the substring modifier is changed. The other characters in the string are not altered. This may not be applied to numeric variables (for example, A[3;2] = "23" is not valid).

If A$ contains ABCDEF:

```
A$[2,4] = "QRS"
```

causes A$ to contain the string AQRSEF.

CHAPTER 8

MEMORY MAPPING SYSTEM

One of the unique features of AlphaBASIC is that it allows you to specify the pattern in which variables are allocated in memory. The advantage to such a "memory mapping" system is that it gives you a way to define entire groups of related information (e.g., a logical record that contains fields of information about a customer). Each element of such a group does not have to be of the same size or data type. You can reference a single element of the group or the group as a whole. You will probably find memory mapping to be of most use when you are performing sophisticated disk I/O or when you are setting up a group of variables for transferring data between your program and an assembly language subroutine. (See Chapter 18, "Calling External Assembly Language Subroutines," for more information on assembly language subroutines.) Memory mapping is a powerful tool, somewhat akin to COBOL data description techniques or Pascal record definitions, that gives you a flexible and efficient way to transfer data in and out of programs.

This chapter discusses how the compiler usually allocates variables in memory, and how you can use the memory mapping features (via the MAP statement) to override the usual storage allocation. We also discuss one of the AlphaBASIC debugging features-- locating variables in memory while in interactive mode.

8.1 ALLOCATING VARIABLE STORAGE

During compilation, BASIC allocates memory storage for all defined variables in an area that is contiguous and predictable. The compiled program references all variables through an indexing scheme. Each variable in the working storage area has a representative item in the index area which contains all the information needed to define and locate that variable. The working storage area therefore contains only the pure variables themselves without any associated or intervening descriptive information. The index area is a separate entity, physically located before the working storage area in memory.

The allocation of the variable storage area for any program is predictable, and BASIC normally does it as it encounters each variable during compilation. Since this scheme is not easily followed by human beings, a different method must be derived which can override normal allocation processes if you wish to have the variables allocated in a predetermined manner. Also, the disk I/O system requires that variables used be in a specific relationship to each other when used in some of the more sophisticated programs. The MAP statement has been included in AlphaBASIC for the purpose of allocating variables in a specific manner. MAP statements are non-executable at run-time, but merely direct the compiler in the definition and allocation of the referenced variables.

Each MAP statement contains a unique variable name to which the statement applies. When the compiler encounters this statement, it allocates the next contiguous space in working storage as required and assigns it to that variable. The type of the variable is also specified in this statement and may be used to override the standard naming conventions of BASIC. All variables not defined in a MAP statement are then automatically assigned storage in sequence, for total compatibility with existing standards.

The mapping system has another distinct advantage for complex programs in its allocation of arrays. With the MAP statement, you have the ability to override the standard array allocation scheme and to force the allocation to proceed in a more flexible manner. Conventional BASIC array elements must all be of the same data type. AlphaBASIC allows several variables of different data types to be combined in a single contiguous array which can provide efficiency in the manipulation of associated data structures.

8.2 MAP STATEMENT FORMAT

The MAP statement has the following form:

```
MAPn variable-name{(dimensions)} {{{{type}, size}, value}, origin}
```

where MAPn gives the level of the MAP statement. The rest of the elements are optional, depending on the kind of variable you are defining. For example, if you are defining an array variable, you will include the optional "dimensions" in the MAP statement. "Type" identifies the data type of the variable; if omitted, the default is Unformatted. "Size" identifies the number of bytes the contents of the variable will use. If you omit "Size" the default is zero bytes for unformatted and string data, two bytes for binary data, and six bytes for floating point variables since such variables are always six bytes long.) "Value" is an optional initial value of the variable; the default is zero for numeric data and null for strings. "Origin" is an optional reference to a previously defined variable's location in memory which permits overlaying of variables in memory.

If you "skip" an element in the MAP statement (for example, you want to specify the "value" but don't want to specify the "size"), you must retain the comma indicating the missing element. For example:

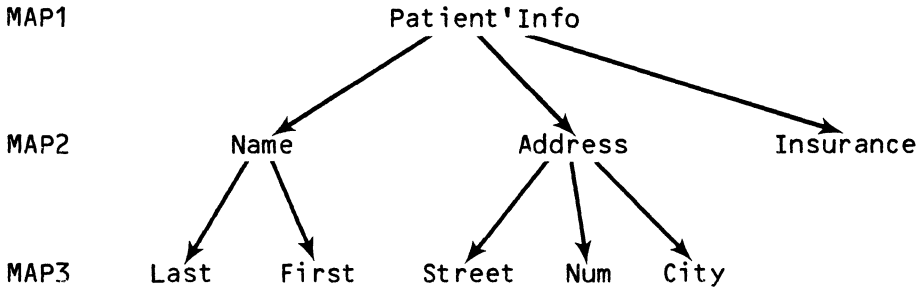
```
MAP 1 NEW'VARIABLE,F,,23
```

the MAP statement above defines the variable NEW'VARIABLE, assigns it the data type F (for floating point), does not assign it a size, and does assign it the initial value of 23. (Without the extra comma, BASIC would think that you were trying to assign a size of 23 bytes to NEW'VARIABLE-- an illegal operation for floating point variables.)

8.2.1 MAP Level

MAPn represents the level of the mapped variable. It must be within the range of MAP1 through MAP16. MAP statements are hierarchial in nature. For example, a variable mapped with a MAP1 statement may consist of several sub-variables mapped via a MAP2 statement. Each of those variables may in turn consist of several variables mapped via a MAP3 statement. And so on, up to MAP16. MAP16 represents the lowest-level (or innermost) variable; MAP1 represents the highest level variable. You do not need to map levels in strict numeric sequence-- for example, a MAP5 statement may follow a MAP3 statement without an intervening MAP4 statement.

You may reference variables at any level of the hierarchy. A graphic example may help to clarify this idea:



The diagram above shows three levels of variables that have been mapped with MAP1, MAP2, and MAP3 statements. You may reference the level 1 variable Patient'Info as a whole, or may reference one of the variables on levels 2 and 3 that represent sub-groups of the variable Patient'Info, such as Name, Address, or Street. When referencing any variable in the group, you automatically get the information in any of the variables below that variable in the hierarchy. For example, when you reference Name you get the information in the variables Last and First. As BASIC allocates the variables Name and Address, it automatically includes them (and their sub-variables) within the variable Patient'Info.

The MAP statements for the variable group above might look like this:

```
MAP1    PATIENT'INFO
  MAP2    NAME
    MAP3    LAST, S, 15
    MAP3    FIRST, S, 13
  MAP2    ADDRESS                 ! Patient address
    MAP3    STREET, S, 30
    MAP3    NUM, S, 10
    MAP3    CITY, S, 30
  MAP2    INSURANCE, B, 1         ! Set flag if has insurance
```

(NOTE: We will discuss each of the elements of a MAP statement in the sections below.)

To eliminate potential allocation problems, BASIC forces all MAP1 level variables to begin on an even memory address. This ensures that certain binary and floating point variables will begin on word boundaries for assembly language subroutine processing. The AM-100 instruction set performs most efficiently when word data is aligned on word boundaries.

8.2.2 Variable Name

The variable name is the name that your program uses to reference the mapped variable; it must follow the rules for AlphaBASIC variable names. However, since you may explicitly specify the type, you do not need to follow the normal conventions for identification such as requiring that a string variable name be followed by a dollar-sign.

If the variable name is followed by a set of subscripts within parentheses, the variable is assigned as an array with the dimensions specified by the subscripts, just as if a DIM statement had been used. For example, the statement MAP1 A,F assigns a single floating point variable called "A," but the statement MAP1 A(5,10),F assigns a floating point array with 50 elements in it (5 X 10), just as if the statement DIM A(5,10) had been executed. Note that since these mapped arrays are assigned memory at compile time and not at run-time, the subscripts must be decimal numbers instead of variables.

8.2.3 Type Code

The type code is a single character code which specifies the type of variable to be mapped into memory. The following variable types are implemented in AlphaBASIC:

X - unformatted absolute data variable
S - string variable
F - floating point variable
B - binary unsigned numeric variable

If no explicit type code is entered, BASIC assumes unformatted data (type X).

8.2.3.1 Unformatted Data - Unformatted data is absolute in memory. You usually only define an unformatted variable so that you can reference a group of other variables as one unit. The contents of unformatted data variables should only be moved to other unformatted data variables. For all practical purposes, unformatted data variables are treated like string variables except that they are terminated only by the explicit size of the variable.

8.2.3.2 String Data - String variables are terminated either by the explicit size of the variable or by a null byte (0) if the string is shorter than the allocated size. Moving a long string to a short one truncates all characters which do not fit into the new string variable. Moving a short string to a long one causes the remainder of the long variable to be filled with null (0) bytes so that the actual data size of the string is preserved for concatenation and printing purposes.

8.2.3.3 Floating Point Data - Each floating point number always takes up six bytes. The record number variable in a random mode OPEN statement must be floating point. The result variable of a LOOKUP statement must also be floating point.

0-255 in one byte Binary
0-65535 in two Byte Binary

8.2.3.4 Binary Data - Binary variables may range in size from 1 to 5 bytes, giving from 8 to 39 bits of binary unsigned numeric data or 40 bits of binary signed data. This is handy for the storage of small integer data in a single byte, such as flags, or for the storage of memory references as word values with a range of up to 65535 in two bytes. Since BASIC converts all binary variables to floating point format before performing any arithmetic calculations, binary arithmetic is actually slower than normal floating point arithmetic and is used mainly for compacting data into files and arrays where the floating point size of six bytes is inefficient. When conversions from floating point to binary are done, any data that does not fit within the defined size of the target variable is merely lost with no error message given. Where required, range checks are your responsibility as the programmer, before you make a floating point number move to a binary variable area. The best way to understand this is to play with a few examples in interactive mode.

Please take note that the use of binary numeric variables is not allowed in some instances. FOR-NEXT loops may not use a binary variable as the control variable, although they may be used in the expressions designating the initial and terminating values of the control variable, as well as in the STEP expression.

8.2.4 Size

The size parameter in the MAP statement is optional but, if it is used, it must be a decimal number specifying the number of bytes in the variable. If it is omitted, it defaults to 0 for unformatted and string types, 6 for floating point types, and 2 for binary types. The size parameter of floating point variables must be 6 or omitted.

8.2.5 Value

An initial value may be given to any mapped variable except an array variable by including any valid expression in the value parameter. This value may be a numeric constant, a string constant or a complete expression including variables. Remember, however, that the expression is resolved when the MAP statement is executed at run-time, and the current value of any variable within the value expression is the one used to calculate the assignment result. MAP statements may be executed more than once if you desire to reload the initial values.

Note that if you omit the size parameter (such as for floating point variables), but you use the value parameter, there must be an extra comma to indicate the missing size parameter:

```
MAP1 PI,F,,3.14159265359
MAP1 HOLIDAY,S,9,"CHRISTMAS"
```

The first example preloads the value 3.14159265359 into the floating point variable called PI. The second example preloads the letters CHRISTMAS into the string variable called HOLIDAY.

8.2.6 Origin

In some instances, it may be desirable to redefine records or array areas of different formats so that they occupy the same memory area. For instance, a file may contain several different record formats with the first byte of the record containing a type code for that record format. The origin parameter allows you to redefine the record area in the different formats to be expected. When the record is read into the area, the type code in the first byte can be used to execute the proper routine for the record type. Each different routine can access the record in a different format by the different variable names in that format. All record formats actually occupy the same area in memory. This feature directly parallels the REDEFINES verb in the COBOL language data division. Using the origin parameter can save large amounts of memory. For instance, suppose you have three very large variables of 256 bytes each that define logical records, and that you never use these variables at the same time. By defining the variables so that they occupy the same area of memory, your program only uses 256 bytes for the variables instead of 768 bytes.

Normally, a MAP statement causes allocation of memory to begin at the point where the last variable with the same level number left off. The origin parameter allows this to be modified so that allocation begins back at the base of some previously defined variable, and therefore overlays the same memory area. If the new variable is smaller than the previous one (or the exact same size), it is totally contained within the previous one. If it is larger than the previous one, it spills over into newly allocated memory or possibly into another variable area of the same level depending on whether there were more variables following it. (Play with this one awhile to get the hang of it).

The origin parameter must be the last parameter on the line. It takes this form: an @ symbol followed by the name of the previously mapped variable whose area you wish to overlay. (This variable must be on the same level as the variable you are presently allocating.) If size and value parameters are not included in this statement, you may omit them with no dummy commas. For example:

```
10 MAP1  CUSTOMER'ID
20  MAP2  NAME, S, 13
30  MAP2  ID'NUM, F
40  MAP2  SEX, B, 1
50 MAP1  PRODUCT'INVENTORY, @CUSTOMER'ID
60  MAP2  BRAND, S, 13
70  MAP2  PARTNO, F
80  MAP2  RESALE, B, 1
```

The MAP statements above allocate the variable CUSTOMER'ID which takes up a total of 20 bytes. Then it allocates the variable PRODUCT'INVENTORY (also taking up 20 bytes), and specifies via the @CUSTOMER'ID origin parameter that PRODUCT'INVENTORY will occupy the same space in memory as CUSTOMER'ID.

The following statements define three areas which all occupy the same 48-byte memory area, but which may be referenced in three different ways:

```
100 MAP1 ARRAY
110    MAP2 INDEX(8),F
200 MAP1 ADDRESS,@ARRAY
210    MAP2 STREET,S,24
220    MAP2 CITY,S,14
230    MAP2 STATE,S,4
300 MAP1 DOUBLE'ARRAY,@ARRAY
310    MAP2 UNIT(6)
320       MAP3 CODE,B,2
330       MAP3 RESULT,F
```

Statements 100-110 define an array with 8 floating point elements: a total of 48 bytes in memory. Statements 200-230 define an area with three string variables in it, for a total of 42 bytes. Normally, this area would follow the 48-byte ARRAY area in memory, but the origin parameter in statement 200 causes it to overlay the first 42 bytes of the ARRAY area instead. Statements 300-330 define another array area of a different format with 6 elements, each element being composed of one 2-byte binary variable (CODE) and one floating point variable (RESULT). The origin parameter in statement 300 also causes this area to overlay the ARRAY area exactly.

Caution: The above scheme allows variables to be referenced in a different format than when they were entered into memory. If you load the 8 elements INDEX(1) through INDEX(8) with floating point values, and then reference the variable STREET as a string, you get the first four floating point variables, INDEX(1) through INDEX(4), which look very strange in string format!

Below is a practical example of the use of the origin parameter. The program below translates the binary data stored in the system DATE location into floating point form.

```
10 ! The system stores the date in binary form; the small program
15 ! below translates the binary date into floating point form.  It
20 ! also allows you to set the system date from within BASIC.
25 MAP1 BINDATE,B,4
30 MAP1 FILLDATE,@BINDATE
35  MAP2 MONTH,B,1
40  MAP2 DAY,B,1
45  MAP2 YEAR,B,1
50 BINDATE = DATE
55 PRINT "Month:";MONTH,"Day:";DAY,"Year:";YEAR
60 INPUT "Enter Month, Day, Year: ",MONTH,DAY,YEAR
65 DATE=BINDATE
70 PRINT "Month:";MONTH,"Day:";DAY,"Year:";YEAR
```

For example, if the system date is set to January 10, 1982, a sample run of the program above might look like:

```
Month: 1        Day: 10         Year: 82
Enter Month, Day, Year: 4,21,52 (RET)
Month: 4        Day: 21         Year: 52
```

8.3 EXAMPLES

The following two statements produce identical arrays:

```
100 MAP1 A1(10),F
110 DIM A1(10)
```

Both statements produce arrays containing ten floating point variables, referenced as A1(1) thru A1(10). Statement 100, however, defines its

placement in memory in relation to other mapped variables. Similarly, the two statements at 300 and 310 produce the same two-dimensional array as the statement at 320:

```
300 MAP1 BX(5)
310    MAP2 B1(20),F
320 DIM B1(5,20)
```

Inspect the following statements:

```
400 MAP1 CX(10)
410    MAP2 C1,F
420    MAP2 D1,F
430 DIM C1(10)
440 DIM D1(10)
```

The statements at 430 and 440 produce two arrays, each with ten variables. The statements at 400, 410 and 420 produce one array with twenty variables in it. The variables are still referenced as C1(1) thru C1(10) and D1(1) thru D1(10), but their placement in memory is quite different. The C1 variables are interlaced with the D1 variables, giving C1(1), D1(1), C1(2), D1(2), C1(3), D1(3) .... C1(10), D1(10). There are also ten unformatted variables CX(1) thru CX(10), which each contain the respective pairs of C1-D1 variables in tandem. Referencing one of these CX variables references a 12-byte, unformatted item composed of the C1-D1 pair of the same subscript. This type of formatting would be useful in sophisticated techniques only.

The following defines a more complex area:

```
100 MAP1 ARRAY1
110    MAP2 UNITX(5)
120       MAP3 SIZA,B,2
130       MAP3 SIZB,B,2
140       MAP3 NTOT,F
150       MAP3 FLAG(10),B,1
160       MAP3 CNAME,S,20
170    MAP2 TOTAL,F
180 MAP1 THING,F
190 MAP1 WORK1,X,40
```

The area that is allocated by the above statements requires a total of 252 bytes of contiguous memory storage. Three levels are represented in various formats. Statement 100 defines a level 1 unformatted area called ARRAY1, which is subdivided into two level 2 items. Statement 110 defines the first of these, which is an area called UNITX. The optional dimension indicates that five of these identical areas exist, which must be referenced in the program by the subscripted variable names UNITX(1) through UNITX(5). Each one of these areas is then further subdivided into five level 3 items (statements 120-160). Since the level 2 is subscripted because it occurs 5 times, so must each of the level 3 items be subscripted. There are 5 variables named SIZA(1) thru SIZA(5) occurring once in each of the respective variables UNITX(1) thru UNITX(5). The same holds true for the

variables SIZB, NTOT, and CNAME. Statement 150, however, creates a special case since it contains a dimension also. Normally this would create an area of 10 sequential bytes referenced as FLAG(1) thru FLAG(10). In our example, however, this 10-byte area occurs once in each of the higher level areas of UNITX(1) thru UNITX(5). This, then, implicitly defines a double-subscripted variable ranging from FLAG(1,1) thru FLAG(5,10). Statement 170 causes the allocation to return to level 2 where one floating point variable is allocated.

The total storage requirement for the level 1 variable ARRAY1 comes out to 206 bytes as follows: 40 bytes for each of the five areas UNITX(1) thru UNITX(5), plus 6 bytes for the one variable TOTAL. Notice that since TOTAL starts a new level 2, it does not occur 5 times, as do the level 3 items which comprise UNITX(1) thru UNITX(5).

Following the above group in memory come two more variables defined in statements 180 and 190. THING is a normal floating point variable which occupies 6 bytes, and WORK1 is an unformatted area whose size is 40 bytes. Note that since WORK1 was not subdivided into one or more level 2 items, a size clause was required to explicitly define its storage requirements.

Note also that the variable UNITX(1) refers to the 40-byte item comprised of the variables (in order): SIZA(1); SIZB(1); NTOT(1); FLAG(1,1) thru FLAG(1,10); and CNAME(1). Moving the variable UNITX(1) to another area, such as WORK1, transfers the entire 40-bytes with no conversions of any data.

You may often use MAP statements to define groups of information that will be transferred in and out of disk files. For example, take a look at the MAP statements below that define a logical record. Our program probably uses a file that contains a large number of logical records in this format, each record containing information about a single check. In effect, MAP statements give us a way to form a template in memory into which we can read information from the file and transfer information from the program to the file. This allows us to quickly and efficiently read in an entire group of information whose elements may be of different types and sizes, and to access information in that group flexibly and simply. For example:

```
!  REM Program to Process Checks.
20
30  MAP1   CHECK'INFO            ! Define logical record.
40   MAP2      CHECK'NUMBER, F
50   MAP2      THE'DATE, S, 6
60   MAP2      AMOUNT, F
70   MAP2      TAX'DEDUCTABLE, B, 1
80   MAP2      PAYEE, S, 20
90   MAP2      CATEGORY, S, 20
100  MAP2      BANK'ACCOUNTS(3)
110   MAP3       SAVINGS, S, 20
120   MAP3       CHECKING, S, 20
130   MAP3       TERM, S, 20

! Define file that contains info about checking account balance.

140 MAP2       ACCOUNT'BALANCE, S, 22, "DSK1:BALANC.DAT[200,1]"
```

Once these MAP statements have been executed, we can access the group of variables as a whole by specifying CHECK'INFO, or we can access specific sub-fields in the record (for example, BANK'ACCOUNT or CHECKING).

8.4 USING THE MAP STATEMENTS

MAP statements may be used as direct statements in interactive mode as a learning tool to see how the variables are allocated. They are not designed to be practical in the interactive mode, however, and are best used by putting them into a program file and compiling the program. In the interactive mode, if an error occurs in the syntax of the statement, the variable will already have been added to the tree in memory. The main advantage to testing MAP statements in interactive mode is that BASIC checks the MAP statement syntax as you enter the statement, thus giving you immediate feedback if any errors occur.

MAP statements must come at the beginning of the program, before any references to the variables being mapped. If a reference is made to the variable before it is mapped (such as LET A = 5.8), the variable is assigned by the normal variable allocation routines and the MAP statement then gives an error, since the variable is already defined. As a convenience, all MAP1 statements force allocation to the next even byte boundary so that binary word data can be assigned properly.

8.5 LOCATING VARIABLES DURING DEBUGGING

Since the mapping scheme is fairly complex to understand fully, a command has been implemented which assists you in locating the mapped variables and in understanding the allocation techniques used by the AlphaBASIC memory mapping system. It is valid only as a BASIC system command and has no

meaning if used within a program text. The command has the general format of an atsign (@) followed by a variable name. The system searches for the requested variable and prints out all parameters about the variable for you on the terminal. (This may actually be two definitions, since the variable "A" may actually be two different variables; one would be a single floating point number and the other would be a subscripted array.) The information returned about the variable is: the type of variable (string, binary, etc.); the dimensions of the array if the variable is indeed an array; the size of the variable in bytes; and the offset to the variable from the base of the memory area which is used to allocate all variables. If you enter a reserved word (such as @PRINT) the system tells you that the name is a reserved word.

The general format of the definition line which is returned by the system is:

memory-type variable-type {dimensions}, size n, location

(For actual examples of the definition line, see Section 8.6.1, "Examples," below.) Memory-type means the method of memory allocation used when defining the variable. The memory-type may be MAPn (where n is a number from 1 to 16), FIXED or DYNAMIC variables. FIXED variables are not defined by a MAP statement and are allocated automatically when the compiler finds references to them in the program. (This is the normal method used by other BASIC versions to allocate variables.) DYNAMIC variable arrays are allocated by a DIM statement or by a default reference to a subscripted variable. Variables defined in a MAP statement are MAP1 through MAP16 variables.

Variable-type is the type of the variable and may be UNFORMATTED, STRING, FLOATING POINT, or BINARY.

If the variable is an array, the dimensions are listed after the variable type code in the format ARRAY (n,n,n), where n,n,n are the values of the subscripts in use by the array. If the array is dynamic and has not been allocated yet, the subscript values are replaced by the letter "X" to indicate that they are not known at this point. Remember that any variable defined in a MAP statement which is in a lower level relative to another variable inherits all subscripts from that higher level variable.

The size of the variables are given in decimal bytes. In the case of arrays, the size represents the size of each single element within the array.

The location of the variable is a little tricky to explain, since it is actually an offset to the base of a storage area that is set aside for the allocation of user variables. As each new variable or array is allocated, it is assigned a location which is relative to the base of this storage area. The location information given here is an example to help you understand the relative placement of the variables in the mapping system, and does not represent the actual memory locations which they occupy. There are two distinct areas in use for variables, and thus the offsets of the variables are to one of these two areas. All FIXED and MAP1 through MAP16

variables are allocated in the fixed storage area, while all DYNAMIC arrays are allocated in the dynamic array storage area. As dynamic arrays are dimensioned, their positions may shift relative to one another and relative to the dynamic storage area base. Variables in the fixed storage area never change position relative to each other or to the storage area base.

Array location information that is given is only pertinent to the base of the array itself, which is the location of the first element within the array. The actual range of locations used by the array may or may not be contiguous in memory depending on whether overlapped dimensioning techniques are being used in the MAP statements. Simple (non-array) variables are defined as a location range which tells exactly where the entire variable lies within the storage area.

Keep in mind that this "@" command is to assist you in following the allocation of variables, particularly in more complex mapping schemes. A few minutes at the terminal with direct MAP statements followed by "@" commands will help you see how the mapping scheme works.

8.5.1 Examples

Given the sample MAP statements below:

```
10 MAP1     CUSTOMER'ID
20  MAP2      NAME
30   MAP3      FIRST, S, 15
40   MAP3      LAST, S, 15
50  MAP2      ADDRESS
60   MAP3      STREET, S, 15
70   MAP3      CITY, S, 10
80   MAP3      STATE, S, 2
90  MAP2      PHONE
100  MAP3      HOME, B, 3
110  MAP3      BUSINESS, B, 3
120 MAP2      TRANSACTIONS(12)
130  MAP3      BALANCE, F
140  MAP3      CREDIT, F
150  MAP3      YTD, F
```

Here are the results of using the @ command in interactive mode to determine the locations of several of the variables above:

```
READY

@CUSTOMER'ID (RET)
MAP1 Unformatted, size 279, located at 0-278

@TRANSACTIONS (RET)
MAP2 Unformatted Array (12), size 18, base located at 63

@CITY (RET)
MAP3 String, size 10, located at 45-54

@HOME (RET)
MAP3 Binary, size 3, located at 57-59

@CREDIT (RET)
MAP3 Floating point Array (12), size 6, base located at 69
```

We can also use the @ command to locate unmapped variables. For example:

```
READY

DIM A(2,3,4) (RET)
@A (RET)
Dynamic Floating point Array (2,3,4), size 6, base located at 1032

A=15 (RET)
@A (RET)
Fixed Floating point, size 6, located at 72-77
Dynamic Floating point Array (2,3,4), size 6, base located at 1032
```

Note that we allocated two different variables: a fixed floating point variable, A, and a dynamic floating point Array variable, A(2,3,4).

CHAPTER 9

INTERACTIVE COMMAND SUMMARY

Whenever AlphaBASIC interactive mode is not either compiling or executing a program, it is in interactive command mode; that means it is waiting for a command from your terminal to initiate some action. The action taken depends on the type of input you enter, which falls into one of the following main categories:

1. Statements. Program statements are either contained within a BASIC program or are used for immediate compilation and execution at the interactive command level. For immediate compilation and execution of a statement, enter the statement without a line number. Statements entered following line numbers (any integer between 1 and 65534, inclusive) are used to build a source program in memory on a single line basis. BASIC automatically adds the single lines to the source program in the numeric order of their line numbers. Entering a line number alone and then a RETURN deletes the line associated with that line number from the source program.

2. Interactive system commands. Commands result in controlled actions by BASIC which can affect the source program in memory, files on the disk, and the system itself. Commands are never entered into the program as statements. If you attempt to do so, AlphaBASIC responds with an error message.

Statements are covered in detail in Chapter 10 of this manual. The remainder of this chapter details the available interactive commands, the corresponding actions performed, and shows examples as you would actually see them. Most of the interactive commands are entered after the prompt READY. We distinguish the commands you may enter by the (RET) symbol, which means "type a RETURN."

9.1 BREAK

This is a debugging feature not usually found in other versions of BASIC. It takes the form:

BREAK {{-}{line#1{,{-}line#2,...{-}{line#N}}

BREAK allows you to set breakpoints from the interactive mode on one or more lines in the program in memory, prior to running the program. During execution, when BASIC encounters a line that has a breakpoint set on it, BASIC suspends program execution and prints the message "Break at line nnnn". The system then returns to interactive command mode to allow you to inspect or change variable values. This suspension of execution occurs before the line that has the breakpoint set on it is executed. There is no limit to the number of breakpoints that may be set in one program. There is no additional overhead paid in execution speed when breakpoints are set. Breakpoints are cleared from within the interactive mode by typing a minus sign in front of the line number, or by recompiling the program (which always clears all breakpoints). If you type BREAK and do not follow it with a line number, BASIC lists all current breakpoints on your terminal.

```
BREAK (RET)                 or      BREAK (RET)
No breakpoints set                  30
```

The following are various forms of the BREAK command:

BREAK	Lists all currently set breakpoints, if any
BREAK 120	Sets a breakpoint at line 120
BREAK -120	Clears the breakpoint at line 120
BREAK 120,130,40,500	Sets breakpoints at lines 120,130,40, and 500
BREAK -50,60	Clears the breakpoint at 50 and sets one at 60

Once a breakpoint has been reached, you may optionally continue the execution of the program by either a CONT command or a single-step command. (For information on the single-step debugging feature, see Section 9.12, "Single-Step (Linefeed)." You may start the program over again by using the RUN command; it will once more break at the first breakpoint set. In any case, the breakpoints remain set after they have been reached until they are explicitly cleared by a BREAK -nn command, are generally cleared by compiling the program, or you leave BASIC.

9.2 BYE

BYE says goodbye to the BASIC interactive mode and returns your terminal to the AMOS command level. You then see the AMOS prompt (.). Remember that any program left in memory is lost forever, so you may want to save it first using the SAVE command. This is the format of BYE:

```
BYE (RET)
.
```

9.3 COMPILE

When using COMPILE in the interactive mode, do not specify a source program. BASIC compiles the current source program in memory. The object code is built up in another area of memory. The compiled program is not executed; instead, control is returned to the interactive command mode and you see the READY prompt. Compilation sets all variables to zero and deletes all variables that may have been generated as a result of direct statements.

```
READY
COMPILE (RET)
Compile time was 7.07 seconds

READY
```

If no program is in memory, you get an error message and a prompt.

```
READY
COMPILE (RET)
No source program in text buffer

READY
```

9.4 CONT

CONT, for "continue," causes a suspended program to continue execution from the point at which it was suspended. You may suspend a program by using a BREAK command previous to program execution or by using a STOP statement within the program. You may not continue a program after it has finished. The following is an example of CONT after a STOP statement suspended a program:

```
Program stop in line 700

READY
CONT (RET)
(The program continues by next executing the
first line numbered higher than line 700.)
```

CONT also continues a program which you have partially executed using the single-step feature.

9.5 CONTROL-C

Pressing the Control and C keys simultaneously interrupts a running program and returns you to interactive command mode. Depending on the operation being performed, the Control-C symbol (^C) may be displayed on the terminal screen. The line number of the source program which was interrupted is displayed via the message "Operator interrupt in line nn." The program may be restarted from the beginning by the RUN command.

```
[Type a Control-C]                    or    [Type a Control-C] ^C [displays]
Operator interrupt in line 700              Operator interrupt in line 700

READY
```

NOTE: In the compiler mode, while running a program, typing a Control-C causes almost the same message to appear. The difference is that it always displays the Control-C (^C) and gives the filename of the program as well. It then returns you to AMOS command level. For example:

```
^C
Operator interrupt in line 10 of NUM.RUN
.
```

In the interactive mode, you may restart a program from the beginning following a Control-C by using the RUN command or single-step (linefeed) command.

9.6 DELETE

The DELETE command is used to delete groups of source lines from the program text. It takes the form:

```
DELETE line#1{,line#2}
```

If the command is followed by a single line number, only that line is deleted. If the command is followed by two line numbers separated by a comma, all lines of text which fall between and including the two line numbers are deleted from the program. (NOTE: Although you usually separate the two line numbers with a comma, you can also use a dash, space, or other non-numeric character.) Here is an example listing before and after a DELETE:

```
LIST (RET)
10 FOR I = 1 TO 10
20      PRINT TAB(I,5)"ONE"
30      PRINT TAB(I,5)"TWO"
40      PRINT TAB(I,5)"SIX"
50      PRINT TAB(I,5)"TEN"
60 NEXT I

READY
DELETE 20,40 (RET)

READY
LIST (RET)
10 FOR I = 1 TO 10
50      PRINT TAB(I,5)"TEN"
60 NEXT I

READY
```

Remember, you can say: "DELETE 20 40" or "DELETE 20-40", too.

9.7 LIST

The LIST command takes the form:

```
LIST {line#1{,line#2}}
```

The source program (if one is loaded into memory) lines are listed in numeric sequence on your terminal. If no line numbers follow the LIST command, BASIC lists the entire program. You may abort the listing by entering Control-C, which returns you to interactive command mode. If one line number follows the LIST command, only the single line following that line number is listed. If the command is followed by two line numbers separated by a comma, space or other non-numeric character, only the indicated lines and the lines between them are listed. Some examples:

```
READY
LIST (RET)
10 X=1
20 ? "POWERS OF TWO:"
30 FOR A=0 TO 10
40      ?"2^";A;"=";X
50 X=X*2:NEXT A

READY
```

```
READY
LIST 10 (RET)
10 X=1

READY
```

```
READY
LIST 10,30 (RET)
10 X=1
20 ? "POWERS OF TWO:"
30 FOR A=0 TO 10

READY
```

(NOTE: Remember that the "?" symbol is an abbreviation for the PRINT keyword.)

9.8 LOAD

The LOAD command copies the specified BASIC program into memory from the disk so that you can edit or execute it. You must give a valid AMOS file specification after the LOAD command. If you do not supply a file extension, BASIC uses the default extension of .BAS. If you do not supply an account and device specification, BASIC assumes the account and device you are logged into. For example:

```
READY
LOAD PAYROL (RET)

READY
```

The command above tells BASIC to search for and load into memory the disk file PAYROL.BAS that exists in the account and device you are logged into.

If BASIC can't find the file you want to load, it displays an error message. For example, if you try to load in the non-existing file LSTSQR.BAS[100,1], you see:

```
?Cannot OPEN LSTSQR.BAS[100,1] - file not found
```

The LOAD command does not clear the text buffer before it loads the requested file, and therefore may be used to concatenate or merge several programs or subroutines together to be saved as a single program. The separate routines must not duplicate line numbers in the other routines that they are to be merged with or else the new line numbers will overlay the old ones just as if the file had been edited in from your terminal. IMPORTANT NOTE: You should always use the NEW command prior to any LOAD command if you desire to ensure that the text buffer is clear.

Two examples of LOAD:

```
READY                          READY
LOAD PWRS2 (RET)               LOAD DSK2:PWRS2.BAS[50,1] (RET)

READY                          READY
```

9.9 NEW

This command clears out all current source code, object code, user symbols and variables. It initializes the compiler to accept new source program statements or direct statements:

```
READY
NEW (RET)

READY
```

If you do not use the NEW command before loading in a new program, any lines in the new program with the same line numbers as other program lines already in memory will overlay and replace the old lines; you will thus merge the old and new programs.

9.1 RUN

This is the usual command to use to initiate the execution of the program which is in memory. BASIC first checks to see if the program has been compiled since the last editing change to the source code. If it has not, BASIC automatically compiles the source program to ensure that the object code is up to date. RUN resets all variables to zero (and strings to null) and it then executes the compiled object code. Execution may be interrupted at any time by typing a Control-C on your terminal.

```
READY
RUN (RET)
(The program currently in memory
begins at the lowest line number.)
```

9.2 SAVE

The SAVE command saves the entire source program on the disk in the specified account and device. You must enter the name of the program (1-6 characters) following the SAVE command. The program is saved in ASCII format. The default extension is .BAS, and the default account and device are the device and account you are logged into. The program may be displayed or edited with the normal text editors outside of AlphaBASIC. If a previous version of the program (same name) already exists on the disk in the account you are writing the file to, that program is first deleted before the new program is saved. BASIC does not automatically create a backup file. The program name may be a full system file specification.

```
SAVE PAYROL(RET)          SAVE DSK2:PAYROL.BAS[50,1](RET)

READY                     READY
```

You may also use the SAVE command to save the compiled object program on disk for later running without recompilation. To save the object program, enter the program name followed by the explicit extension .RUN. If you have changed the program since the last time it was compiled, BASIC now automatically compiles the program for you. Then the object program is saved on the disk:

```
SAVE PAYROL.RUN (RET)
(Saves the object program on the disk as PAYROL.RUN.)

READY
```

In the interests of security, BASIC will not let you save a program that is in an account that is not within the same project as the account you are logged into. For example, if you are logged into DSK2:[100,2] and want to save a program in DSK2:[340,1], you see:

```
READY
SAVE NEWPRG[340,1](RET)

?Cannot OPEN NEWPRG.BAS[340,1] - Protection violation
```

9.3 SINGLE-STEP (LINEFEED)

The single-step function is a feature not found in many versions of BASIC, and is very useful in debugging programs and in teaching the principles of BASIC programming to newcomers. To use the single-step command, type a linefeed. (That is, press the terminal key labeled LF, LINEFEED, or ↓ .) The single-step function causes the current line in the program to be listed on your terminal and then executed. Any output generated by the execution of a PRINT statement then follows on the next line. After the line has been executed, the execution pointer is advanced to the next line and control returns to you in the interactive command mode. Successive single-step commands may be used to follow the program through its paces. Single-step is legal at the beginning of the program, after program STOP statements, breakpoint interrupts, and other functions that suspend program execution. After partially single-stepping through a program, you may execute the remainder of it normally by using the CONT command. Also, you may start over at the beginning and execute it normally by using the RUN command. If you try to single-step past the end of the program, you see:

```
***End of Program***
```

and the next linefeed executes the first program statement again.

If you single-step a statement that asks for input from the terminal, enter the input followed by a RETURN; then you may proceed to the next statement by typing another linefeed.

Remember that the single-step function is performed by hitting the linefeed key and not by actually entering the words "single-step."

The following is a demonstration of the single-step process for a small program as you would see it on your CRT. The symbol ↓ represents the linefeed key which you press to see the next statement and the results of it. (You do not actually see an echo of the linefeed key on the CRT.) Note that line 30 is a multi-statement line. When single-stepping, all statements on a line are executed. BASIC returns control to the interactive mode at the beginning of each line.

```
LIST (RET)
10 PRINT "This is a demonstration of single-step"
20 FOR I = 1 TO 3
30     PRINT 10*I : PRINT 10*I^I : PRINT 10^I*I
40 NEXT I

READY
↓
COMPILING
Compile time was 0.20 seconds
10 PRINT "This is a demonstration of single-step"
This is a demonstration of single-step
↓
20 FOR I = 1 TO 3
↓
30     PRINT 10*I : PRINT 10*I^I : PRINT 10^I*I
 10
 10
 10
↓
40 NEXT I
↓
30     PRINT 10*I : PRINT 10*I^I : PRINT 10^I*I
 20
 40
 200
↓
40 NEXT I
↓
30     PRINT 10*I : PRINT 10*I^I : PRINT 10^I*I
 30
 270
 3000
↓
40 NEXT I

*** End of Program ***
```

CHAPTER 10

PROGRAM STATEMENTS

The source program contains statements which are executed in sequence, one at a time, as BASIC encounters them. Each of these statements normally starts with a verb followed by optional variables or statement modifiers. Many of these statements can also be used in the interactive mode as direct statements. This chapter lists all the program statements and gives some examples for clarity.

10.1 ALLOCATE

The format is:

```
ALLOCATE filespec,number-of-blocks
```

This statement allocates a random access file on the disk. It is discussed in detail in Chapter 15, "AlphaBASIC File I/O System."

10.2 CHAIN

The format is:

```
CHAIN filespec
```

where the filespec may take the forms:

```
{Devn:}BASIC-program-name{.RUN}{[p,pn]}
{Devn:}AMOS-monitor-command.PRG{[p,pn]}
{Devn:}command-file.CMD{[p,pn]}
{Devn:}command-file.DO{[p,pn]}
```

The CHAIN statement causes control to be passed to the specified BASIC program, command file, or monitor command program. The program name may be a full file specification, including device and account specifications. The

CHAIN statement causes the current program to be cleared from memory. The specified file is then located and executed from the beginning. A chained BASIC program must be a fully compiled program with the extension .RUN in order to be referenced by the CHAIN command. It may be in user memory (having previously been loaded via the monitor LOAD command) or it may be in system memory. (The System Operator may place a file in system memory by modifying the system initialization command file.) If it is not already in memory, it is loaded from the specified disk account into user memory and then executed. If it cannot be located, you are returned to AMOS command level with the error message:

```
?Cannot find program NAME.RUN
.
```

Some examples of the CHAIN statement:

```
70 CHAIN "PAYROL"          70 CHAIN "DSK1:PAYROL.CMD[100,7]"
```

There is no provision to start the chained file at any point other than the beginning. You may pass common variables between chained BASIC programs either by writing them out to a file and then having the chained program read them back in, or by using the COMMON assembly language subroutine. (See COMMON - BASIC Subroutine to Provide Common Variable Storage, (DWM-00100-18) in the "BASIC Programmer's Information" section of the AM-100 documentation package.)

For more information on CHAIN, see Chapter 16, "Chaining to BASIC and System Programs."

10.3 CLOSE

The format is:

```
CLOSE #file-channel
```

This statement closes an I/O file to further processing. It is discussed in detail in Chapter 15, "AlphaBASIC File I/O System."

10.4 DIM

The format is:

```
DIM variable1(expr1{,expr2,...exprN}){,...{,variableN(expr1{,expr2,...exprN})}
```

The dimension statement defines an array which is allocated dynamically at execution time. Once allocated, an array cannot be redimensioned during the execution of the program. There is no limit to the number of subscripts that may be used to define the individual levels within the array. The statement DIM A(20) defines an array with 20 elements, referenced as A(1)

through A(20). Multiple arrays may be dimensioned by a single DIM statement by separating them with commas.

Subscripts are evaluated at execution time and not at compile time, thereby allowing variables as well as numeric constants to be used as subscripts. The statement DIM A(B,C) allocates an array whose size depends on the actual values of B and C at the time the DIM statement is executed.

If a reference to an array is made during program execution without a previous DIM statement to define the array, BASIC assigns a default array size of 10 elements for each subscript level referred to.

String arrays may be allocated, such as DIM A$(5). The size of the array depends on the current default string size in effect as specified by the last STRSIZ command, since each element in the array must be this number of bytes. For instance, if the current STRSIZ is 10, the statement DIM A$(5) would allocate 5 elements * 10 bytes per element, or 50 bytes of memory for the array. Below are some examples of valid DIM statements:

```
DIM A(10)
DIM C(8,8), C$(10,4)
DIM TEST(A,B*4)
DIM A(B(4))
```

10.5 END

The format is:

```
END
```

This statement causes the program to terminate execution. The END statement does not terminate compilation of the program nor is it required at the end of the program. If other program statements follow the end of the program (e.g., subroutines), terminating the program with END prevents your program from incorrectly entering those statements and trying to execute them.

10.6 FILEBASE

The format is:

```
FILEBASE n
```

This statement sets the number used to refer to the first record of a random file. It is discussed in detail in Chapter 15, "AlphaBASIC File I/O System."

10.7 FOR, NEXT AND STEP

The format is:

```
FOR control-variable = expression1 TO expression2 {STEP {-}expression3}
                       {Statements}
NEXT {control-variable}
```

These statements initialize and control program loops. A loop is a structure in which the same statement or statements can be performed several times. Whether or not a loop is executed depends upon the value of a special "control-variable." AlphaBASIC FOR-NEXT loops follow the same format and restrictions as do other forms of BASIC. The control-variable used may be subscripted, and must be a floating point variable. The delimiters indicating the number of incrementations or decrementations to be performed on that variable may be any valid expression. FOR initializes the variable to the first expression. NEXT increments or decrements the value of the variable each subsequent loop. The variable name may be omitted in the NEXT statement, in which case the variable of the previous FOR statement is the one that is incremented. The control-variable is incremented or decremented in units indicated by the STEP statement. If no STEP modifier is used, the step value is assumed to be a positive 1. Unlike some other BASICs, an AlphaBASIC FOR-NEXT loop will always be performed at least once, even if you specify something like FOR I = 0 to 0. FOR and NEXT statements are illegal as direct statements except when they are incorporated into the same multi-statement line. For example:

```
FOR I = 1 TO 10 : PRINT I : NEXT I
```

Here are examples of some of the different forms FOR-NEXT loops may take:

```
10 FOR COUNTER = 1 TO 10
20 IF COUNTER/2 = INT(COUNTER/2) THEN PRINT COUNTER "is even." &
     ELSE PRINT COUNTER;"is odd."
30 NEXT COUNTER
```

```
10 INPUT "Enter date of first Sunday in the month: ",DAY
20 PRINT "The Sundays this month are on these dates:" : PRINT DAY
30 FOR A=DAY+7 TO 31 STEP 7 : PRINT A : NEXT A
```

```
10 FOR I = 10 TO 1 STEP -1
20     PRINT I
30 NEXT
```

Loops within loops are legal and are called nested loops. Loops may be nested to many levels. Each time the outermost loop is incremented (or decremented) once, the loop nested within it is executed from beginning to end. During the execution of the second loop, the third loop (if any) is fully executed each time the second variable is incremented. And so on, for each nested loop in the series. For example:

```
10 ! This program prints out a two-dimensional array,
20 ! and demonstrates nested loops.
```

```
30 DIM MATRIX(5,5)
40 ! The nested loops:
50   FOR I = 1 TO 5
60     FOR J = 1 TO 5
70       MATRIX(I,J)= I-J
80       PRINT MATRIX(I,J);
90     NEXT J
100  PRINT
110  NEXT I
```

The program above prints:

0	-1	-2	-3	-4
1	0	-1	-2	-3
2	1	0	-1	-2
3	2	1	0	-1
4	3	2	1	0

It is not good programming practice to branch out of a loop before its completion (via GOTOs, ON GOTOs, etc.) unless you give careful consideration to the BASIC system stack area. The stack area used by the loop is not reclaimed if you branch out of the loop, and doing so can cause a stack overflow error during program execution. A cleaner way of exiting a loop is simply to set the control-variable to the terminal value specified in the FOR statement. For example:

```
10 REM Example of exiting out of a FOR-NEXT loop.
20
30 START'LOOP:
40     FOR I=1 TO 100
50       INPUT "Enter number of pennies:",PENNIES
60       IF PENNIES<0 GOTO NEGATIVE'VALUE ! Don't jump out of the loop!
70       PRINT "You have";PENNIES/100;"dollars." : GOTO END'LOOP
80 ! If # < 0, print error message and set I to terminal value.
90       NEGATIVE'VALUE:
100        PRINT "You can't have negative pennies!" : I=100
110 ! End of loop, where we increment or decrement I.
120 END'LOOP:
130      NEXT I          ! If I = 100, we're all done.
140    PRINT "We're all done."
```

10.8 GOSUB (OR CALL) AND RETURN

The formats are:

```
GOSUB label or line number
CALL label or line number

RETURN
```

Calls a subroutine which starts at the line number or label referenced by the GOSUB or CALL statements. The subroutine exits via the RETURN statement, which returns control to the statement following the GOSUB or CALL statement. Executing a RETURN statement without first executing a GOSUB statement results in an error message. Both GOSUB and RETURN are illegal as direct statements. Note that the CALL verb is merely another way of specifying GOSUB for those programmers used to this verb from other languages.

It is often the case that you want to perform the same operation at various points within your program. A subroutine is a set of program statements that you may execute more than once simply by including an invocation for that subroutine (called a "call") within your program at the point where you would like to execute the routine. For example:

```
10  ! This program contains a subroutine that validates numeric entries
20  ! to make sure that they are greater than 0 and are less than 100.
30  PRINT "We are going to perform several mathematical operations."
40  PRINT "Your entries must be greater than 0 and less than 100."
50  PRINT : INPUT "Enter two numbers to be added: ",A,B
60  GOSUB VALIDATE      ! Check to make sure numbers are valid.
70  PRINT A;"+";B;"=";A+B
80  PRINT : INPUT "Enter two numbers to be subtracted: ",A,B
90  GOSUB VALIDATE      ! Check to make sure numbers are valid.
100 PRINT A;"-";B;"=";A-B
110 PRINT : INPUT "Enter two numbers to be divided: ",A,B
120 GOSUB VALIDATE      ! Check to make sure numbers are valid.
130 PRINT A;"/";B;"=";A/B
140 PRINT : PRINT "That's all..."
150 END
200 ! Subroutine to validate the data
210 VALIDATE:
220   IF A <= 0 OR B <= 0 THEN &
         PRINT "Error - negative or zero number!" : END
230   IF A >= 100 OR B >= 100 THEN &
         PRINT "Error - Number too big!" : END
240  RETURN
```

Remember that & (ampersand) is the symbol for a continuation line.

Note that we included an END statement at line 150 to separate the main program from our subroutine; otherwise, BASIC executes the VALIDATE subroutine after it reaches line 140, and we get a "RETURN without GOSUB" error.

Also note that the use of GOSUBs helps to modularize your programs, and thus makes them easier to design and maintain. Even before you completely "flesh out" your programs, you can insert dummy routines that will later contain complete code. For example:

```
10 ! This program will be a complete dental package.
20 PRINT "Welcome to the Acme Dental Package."
30 ! Perform initialization of data files
```

```
40 GOSUB INIT
50 ! Ask user to pick function from main menu.
60 GOSUB MENU
70 ! Do End-of-day Processing
80 GOSUB DAY'END
90 ! Finish up, close files, and exit.
100 GOSUB FINISH'UP
110 END
115             ! The subroutines start here.
200 INIT:
210     PRINT "This section will initialize files."
220     RETURN
300 MENU:
310     PRINT "This section will display the main menu and"
320     PRINT "ask user for selections."
330     RETURN
400 DAY'END:
410     PRINT "This section will perform day-end processing."
420     RETURN
500 FINISH'UP:
510     PRINT "This section will close files and clean up final data."
520     RETURN
```

You can nest subroutines. For example:

```
10  ! Demonstrating nested subroutines
20  PRINT "Main Program:"
30  GOSUB OUTERMOST ! OUTERMOST calls NEXTMOST and INNERMOST
40      PRINT "   Return from Outermost"
50  END
60
100 ! Here are the subroutines:
110 OUTERMOST:
120     PRINT "   Outermost subroutine"
130     GOSUB NEXTMOST
140     PRINT "         Return from Nextmost"
150 RETURN
160
200 NEXTMOST:
210     PRINT "         Nextmost subroutine"
220     GOSUB INNERMOST
230     PRINT "                 Return from Innermost"
240     RETURN
250
300 INNERMOST:
310     PRINT "                 Innermost subroutine"
320     RETURN
```

The program above prints:

```
Main Program:
   Outermost subroutine
                Nextmost subroutine
                        Innermost subroutine
                        Return from Innermost
                Return from Nextmost
   Return from Outermost
```

NOTE: You should always exit a subroutine via the RETURN statement for that subroutine rather than using a GOTO statement. The reason for this is that subroutine processing places certain information on BASIC's stack area; if you do not execute a RETURN statement, the stack area used by that subroutine is not reclaimed. Doing multiple branches out of a subroutine thus results in a "stack overflow" error message.

10.9 GOTO

The format is:

```
GOTO label or line number
```

or:

```
GO TO label or line number
```

The GOTO statement transfers execution of the program to a new program line. This program line must be identified either by a line number or a label somewhere in the program. You may use GOTOs to transfer control to a program line that is either before or after the program line containing the GOTO statement itself. For example:

```
10  ! Program to demonstrate use of GOTOs.
20    PRINT "This program computes your account balance. Enter a"
30    PRINT "Control-C to stop; enter deposits as negative amount."
40    INPUT "Enter old account balance: ",BALANCE
50  CALCULATE'BALANCE:
60    PRINT : INPUT "Enter debit amount: ",DEBIT
70    BALANCE = BALANCE - DEBIT
80    PRINT "Debit was:";DEBIT;"-- Current balance is:";BALANCE
90    GOTO CALCULATE'BALANCE
```

You can see that lines 50 through 90 constitute an endless loop in which control is eternally transferred from line 90 back to line 50 until the user types a Control-C.

If you use GOTOs on a multi-statement line, remember to place it last on the line; any statements after the GOTO will never get executed. For example:

```
10 PRINT GROSS : NET = GROSS - DEDUCTION : GOTO GET'TAX : PRINT DEDUCTION
```

the last statement, PRINT DEDUCTIONS, can never be executed.

10.10 IF, THEN AND ELSE

The format is:

IF expression {THEN} {statement}{label/line#}{ELSE{statement}{label/line#}}

The conditional processing features in AlphaBASIC give a wide variety of formats which duplicate just about all the functions performed by other versions of BASIC. Some of the format combinations that are acceptable are:

```
IF expression THEN label/line#
IF expression THEN label/line# ELSE label/line#
IF expression statement
IF expression statement ELSE statement
IF expression THEN statement
IF expression THEN statement ELSE statement
```

The above formats may be nested to any depth, and rather than go into detail we suggest that you play around with them to determine the actual restrictions that exist. Some examples:

```
IF A=5 THEN GOTO PROGRAM'EXIT
IF A=5 PROGRAM'EXIT
IF A>14 THEN 110 ELSE 220
IF B$="END" PRINT "END OF TEST"
IF TOTAL > 14.5 GOTO START
IF P=5 AND Q=6 IF R=7 PRINT 567 ELSE PRINT 56 ELSE PRINT "NONE"
IF A=1 PRINT 1 ELSE IF B=2 THEN 335 ELSE 345
IF A AND B THEN PRINT "A and B are nonzero."
```

Note that the expression evaluated by the IF statement is usually an expression that contains relative operators (e.g., IF A = B; IF A > 0; etc.). However, the expression may be any legal expression. For example:

```
A = 0
B = 1
IF B THEN PRINT "B is not zero."
IF (B AND A) PRINT "nonzero numbers" ELSE PRINT "at least one zero number."
```

When the IF statement evaluates the expression, it returns either a zero (for false) or a -1 number (for true), and conditionally performs the specified operations in response to that evaluation.

NOTE: A multi-statement line may take the place of a single statement in an IF-THEN statement. For example:

```
IF A = 3 THEN PRINT 4 : PRINT 5 ELSE PRINT "Answer is 0"
```

If A equals three, the statement above prints:

```
4
5
```

Otherwise, it prints:

```
Answer is 0
```

10.11 INPUT

The format is:

```
INPUT {"prompt-string",}variable1{,variable2...,variableN}
```

Allows data to be entered from your terminal and loaded into specific variables at run-time. The INPUT statement contains one or more variables separated by commas. If you omit the optional prompt string, BASIC displays a question mark on the terminal display to signal a request for data entry. If you provide the prompt string, BASIC displays it instead of the question mark to prompt the user of your program for data. (NOTE: If you wish to suppress a prompt altogether, use a null prompt string; for example: INPUT "",A$,B$.) Your prompt string must be in the form of a string literal; that is, it must be enclosed with quotation marks. For example:

```
INPUT "Enter your account number: ",ACCOUNT'NUM
Enter your account number:
```

You may specify both numeric and string variables in the INPUT statement. A numeric variable requires that the data entered be in one of the acceptable floating point formats. String variables require that the data be an ASCII string of characters. Some examples of valid INPUT statements are:

```
INPUT A
INPUT "Enter account #, name, and age: ",NEW'ACCOUNT,NAME$,AGE
INPUT "",A,B,C
INPUT "Enter positive number:",NUMBER
INPUT Q(8)
```

If you specify multiple variables in the INPUT statement, the user of your program is expected to enter multiple items of data. If the data being entered is numeric, the user may separate data items with commas or spaces. If the data being entered is string, the user must separate data items with commas. (NOTE: For information on the statement to use if you want to enter strings that contain commas, quotes, and other special characters, see Section 10.12, "INPUT LINE.")

If a user of your program does not enter as many items of data as are expected by the variables in the INPUT statement, BASIC displays a double question mark to ask for more. For example:

```
INPUT A,B,C (RET)
? 1,2 (RET)
?? 3 (RET)
```

The direct statement above asks for three items of numeric data. Because we only entered two values, BASIC responded with a "??" symbol to ask for the third value.

Be careful to correctly enter the type of data that the variables in the INPUT statement expect. If an error occurs (for example, if you enter a string for a numeric variable), BASIC sets that variable to zero. For example:

```
INPUT A1(RET)
? ME(RET)
PRINT A1(RET)
 0
```

Therefore, your programs should make sure that the correct data has been entered. (Remember that the mode independence of AlphaBASIC permits the entry of numeric data for string variables; AlphaBASIC automatically converts such data to string format.)

If a value has not been assigned to a variable, BASIC assumes that the variable contains a zero (if a numeric variable) or a null (if a string variable). If you type a RETURN or a Control-C in response to an INPUT statement request for data, BASIC leaves the variable being inputted set to a zero or null (if a value has not yet been assigned) or to the value previously assigned to the variable.

For example:

```
A=3(RET)
INPUT A(RET)
?(RET)
PRINT A(RET)
 3
```

If you type a RETURN or Control-C in response to a data request, and the INPUT statement contains several variables, BASIC skips over any variables remaining in the INPUT statement, leaving their values unchanged. An example might help to clarify:

```
10 INPUT "Enter day, month, year: ",DAY,MONTH,YEAR
20 PRINT "Day:";DAY,"Month:";MONTH,"Year:";YEAR
30 PRINT : GOTO 10
```

```
RUN (RET)

Enter day, month, year: 21,4 (RET)
?? (RET)
Day: 21         Month: 4         Year: 0

Enter day, month, year: 8 (RET)
?? (RET)
Day: 8          Month: 4         Year: 0

Enter day, month, year: 31,12,1980 (RET)
Day: 31         Month: 12        Year: 1980

Enter day, month, year:  ^C
Operator interrupt in line 10
```

You may also use the INPUT statement to read data from sequential files. It takes the form:

```
INPUT #file-channel,variable1{,variable2,...variableN}
```

NOTE: INPUT skips over nulls in data, and just waits for the next character. (This is important to know if you plan to input from devices.)

For more information on this use of the statement, see Chapter 15, "AlphaBASIC File I/O System."

10.1 INPUT LINE

The format is:

```
INPUT LINE {"prompt-string",}variable1
```

Although you may specify a numeric variable, the real purpose of INPUT LINE is to allow you to enter string data from your terminal that includes commas, quotation marks, blanks, and other special characters. You will usually want to use INPUT (see the section above) for inputting numeric data or multiple items of string data.

INPUT LINE loads into the specified string variable an entire line up to but not including the carriage return and linefeed that end the line. Do not specify more than one string variable in the INPUT LINE statement.

BASIC never prints a question mark prompt for INPUT LINE as it does for INPUT, but you may include your own prompt string, which BASIC will display as a request for data. Such a prompt string must be a string literal enclosed in quotation marks.

Unlike INPUT, if you type a RETURN in response to a data request, INPUT LINE sets the variable to zero (if numeric variable) or null (if string variable). (Remember, in like case, INPUT leaves the value of the variable unchanged.)

When you use INPUT LINE, remember that the default size of unmapped string variables is ten bytes; if you want to use strings larger than that, use the STRSIZ statement to reset the default string size. (See Section 10.26 for information on STRSIZ.)

Some examples of the statement are:

```
INPUT LINE A$
INPUT LINE "ENTER YOUR FULL NAME, PLEASE: ",NAME
```

You may also use the INPUT LINE statement to read data from a sequential file. It takes the form:

```
INPUT LINE #file-channel,variable1
```

For more information on using INPUT LINE and files, see Chapter 15, "AlphaBASIC File I/O System."

10.2 KILL

The format is:

```
KILL filespec
```

KILL deletes a file from a disk. It is discussed in detail in Chapter 15, "AlphaBASIC File I/O System."

10.3 LOOKUP

The format is:

```
LOOKUP filespec, result-variable
```

The result variable must be a floating point nummber.

This statement searches for a file and returns its size. It is discussed in detail in Chapter 15, "AlphaBASIC File I/O System."

10.4 LET

The format is:

```
LET variable = expression
```

Assigns a calculated value to a specific variable during execution of the program. You do not have to specify the LET keyword in an assignment statement.

```
LET A5 = 12.4
LET SUM(4,5) = A1+SQR(B1)
LET C$ = "JANUARY"
A5 = 12.4
SUM(4,5) = A1+SQR(B1)
C$ = "JANUARY"
```

10.5 ON - GOSUB (CALL)

The formats are:

```
ON expression GOSUB label/line#1{,label/line#2,...label/line#N}
ON expression CALL label/line#1{,label/line#2,...label/line#N}
```

The expression can be any valid expression which is evaluated and truncated to a positive integer result. The result of the expression evaluation is then tested. The subroutine at label/line#1 is executed if the result is 1, the subroutine at label/line#2 is executed if it is 2, etc. If the result is zero, negative or greater than N, the program falls through to the next statement.

As with the GOSUB statement, the verb CALL may be used in place of the verb GOSUB, giving an ON CALL statement. Here is an animation program using ON - GOSUB:

```
10 I = INT(3*RND(0)+1)                    !Random number from 1 to 3.
20 ON I GOSUB UP, DOWN, STRAIGHT          !Go to 1 of 3 subroutines.
30 GOTO 10
40 UP: PRINT "/"; TAB(-1,3); : RETURN     !Draw symbol, go up 1 row.
50 DOWN: PRINT TAB(-1,4);"\"; : RETURN    !Go down 1 row, draw symbol.
60 STRAIGHT: PRINT "__"; : RETURN         !Draw symbol.
```

10.6 ON - GOTO

The format is:

```
ON expression GOTO label/line#1{,label/line#2,...label/line#N}
```

(Changed 30 October 1980)

The ON GOTO statement allows multi-path GOTO branching to one of several points within the program based on the result of evaluating an expression.

The expression can be any valid expression which is evaluated and truncated to a positive integer result. The result is then tested to branch to label/line#1 if 1, label/line#2 if 2, label/line#3 if 3, etc. If the result is zero, negative or greater than N, the program falls through to the next statement. The following is a portion of a menu-selection program:

```
10 PRINT TAB(22)"Select One of the Following Operations:" : PRINT
20 PRINT TAB(25)"1.  Insert/Edit NAME Information."
30 PRINT TAB(25)"2.  Insert/Edit PHONE NUMBER Information."
40 PRINT TAB(25)"3.  Quit without insertion or editing."
50 PRINT : INPUT "Your choice (1, 2 or 3)? ",A
60 ON A GOTO NAME, PHONE, QUIT
100 NAME:  INPUT "Select a name: ",N

       [THE PROGRAM CONTINUES WITH ALL THREE ALTERNATIVES]
```

10.18 OPEN

The format is:

```
OPEN #file-channel,filespec,mode{,recsize,recnum}
```

Opens an I/O file for processing. It is discussed in detail in Chapter 15, "AlphaBASIC File I/O System."

10.19 PRINT

The format is:

```
PRINT expression-list
```

The PRINT statement tells BASIC to evaluate and display the expressions that you specify. For example:

```
PRINT 3+4;"HELLO"+" YOU"
```

returns:

```
7 HELLO YOU
```

BASIC prints a carriage return/line-feed after the expression list. Remember that an expression may consist of a string or numeric variable, numeric constant, string literal, function with arguments, operator symbols, or a combination of these elements. For example, the following is one string expression: "STRING DATA" + NAME$ + MID$(A$,1,2).

BASIC displays numeric data with a trailing blank. It also prints one leading blank if the number is positive, or no leading blank if the number is negative. BASIC displays string data with no leading or trailing blanks.

You may place more than one expression after the PRINT keyword if you separate them with commas or semicolons. If you separate the expressions by semicolons, BASIC does not print extra spaces when it prints the evaluations of those expressions. For example:

```
PRINT 12+12;-32;8/2
```

returns:

```
 24 -32  4
```

There are no blanks between the numbers above except for the normal leading and trailing blanks displayed with numeric data.

If you separate the expressions by commas, BASIC prints the data in "print zones." BASIC divides the area in which data is to be displayed into five zones of 14 spaces each. If an expression in a PRINT statement is followed by a comma, BASIC prints that expression in the next available print zone. For example, the statements:

```
20 PRINT 34,1024,-32,-100.2,20
30 PRINT "AA","BB","C","DDD","A","B","C"
```

display:

```
 34            1024          -32           -100.2        20
AA             BB            C             DDD           A
B              C
```

When you look at the display above, remember that BASIC prints numeric data with a leading and trailing blank if the number is positive, but just a trailing blank if the number is negative.

Note that the strings in line 30 were displayed on two different lines; that is because when BASIC still has an expression to print after it has printed something in the fifth zone, it starts over again with the first zone on the next line.

If you end the PRINT statement expression list with a semicolon or comma, BASIC does not output a carriage return/line-feed when it finishes displaying that expression list. This will make the output resulting from the next PRINT or INPUT statement to appear on the current display line. The next output will appear in the next print zone if the current PRINT statement ends with a comma; or, the next output will appear immediately following the last character of the current PRINT statement if the PRINT statement ends with a semicolon.

Here are a few examples of the PRINT statement (for illustrative purposes, we are assuming that A$ is "HERE" and A equals 7):

```
PRINT                          !Yields a blank line
PRINT A                        !Yields 7
PRINT A$                       !Yields HERE
PRINT 1+2                      !Yields 3
PRINT "ANY TEXT"               !Yields ANY TEXT
PRINT "NOTE THE COMMA",A$      !Yields NOTE THE COMMA       HERE
? "YOU ARE NUMBER";A           !Yields YOU ARE NUMBER 7
? "YOU ARE #";A;"IN CLASS."    !Yields YOU ARE # 7 IN CLASS.

PRINT "THERE ARE";             !Semicolon suppresses carriage-
PRINT A;"DAYS LEFT."           !return/linefeed and yields
                               !THERE ARE 7 DAYS LEFT.
```

(Remember that the "?" symbol is an abbreviation for the PRINT keyword.)

You may also use the PRINT statement for writing data to sequential files. It takes the form:

```
PRINT #file-channel,expression-list
```

For details on this, refer to Chapter 15, "AlphaBASIC File I/O System."

10.20 PRINT USING

The formats are:

```
variable=expression USING format-string
PRINT USING format-string, expression-list
PRINT expression USING format-string
```

PRINT USING is supported for formatting output and is described extensively in Chapter 13, "Formatting Output (PRINT USING and Extended Tabs)."

10.21 RANDOMIZE

The format is:

```
RANDOMIZE
```

Resets the random number generator seed to begin a new random number sequence starting with the next RND(X) function call. (See Section 11.1.9 for information on the random number generator.)

10.22 READ, RESTORE, AND DATA

The formats are:

```
READ variable1{,variable2,...variableN}
RESTORE
DATA data1{,data2,...dataN}
```

These calls allow data to be an integral part of the source program with a method for getting this data into specific variables in an orderly fashion. DATA statements are followed by one or more literal values separated by commas. String literals need not be enclosed in quotes unless the literal data contains a comma. All data statements are placed into a dedicated area in memory no matter where they appear in the source program. READ statements are followed by one or more variables separated by commas. Each time a READ statement is executed, the next item of data is retrieved from the DATA statement pool and loaded into the variable named in the READ statement. If there is no more data left in the data pool, the program can only continue to read data if a RESTORE statement is executed, which reinitializes the reading of the data pool from the beginning again. Otherwise, an error message results and the program is aborted. Here are some forms that READ and DATA may take.

```
DATA 1,2,3,4,5
DATA 2.3,0.555,ONE STRING,"4,4"
READ A,B,C
READ A$
READ C(2,3),B$(4)
```

The following is a program example using READ, RESTORE, and DATA:

```
 10 !Sample program to illustrate READ, DATA and RESTORE
 20 PRINT TAB(10)"This program gives you an estimate of your automobile's"
 30 PRINT TAB(10)"value (due to depreciation) over a period of five years."
 40 PRINT : INPUT LINE "How much did you pay for your car?  $",WORTH
 50 PRINT "Based on national averages, your car will depreciate this way:"
 60 PRINT : FOR I = 1 TO 5
 70 PRINT "After the "; : READ YEAR$ : PRINT YEAR$;" year, your car ";
 80 PRINT "will be worth about"; : READ PERCENT
 90 WORTH = WORTH * PERCENT
100 PRINT WORTH USING "$$######,.##"
110 NEXT I : PRINT
200 DATA first,.77,second,.78,third,.79,fourth,.81,fifth,.84
300 RESTORE
310 INPUT LINE "Would you like to see another depreciation schedule?  ",L$
320 IF L$[1,1]="Y" OR L$[1,1]="y" THEN GOTO 40 ELSE PRINT "Goodbye."
```

A program run of the above example might read:

```
        This program gives you an estimate of your automobile's
        value (due to depreciation) over a period of five years.

How much did you pay for your car? $8634.79 (RET)
Based  on national averages, your car will depreciate this way:

After the first year, your car will be worth about  $6,648.79
After the second year, your car will be worth about  $5,186.05
After the third year, your car will be worth about  $4,096.98
After the fourth year, your car will be worth about  $3,318.56
After the fifth year, your car will be worth about  $2,787.59

Would you like to see another depreciation schedule?  N (RET)
Goodbye.
```

Statement 300 restored the data in the data pool, built from line 200, in case the user of this program had elected to continue.

The READ statement is also used for reading data from random access files. The format is:

```
READ #file-channel,variable1{,variable-2,...variableN}
```

It is discussed in detail in Chapter 15, "AlphaBASIC File I/O System."

10.23 SCALE

The format is:

```
SCALE value
```

SCALE is a scaled arithmetic modifier. It is discussed in detail in Chapter 14, "Scaled Arithmetic."

10.24 SIGNIFICANCE

The format is:

```
SIGNIFICANCE value
```

The significance statement allows you to dynamically change the default value of the numeric significance of the system for unformatted printing. The significance value can be any value from 1 through 11 and represents the maximum number of digits to be printed in unformatted numbers. Rounding off to the specific number of digits is not performed until just before the printing of the result. The statement SIGNIFICANCE 8, for instance, sets the number of printable digits to 8. The value is interpreted at run-time and therefore may be any valid numeric expression, including variables. The current significance of the system is ignored when PRINT USING is in effect.

Note that the SIGNIFICANCE statement only affects the final printed result of all numeric calculations. The calculations themselves and the storage of intermediate results are always performed in full 11-digit precision to minimize the propagation of errors.

The significance of the system is set at 6 digits when the system is first started. This is equivalent to standard single-precision formats used in most of the popular versions of BASIC. The significance is not reset by the RUN command and therefore may be set in interactive mode in a direct statement just prior to the actual running of a test program. Of course, any SIGNIFICANCE statements encountered during the execution of the program reset the value.

10.25 STOP

The format is:

```
STOP
```

Causes the program to suspend execution and print the message "Program stop at line nnnn." If you are in interactive mode, you may then continue to the next statement in sequence by executing a CONT command or a single-step command.

10.26 STRSIZ

The format is:

```
STRSIZ value
```

The string size statement sets the default value for all strings which are encountered for the first time during the compilation phase. Initially, the default value of all strings in the absence of a STRSIZ statement is 10 bytes. The statement STRSIZ 25, for instance, causes all newly allocated strings which follow to have a maximum size of 25 bytes instead of 10 bytes. This includes the allocation of string arrays. The size value is evaluated at compilation time and therefore must be a single positive integer.

10.27 WRITE

The format is:

```
WRITE #file-channel,expression-list
```

Writes a record to a random access file. It is discussed in detail in Chapter 15, "AlphaBASIC File I/O System."

10.28 XCALL

The format is:

```
XCALL routine{,argument1{,argument2,...argumentN}}
```

Executes an external assembly language subroutine. Assembly language subroutines are discussed in detail in Chapter 18, "Calling External Assembly Language Subroutines."

For information on the assembly language subroutines available for use with BASIC programs, see the "BASIC Programmer's Information" section of the AM-100 documentation packet.

CHAPTER 11

BASIC FUNCTIONS

The following is a list of the currently implemented AlphaBASIC functions. Functions compute and return a value and are elements of an expression. The function either operates on or is controlled by the argument, which is enclosed in parentheses. There are four main categories of functions. Numeric and trigonometric functions return numeric values. Control functions are used to indicate the status of file input and output operations and system operations. String functions operate on numeric values or strings of one or more characters in length, and return string values.

Functions are different from program statements in that they return a value. In order to see or use that value, you must include the function in a program statement that evaluates the expression that the function call is a part of. For example:

```
10 SQR(16)
```

will not display a value. You must either assign the value returned by the function to a variable or display the value via a PRINT statement if you want to use or see the value returned. For example:

```
20 ROOT = SQR(16)
30 RESULT = ROOT * (SQR(NUMBER) + 24)
```

or:

```
40 PRINT "Answers are: "; SQR(16) + 100, SQR(24)
```

11.1 NUMERIC FUNCTIONS

Numeric functions accept a string or numeric argument, and return a numeric value. Note that the mode independence feature of the expression processor performs automatic conversions if a numeric argument is used where a string argument is expected, and vice versa.

11.1.1 ABS(X)

Returns the absolute value of the argument X. For example, ABS(-32.4) returns 32.4, and ABS("17.2") returns 17.2.

11.1.2 ASC(A)

Returns the ASCII decimal value of the first character of argument A. The argument may be either a string literal or string variable. For example:

```
ASC("A")
ASC(A$)
```

11.1.3 EXP(X)

Returns the constant e (2.7182818285) raised to the power X.

11.1.4 FACT(X)

Returns the factorial of X.

11.1.5 FIX(X)

Returns the integer part of X (fractional part truncated).

11.1.6 INT(X)

Returns the largest integer less than or equal to the argument X. The only time you will see a difference between using INT and FIX is if you are working with negative numbers. For example, the largest integer less than or equal to 23.4 is 23. However, the largest integer less than or equal to -23.4 is -24. (FIX would have returned -23.)

11.1.7 LOG(X)

Returns the natural (base e) logarithm of the argument X.

11.1.8 LOG10

Returns the decimal (base 10) logarithm of the argument X.

11.1.9 RND(X)

Returns a random number generated by a pseudo-random number generator. The number returned is based on a previous value known as the "seed," and is between 0 and 1. The argument X controls the number to be returned. If X is negative, it is used as the seed to start a new sequence of numbers. If X is zero or positive, the next number in the sequence is returned, depending on the current value of the seed (this is the normal mode). The RANDOMIZE statement may be used to create a seed which is truly random and not based on a fixed beginning value set by the system.

NOTE: If you want to generate a random number greater than or equal to number A and less than number B, you can use the expression: (B-A)*RND(0)+A. Note that the INT function is used when generating random integer numbers. For example, to generate a random integer greater than or equal to 5 and less than 31, use the expression: INT(26*RND(0)+5) where 26=B-A.

11.1.10 SGN(X)

Returns a value of -1, 0 or 1 depending on the sign of the argument X. Gives -1 if X is negative, 0 if X is 0 and 1 if X is positive.

11.1.11 SQR(X)

Returns the square root of the argument X.

11.1.12 VAL(A)

Returns the numeric value of the string variable or string literal A converted to floating point under normal BASIC format rules. For example, VAL("123") returns 123.

11.2 TRIGONOMETRIC FUNCTIONS

The following trig functions are implemented in full 11-digit accuracy:

SIN(X)	Sine of X
COS(X)	Cosine of X

TAN(X)	Tangent of X
ATN(X)	Arctangent of X
ASN(X)	Arcsine of X
ACS(X)	Arccosine of X
DATN(X,Y)	Double arctangent of X,Y

11.3 CONTROL FUNCTIONS

Control functions indicate the status of file input and output operations, and provide information on system operations.

11.3.1 EOF(X)

The EOF function returns a value giving the status of a file whose file-channel number is X. The file is assumed to be open for sequential input processing. The values returned by the EOF function are:

-1	if the file is not open or the file-channel number X is zero. (NOTE: A file-channel number of zero indicates that the terminal is being used as the file.)
0	if the file is not yet at end-of-file during input calls
1	if the file has reached the end-of-file condition

Due to the method used by the AMOS operating system for processing files, the end-of-file status is not achieved until after an INPUT statement has been executed which reaches the end-of-file condition. Any INPUT statements which reach end-of-file return numeric zero or null string values forever more. This means that the normal sequence for processing sequential input files would be to INPUT the data into the variables and then test the EOF(X) status before actually using the data in those variables, since if an end-of-file has been reached that data will be no good.

End-of-file should only be tested for sequential input files. Files open for output or for random processing always return a zero value.

11.3.2 ERF(X)

The ERF function returns an indication of a file soft error condition. Soft errors during file access operations do not give you any indication unless you query the file with the ERF function. If the returned value of X is not zero, an error or abnormal condition exists as a result of the preceding file operation. The only soft errors currently returned concern ISAM file operations. For more information, see Chapter 19, "Using ISAM From Within BASIC."

11.3.3 ERR(X)

Returns a status code which refers to program status during error trapping. There are 33 separate codes. A complete list of these codes is found in Section 17.2.1, "Error Codes Returned by ERR." If X is 0, ERR returns the specific code of the error detected; if X is 1, ERR returns the number of the last program line encountered before the error occurred. If X is 2, ERR returns the file number of the last file accessed.

11.3.4 OTHER CONTROL FUNCTIONS

See Chapter 12, "System Functions," for information on the following functions:

MEM(X)	-	Returns the number of free bytes in system memory.
BYTE(X)	-	Enables you to bring in 8 data bits from a memory location.
WORD(X)	-	Enables you to bring in 16 data bits from a memory location.
DATE	-	Sets or reads the system date.
IO(X)	-	Enables the 256 I/O ports to be read from or written to.
TIME	-	Sets or reads the system time.

11.4 STRING FUNCTIONS

The following string functions accept numeric or string arguments, and return strings. Note that the mode independence feature of the expression processor performs automatic conversions if a numeric argument is used where a string argument is expected, and vice versa.

11.4.1 ASC(X)

Returns the ASCII decimal value of the first character in string A$. If the string A$ reads, for example, "Zirconium's atomic number is:", the result of the statement PRINT ASC(A$) is 90, the ASCII value (in base 10) of upper case Z. For the statement PRINT ASC("A$"), where the argument is between quotation marks and is the literal string to be operated upon, at execution time BASIC returns the ASCII value of A, or 65.

11.4.2 CHR$(X) OR CHR(X)

Returns a single character having the ASCII decimal value of X. Only one character is generated for each CHR function call. For instance, if you type PRINT CHR$(90) as a direct statement, the upper case letter Z is returned to you.

11.4.3 INSTR(X,A$,B$)

Performs a search for the substring B$ within the string A$, beginning at the Xth character position. It returns a value of zero if B$ is not in A$, or the character position if B$ is found within A$. Character position is measured from the start of the string, with the first character position represented as one. Some direct statements will illustrate:

```
A$="ELEPHANT"
B$="ANT"
PRINT INSTR(1,A$,B$)
 6
```
(Substring B$ starts the sixth character from the left)

```
A$="CROCODILE"

? INSTR(2,A$,"COD")
 4
```
(The specified string begins at the fourth character position)

```
?INSTR(8,"MEADOWLARK","LARK")
 0
```
(The specified string "LARK" is not found in the string "ARK", which is the string starting at the 8th position)

NOTE: Remember the "?" symbol is an abbreviation for "PRINT."

11.4.4 LCS(A$)

Returns a string which is similar to the argument string (A$), but with all characters translated to lower case. If A$ is "A is for Alpha", the function LCS(A$) yields the string "a is for alpha".

11.4.5 LEFT(A$,X) or LEFT$(A$,X)

LEFT$(A$,X) Returns the leftmost X characters of the string expression A$. If A$ reads "Now is the time", the function LEFT$(A$,7) produces the substring "Now is ", which includes the trailing blank after "is".

11.4.6 LEN(A$)

Returns the length in characters of the string expression A$. If A$ is "`Wherefore art thou, Romeo?'", the function LEN(A$) returns the number 28 because there are 28 characters in that string, including spaces and punctuation.

11.4.7 MID(A$,X,Y) or MID$(A$,X,Y)

Returns the substring composed of the characters of the string expression A$ starting at the Xth character and extending for Y characters. A null string is returned if X is greater than the length of A$. If A$ reads "The quick brown fox jumped over the sleeping dog", then the function

MID(A$,17,15) returns the substring "fox jumped over", which begins at the seventeenth letter of the string and is fifteen characters long.

11.4.8 RIGHT(A$,X) or RIGHT$(A$,X)

Returns the rightmost X characters of the string expression A$. If A$ is "I THINK, THEREFORE I AM", the function RIGHT(A$,4) produces the substring "I AM". As another example, RIGHT(1234,2) returns 34. (Remember that you can use numeric arguments for many string functions.)

11.4.9 SPACE(X) or SPACE$(X)

Returns a string of X spaces in length. The statement

```
70 PRINT "COLUMN A"; : PRINT SPACE(10); : PRINT "COLUMN B"
```

outputs the following:

```
COLUMN A          COLUMN B
```

where the 10 spaces between the first and second strings are the result of the SPACE(10) function. SPACE is especially handy for padding strings to a fixed length. For example:

```
5 STRSIZ 25
10 !Name must be  25 spaces
20 INPUT "Name?",NAME$
30 IF LEN(NAME$)<25 THEN NAME$ + SPACE(25-LEN(NAME$))
```

11.4.10 STR(X) or STR$(X)

Returns a string which is the character representation of the numeric expression X. No leading space is returned for positive numbers.

11.4.11 UCS(A$)

Returns a string which is similar to the argument string (A$), except that all characters are translated to upper case. If A$ is "M is for Micro," the function UCS(A$) yields the string "M IS FOR MICRO."

CHAPTER 12

SYSTEM FUNCTIONS

AlphaBASIC supports a unique group of operators called system functions, which provide the ability to get to the I/O ports, physical memory (sometimes referred to in other BASICs as PEEK and POKE), and various system parameters. The syntax of a system function parallels that of a standard function, with the reserved word representing the desired function followed by optional arguments enclosed within parentheses. The major difference is that the reserved word of a system function may appear on the left side of an assignment statement, where it is used as an output or write condition to the system function. System functions used within expressions on the right side of an assignment statement perform an input or read operation and deliver back a result to be used in the expression evaluation.

12.1 BYTE(X) AND WORD(X)

The BYTE and WORD system functions allow you to inspect and alter any memory locations within the 64K memory addressing range of the machine. These operations have often been called PEEK and POKE statements in other implementations of BASIC. The BYTE functions deal with 8 bits of data in the range of 0-255, and the WORD functions deal with 16 bits of data in the range of 0-65535, inclusive. Any unused bits are ignored, with no error message. Note that these commands are not protected; it is possible to cause severe damage to the operating system in memory if you use the commands improperly.

```
BYTE(X) = <expr>    !writes the low byte of expr into decimal memory loc X
WORD(X) = <expr>    !writes the low word of expr into decimal memory loc X
A       = BYTE(X)   !reads decimal memory loc X and places the byte into A
A       = WORD(X)   !reads decimal memory loc X and places the word into A
```

12.2 DATE

The DATE system function is identical to the TIME function except that it sets and returns the two-word system date.

```
DATE = <expr>   !sets system date to expr
A    = DATE     !returns system date into A
```

The following program translates the binary data stored in the system DATE location into floating point form.

```
10 ! The system stores the date in binary form; the small program
15 ! below translates the binary date into floating point form.  It
20 ! also allows you to set the system date from within BASIC.
25 MAP1 BINDATE,B,4
30 MAP1 FILLDATE,@BINDATE
35  MAP2 MONTH,B,1
40  MAP2 DAY,B,1
45  MAP2 YEAR,B,1
50 BINDATE = DATE
55 PRINT "Month:";MONTH,"Day:";DAY,"Year:";YEAR
60 INPUT "Enter Month, Day, Year: ",MONTH,DAY,YEAR
65 DATE=BINDATE
70 PRINT "Month:";MONTH,"Day:";DAY,"Year:";YEAR
```

12.3 IO(X)

The IO system function allows the 256 I/O ports to be selectively read from or written to. In both cases only one byte is considered, and an output expression greater than 255 merely ignores the unused bits. The range of ports available is 0 to 255.

```
IO(X) = <expr>  !writes the low byte of expr to decimal port X
A     =  IO(X)  !reads decimal port X and places the result into A
```

12.4 MEM(X)

Returns a positive integer value which specifies the decimal number of bytes currently in use for various memory areas used by the compiler system. The most common use of this is to return the number of free bytes left in the user memory partition. This MEM(0) call duplicates the action performed by the FRE(X) function in other versions of BASIC. Other values of the argument X return memory allocations which pertain to various areas in use by the compiler, and may or may not be of use to you. The byte counts returned for the various values of X are:

0 - Free memory space remaining in current user partition
1 - Total size of current user partition
2 - Size of source code text area

3 - Size of user label tree
4 - Size of user symbol tree (variable names and user function names)
5 - Size of compiled object code area
6 - Size of data pool resulting from all compiled DATA statements
7 - Size of dummy data termination field (always zero)
8 - Size of array index area (dynamic links to variable arrays)
9 - Size of variable storage area (excluding arrays)
10 - Size of file I/O linkage and buffer area
11 - Size of variable array storage area (dynamically allocated at run-time)

Some of these values will be meaningless when running the run-time object module in compiler mode, such as 2, 3 and 4.

12.5 TIME

The TIME system function requires no argument and is used to set and retrieve the time of day as stored in the system monitor communications area. The time is stored as a two-word integer representing the number of clock ticks since midnight. You are responsible for conversions to printable format in those cases where it is required. One clock tick represents one interrupt from the CPU line clock, which is usually 60 hz for domestic systems and 50 hz for overseas systems. Dividing the time by the clock rate gives the number of seconds since midnight. Converting this to current time is then accomplished by successive divisions by 60 to get minutes, and again by 60 to get hours.

```
TIME = expression       !sets time-of-day in system to expression
   A = TIME             !returns time-of-day in clock ticks into A
```

The small program below converts the value returned by TIME into actual hours, minutes, and seconds.

```
100  T = TIME            ! Get time
120  CLOCK = 60          ! Clock frequency in Hz
130  HOURS = INT(T/(CLOCK^3))                    ! Compute hours
140  MINS = INT(T/(CLOCK^2)) - (HOURS * 60)      ! Compute minutes
150  ! Compute seconds
160  SECS = INT(T/CLOCK) - ((HOURS * (60^2)) + (MINS * 60))
170  H'MOD:  ! Adjust HOURS to 24-hour clock range.
180  IF HOURS > 23 THEN HOURS = HOURS - 24 : GOTO H'MOD
190  PRINT (HOURS USING "#Z"); ":";(MINS USING "#Z"); &
            ":"; (SECS USING "#Z");
```

There are a couple of things you should note about the program above:

1. The value CLOCK'FREQUENCY will vary depending on whether your system operates on 60 Hz or 50 Hz.

2. Since TIME returns the number of clock ticks since 12:00, if your system has been on for a couple of days this number can easily

cause HOURS to exceed 23; line 160 converts the value of HOURS to a number within the range of a 24-hour clock.

3. Note the use of the PRINT USING statement in line 170 to print single-digit time values with a leading zero. (The next chapter contains more information on PRINT USING.)

CHAPTER 13

FORMATTING OUTPUT (PRINT USING AND EXTENDED TABS)

Most BASIC business applications programs spend a great deal of effort in generating reports and printouts in which data must be neatly and clearly presented. In other words, correctly formatting output is usually a major concern of the BASIC programmer.

AlphaBASIC provides several important features that help you to format data. This chapter discusses how to employ the USING modifier to format numeric and string data via format strings. We also discuss the extended tab functions that allow you to control the output of data on the terminal screen.

13.1 THE USING MODIFIER

The USING modifier allows you to format numeric or string data using a format string (sometimes called an "editing mask") specified by you. Although you can use the USING modifier to store the formatted data in a string variable, you may also use it in combination with the PRINT statement to send the formatted data to a terminal display or to a file. (For information on PRINT, see Section 10.19, "PRINT" and Section 15.3.9, "PRINT.")

By "formatting" data, we mean the process of adjusting the appearance of data (e.g., by inserting commas or spaces) so that it fits the pattern of a specific format string. It might help to think of the format string as a template or pattern with which you are going to control the format of your data. The USING modifier allows you to apply the format string to your data.

Using format strings and the USING modifier, you can do such things as: line columns of numbers up by their decimal points; insert dollar signs and commas into numeric data to represent dollar amounts; line up numeric and string data within specified fields; generate and print leading zeros for numeric data; print asterisks instead of leading spaces; print numeric data in exponential form, etc.

The sections below talk about the special formatting characters within the format string that allow you to perform such adjustments.

The statements in which you use the USING modifier take these forms, where expression is usually a numeric or string constant, or a numeric or string variable:

```
variable = expression USING format-string

PRINT expression USING format-string

PRINT USING format-string, expression-list
```

For example, if you want to format the number 2345.678 with the format string "$$#####.##", you could say:

```
NUMBER = 2345.678 USING "$$#####.##"
PRINT 2345.678 USING "$$#####.##"
PRINT USING "$$#####.##",2345.678
```

(NOTE: The first format may only be used for numeric data; the other two formats may be used for string or numeric data. Also, remember that USING has the lowest precedence of all operators. Therefore, all other operations in expressions surrounding the USING operator are performed before formatting is done. For example, PRINT 23+4 USING "###"+".#" produces 27.0.) The format string may be a string expression (for example, MID$(A$,4,5), a string constant (for example, "###.##"), or a string variable (for example, MASK$).

If you use the third PRINT USING variant above, you may supply a list of expressions to be formatted, separating the expressions with commas as with the regular PRINT statement (e.g., PRINT USING "#####.##",A,B,C,D,E). If you supply more expressions than the format string is meant to handle, BASIC re-uses the format string until each of the elements in the expression list has been formatted. If you supply fewer expressions than the format string is meant to handle, BASIC ignores the unused portion of the format string.

NOTE: You may also send formatted data to a file by specifying a file number after the PRINT keyword (e.g., PRINT #1, USING format-string, expression-list). For information on sending data to files, see Section 15.3.9, "PRINT."

13.2 FORMATTING CHARACTERS

The sections below discuss the special characters that make up the format string; these special characters control the output of your data. Characters other than these special formatting characters which appear in a format string are output literally as part of your data.

13.2.1 The \ Symbol (String Fields)

Although you will most often be interested in formatting numeric data, you may also specify fields for string data via the backslash symbol (\). Two backslashes define a string field whose size equals the number of characters enclosed in the backslashes plus the backslashes themselves.

Although the usual practice is to enclose blanks in the string field (e.g., "\ \"), AlphaBASIC permits the use of any characters. Since these characters are never printed, but simply define the size of the field by which a string is to be formatted, non-blank characters serve only as a comment. However, when using several string fields within a single format string, it can be useful to visually separate them from the spaces between the fields by using non-blanks within the backslashes. For example:

```
"\---field1----\     \-----field2-----\   \-field3-\"
```

String fields allow you to define the placement and size of string data. For example, if A$="Now is the time.", then:

```
PRINT USING "As he once said, '\--------------\'",A$
```

produces:

```
As he once said, 'Now is the time.'
```

If the string to be formatted is larger than the string field, BASIC ignores the extra characters. If the string to be formatted is smaller than the string field, BASIC adds trailing blanks to the string to make it the same size as the field, and thus left justifies it in the field.

You may combine string fields and numeric fields within a single format string. (See the section below for information on numeric fields.) For example:

```
 5 STRSIZ 25
10 MAP1 MASK,S,42,"\-10char-\     ####.##     \---15 char---\"
15 C$="(in millions)"
20 PRINT USING MASK,"YEAR 1979",234.556,C$,"YEAR 1980",5678.456,C$
```

produces:

```
YEAR 1979     234.56     (in millions)
YEAR 1980    5678.46     (in millions)
```

NOTE: Remember that the default string size is 10 characters, so you will want to explicitly define any strings over 10 characters via MAP statements or include a STRSIZ statement in your program to adjust the default string size.

13.2.2 The ! Symbol (One-character String Field)

The exclamation mark identifies a one-character string field. BASIC replaces the exclamation mark with a corresponding string. (If the string constant or string variable contains more than one character, BASIC ignores any characters past the first.) For example:

```
10 STRSIZ 40
20 MASK$="The temperature is:   ###!  =  ##!"
30 PRINT USING MASK$,50,"F",10,"C",68,"F",20,"C",86,"F",30,"C",104,"F",40,"C"
```

prints:

```
The temperature is:   50F  =  10C
The temperature is:   68F  =  20C
The temperature is:   86F  =  30C
The temperature is:  104F  =  40C
```

If no string is available to be substituted for the ! symbol, BASIC simply prints the ! symbol instead. For example, if we took our sample program above and removed the first "F" from the PRINT USING expression list, the first line of our display would look like this:

```
The temperature is:   50!  =  10C
```

13.2.3 The # Symbol (Numeric Fields)

The # symbol in a format string always indicates that you want to format numeric data. Each # symbol in a format string represents one numeric digit. The simplest numeric format string would consist of just # symbols. For example:

```
PRINT C USING "####"
```

The statement above tells BASIC to format the numeric variable C into a field of four digits, with no fractional part. If the format string causes BASIC to remove the fractional part of a number, BASIC rounds the number to the next integer, rather than truncating it. For example:

```
PRINT 2367.88 USING "####" RET
2368
```

If the numeric field is too small to contain the specified number (for example, if we had specified the number 650456.56 with the format string "####"), BASIC prints the number in standard BASIC format preceded by a % symbol, indicating overflow. For example:

```
PRINT 150450 USING "####" RET
%150450
```

If the numeric field is larger than the specified number, BASIC right justifies the number in the field, inserting leading blanks into the digit positions not needed. For example:

```
PRINT USING "######",23 (RET)
    23
```

(Four blanks precede the number 23.) Note that other formatting characters discussed below (e.g., the $$ and ** symbols) also define digit positions as well as perform special formatting functions.

NOTE: You cannot format string data with a numeric field format string. If you try to do so, BASIC just prints the format string, indicating that it was unable to format the data. For example:

```
PRINT USING "#####","Hi there" (RET)
#####
```

13.2.4 The Period Symbol (Decimal Point)

You may include one period within a numeric field to specify where a decimal point is to appear in the formatted number. For example:

```
PRINT USING "####.##",2345.502,1100.657,200,3.95
```

produces:

```
2345.50
1100.66
 200.00
   3.95
```

If the number specified contains more digits to the right of the decimal point than the format string, BASIC rounds the number so that it contains the right number of digits in the fractional part. If the format string contains more digits to the right of the decimal point than the specified number, BASIC fills in the unused digit positions with zeros (as in the case of the number 200, above). If the format string specifies any digits in front of the decimal point, BASIC prints at least one digit in front of the decimal point for each number, even if that digit is a zero.

13.2.5 The $$ Symbol (Floating Dollar Sign)

The $$ symbol at the front of a numeric field format string tells BASIC to insert a dollar sign at the front of the formatted number. The double dollar sign symbol defines two digit positions, one of which is taken up by the dollar sign itself.

For example:

```
PRINT USING "$$######.##",17500.66,100,345.2
```

produces:

```
$17500.66
  $100.00
  $345.20
```

Notice the difference between using the double dollar sign to produce a floating dollar sign, and simply using the single non-formatting character "$" in the format string:

```
PRINT USING "$######.##",17500.66,100,345.2
```

produces:

```
$ 17500.66
$   100.00
$   345.20
```

Because you will use the $$ symbol to format data that represents money amounts, you may want to use the floating comma symbol in combination with the $$ symbol. (See the paragraph below for information on this formatting character.)

Remember that you can include non-formatting characters in a format string. In the case above, a single dollar sign is not a formatting character, and so BASIC simply prints it as part of the formatted data. As another example:

```
PRINT USING "###%",23.45,56.78,99.84
```

produces:

```
 23%
 57%
100%
```

In the example above, the "%" symbol is not a special formatting character. As another example:

```
PRINT USING "The telephone number is: (###) ### ####",714,555,1212
```

produces:

```
The telephone number is: (714) 555 1212
```

13.2.6 The Comma Symbol (Floating Commas)

By including a comma in your format string, you tell BASIC to insert a comma every three digits to the right of the decimal point. For example:

```
PRINT 6507501.89 USING "########,.##"
```

produces:

```
6,507,501.89
```

BASIC treats any comma to the right of the decimal point as a non-formatting, printable character. Each comma defines one digit position.

13.2.7 The ** Symbol (Asterisk Fill)

By including a double asterisk symbol at the front of your format string, you tell BASIC to replace any leading blanks that would normally be output in front of a number with asterisks. This is especially useful when printing checks. The double asterisk defines two digit positions. For example:

```
PRINT 231.69 USING "**#######.##"
```

produces:

```
*****231.69
```

NOTE: You will probably use asterisk-fill formatting when printing dollar amounts; remember that you may include a dollar sign symbol in the format string. For example:

```
PRINT 231.69 USING "**$######.##"
```

prints:

```
***$231.69
```

13.2.8 The Z Symbol (Leading Zeros)

To generate leading zeros, include the Z symbol within your format string. The format string must begin with one # symbol followed by a series of Zs. The total size of the formatted string is the number of Zs plus the one # symbol. For example:

```
PRINT 123 USING "#ZZZZZ"
```

produces:

PRINT 123 USING "#ZZ.#
produces: 123.0

PRINT 23 USING "#ZZ.
produces: 023.0

```
000123
```

13.2.9 The Minus Symbol (Trailing Minus Sign)

You may cause the sign of a number to be printed following the number by ending a numeric field in a format string with a minus sign. If the number is positive, BASIC prints a blank after the number; if it is negative, BASIC prints a minus sign after the number. For example:

```
10 MAP1 MASK,S,26,"\--7--\        $$######.##-"
20 C$="Credit:" : D$="Debit:"
30 PRINT USING MASK,C$,345.67,D$,-567.89,C$,100.89,D$,-3456.33
```

produces:

```
Credit:        $345.67
Debit:         $567.89-
Credit:        $100.89
Debit:        $3456.33-
```

13.2.10 The ^^^^ Symbol (Exponential Format)

You may specify exponential format by following the numeric field in a format string with four circumflexes (^^^^). These symbols define the spaces taken up by the "E nn" exponent characters. BASIC left justifies the significant digits, adjusting the exponent as necessary. (As with other numeric formats, BASIC allows any decimal point arrangement.) For example:

```
10 PRINT USING ".#####^^^^",100,2345.66,5000,.0004
```

prints:

```
.09999E+03
.23456E+04
.50000E+04
.39999E-03
```

13.3 FORMATTING EXAMPLES AND HINTS

All of our examples above used the PRINT statement to print formatted data. Remember that you may also format a value without displaying it by using the USING modifier without the PRINT statement. For example:

```
A$ = B USING C$
```

The statement above formats the number in B using the format string in C$, and leaves a string result in A$. (NOTE: This format of the USING modifier

is only for formatting numeric data. Also note that even though we are formatting numeric data, the result is always a string.) This type of format allows you to create headings and image lines that you use more than once, and to inspect and manipulate formatted data before printing it.

You may not use the USING modifier recursively. That is, you may not use a format string that is itself the result of a USING modifier. (For example, if you have specified C$ = B USING "###.##", you may not say: N$ = D USING C$.)

When using the PRINT USING format, remember that PRINT USING differs from the regular PRINT statement in that the use of semicolons to separate the elements of the print list has no effect on the spacing of those formatted elements.

Below is a sample program that uses the USING modifier to format output into a small report. It also demonstrates the use of subroutines, MAP statements, and file-handling.

```
5   ! Tiny report generator
10  STRSIZ 100
20  MAP1  HEADING,S,49,   "     \---10---\        \---10---\       \---10---\"
30  MAP1  MASK,S,54,      "     \---10---\     $$#######,.##      #ZZZZZZZZZ"

40  ! Main Program
50  GOSUB INSTRUCTIONS          ! Display Instructions.
60  OPEN #1,"REPORT.DAT",OUTPUT ! Open file to hold report.
70  GOSUB GET'HEADER            ! Get and write header for report.
80  I = 1                       ! Initialize line counter.
90  GOSUB WRITE'REPORT          ! Get and write data to report.
100 CLOSE #1                    ! Close out file.
110 END

200 INSTRUCTIONS:  ! Display instructions
210    PRINT "       Welcome to the Mini Report Generator" : PRINT
220    PRINT "We will first ask you to enter three titles (max 10 char-"
230    PRINT "acters each). These will form the heading of your report."
240    PRINT "Then we'll ask for each line of the report." : PRINT
250    PRINT "  Field #1 is a string (maximum of 10 characters."
260    PRINT "     Enter zero to end report.)"
270    PRINT "  Field #2 is a number (maximum of 7 characters) to"
280    PRINT "     be expressed as a dollar amount.  Don't enter commas.)"
290    PRINT "  Field #3  is a number  (maximum of 10 characters)"
300    PRINT "     that can represent any non-dollar data." : PRINT
310    RETURN

400 GET'HEADER: ! Input and write header to file.
410   INPUT "Enter Title #1: ",TITLE1$
420   INPUT "Enter Title #2: ",TITLE2$
430   INPUT "Enter Title #3: ",TITLE3$
440   ! Write header to file.
450   PRINT #1, USING HEADING,TITLE1$,TITLE2$,TITLE3$ : PRINT #1
460   RETURN

500 WRITE'REPORT: ! Input and write data to file.
510    PRINT : PRINT "Line #";I;"--"          ! Keep track of number of lines.
520    INPUT "   Enter Field #1: ",FIELD1$
530    IF FIELD1$="0" THEN RETURN
540    INPUT "   Enter Field #2: ",FIELD2
550    INPUT "   Enter Field #3: ",FIELD3
560    PRINT #1, USING MASK,FIELD1$,FIELD2,FIELD3
570    I = I+1
580    GOTO WRITE'REPORT
```

We can use the program to generate very different types of reports. For example:

ITEM	COST/UNIT	PART NO.
Axle, Half	$349.67	0000002376
Cntrl Box	$45.67	0000002985
K27 Engine	$1,289.45	0000005678
Shaft #2	$32.56	0000005645

or:

EMPLOYEE	SALARY	PAYROLL#
R. Smith	$239,234.33	0000000654
J. Swann	$34,123.78	0000000834
L. Knowles	$18,345.43	0000000235
T. Filbert	$1,203,456.77	0000000263

13.4 EXPANDED TAB FUNCTIONS

The TAB function in AlphaBASIC has been expanded beyond the normal usage to include terminal screen handling, such as cursor control and other special functions. To be used only in a PRINT statement, the TAB function operates in the traditional manner when supplied with only a single numeric argument such as TAB(X). In this case the function causes the carriage to be positioned over to the "X" column on the current line. When supplied with two arguments such as TAB(R,C), however, the TAB function performs special CRT functions.

If the value of R is positive, the R,C arguments are treated as (row,column) coordinates for positioning the cursor on the terminal screen. The specified characters are then printed beginning in that position. As in other functions, the R and C arguments may be expressions. Terminals are assumed to begin with row 1 (top of screen) and column 1 (left end of each row). If you use TAB for cursor positioning, remember to follow the TAB function with a semicolon (e.g., PRINT TAB(23,5);)-- otherwise, BASIC will output a carriage return/linefeed after it positions the cursor, thus destroying your careful positioning.

If the value of R is -1, the function is interpreted as a special terminal command and the appropriate command code must be specified as the C argument. The codes are transmitted to the terminal driver (TDV file in DSK0:[1,6]), which does the actual interpretation and performs the special function for your terminal. The following list gives the standard decimal codes in use for all the terminal drivers supported by Alpha Micro:

Code	Function
0	Clear screen and set normal intensity
1	Cursor home (move to 1,1 - upper left corner)
2	Cursor return (move to column 1 without line-feed)
3	Cursor up one row
4	Cursor down one row

5	Cursor left one column
6	Cursor right one column
7	Lock keyboard
8	Unlock keyboard
9	Erase to end of line
10	Erase to end of screen
11	Enter background display mode (reduced intensity)
12	Enter foreground display mode (normal intensity)
13	Enable protected fields
14	Disable protected fields
15	Delete line
16	Insert line
17	Delete character
18	Insert character
19	Read cursor address
20	Read chracter at current cursor address
21	Start blinking field
22	End blinking field
23	Start line drawing mode
24	End line drawing mode
25	Set horizontal position
26	Set vertical position
27	Set terminal attributes

The actual routines that perform the screen controls are in the specific terminal drivers and not in AlphaBASIC itself. Not all terminal drivers have all of the functions above simply because not all terminals are able to perform all of these functions. If your terminal has additional features, Alpha Micro recommends starting at 64 (decimal) when you assign function codes in your terminal driver.

CHAPTER 14

SCALED ARITHMETIC

AlphaBASIC uses a floating point format which gives an accuracy of 11 significant digits. Unfortunately, this accuracy is absolute only when dealing with numbers that are total integers (i.e., there are no numbers to the right of the decimal point). This fact stems from the conversions that are required from decimal input to the binary floating point format used in the hardware. For most business users, the actual range of numbers contains two digits to the right of the decimal point and nine digits to the left of the decimal point. When the fractional part of the number is converted between decimal and binary formats, a small but significant error is sometimes introduced which may propagate into inaccuracies when dealing with absolute dollars-and-cents values.

As an example of the kinds of inaccuracies that can occur, take a look at the following program:

```
10 SIGNIFICANCE 11
20 PRINT .001
```

Instead of the expected answer of .001, we see the answer:

```
9.9999999999E-4
```

This is not an error in BASIC, but simply represents the side effects of converting a decimal fraction to binary representation and back again. Some decimal fractions cannot be exactly expressed as a binary fraction in a finite number of digits, and so round-off error occurs.

The error was only visible because our program set the number of significant digits to 11. (The usual number of significant digits is six.) Such errors can accumulate and present themselves when you do a large number of multiplications and divisions using decimal fractions.

AlphaBASIC incorporates a scaling feature which helps to alleviate this problem by storing all floating point numbers with a scale offset. This offset designates where the 11 absolute accuracy digits are located in relation to the decimal point. BASIC does this by multiplying every input number by the scaling factor and then dividing it out again before printing.

(This is a simplified explanation, and many other checks and conversions are done internally to scaled numbers.)

The scaling factor represents the number of decimal places that the 11-digit "window" is effectively shifted to the right in any floating point number. For example, the most common application is in a business environment where the scaling factor of 2 would be used to give absolute 11 place accuracy to numbers which extend 2 places to the right of the decimal point. This means that the value of 50.12 is multiplied by the scaling factor of 2 digits (100) and stored as the floating point value of 5012. Since this value is an integer, it has absolute accuracy. Just before printing, BASIC divides this number by the scaling factor to reduce it to its intended value of 50.12.

Other conversions have been included into the system to take care of all the little subtle effects of storing scaled numbers. For example, when converting scaled numbers to integer or binary format, BASIC must unscale the number first before converting it. When BASIC multiplies two scaled numbers together, the result is a number which must be unscaled once, while division of two scaled numbers creates exactly the opposite problem. Dealing with scaled numbers for exponential, logarithmic and trigonometric functions creates even more exotic problems. All these conversions are done automatically by AlphaBASIC, so you are relieved of the programming task of keeping track of them.

14.1 SCALE

Scaled arithmetic is normally entered at the start of a program and continues in effect throughout the program. The statement for setting the program into scaled mode is:

```
SCALE n
```

The scaling factor "n" must be a decimal digit in the range of -30 to +30. It may not be a variable, since scaling is done at compile time for constant values as well as at run-time for input and output conversions. Negative scaling moves the 11-digit window to the left. NOTE: You won't often use a negative scaling factor, since that takes care of the case where your numbers are too large, rather than too small. For example:

```
PRINT 100000000000000000
```

produces the number:

```
9.9999999999E16
```

when SIGNIFICANCE is set to 11 because AlphaBASIC cannot handle a number that large with eleven significant digits. However, if you use SCALE -1, you get the expected answer of 1E17, since you have adjusted the number to the range that AlphaBASIC can handle.

A few words of caution are in order here. Once BASIC detects the SCALE statement during compilation, BASIC scales all constant values that follow by the scaling factor so that they are stored properly. In addition, a run-time command is generated in the executable program which causes the actual scaling to be performed on INPUT and PRINT values when the program is running. If two or more different SCALE statements are executed in the same program, some very strange results may come out unless you are totally familiar with what is happening with compile-time and run-time conversions. We suggest that you play with this one a bit before delving into it full steam.

If you are using a positive scaling factor to adjust real numbers, note that using SCALE does nothing to prevent inaccuracies if the scale factor you use is not large enough to cause AlphaBASIC to handle your data as integers. For example, if you want to handle numbers that have three digits to the right of the decimal point, a scaling factor of 2 will leave one digit to the right of the decimal point, and scaling error can still occur. So, if you will be using numbers with a fractional part of two digits, use a scaling factor of 2; if the fractional part will be three digits, use a scaling factor of 3; and so on.

One other word of caution. Floating point numbers that are stored in files by the sequential output PRINT statement are unscaled and output in ASCII with no problems. Floating point numbers that are written to random access files by using the WRITE statement are not unscaled first; any program that reads this file as input must either be operating in the same scaling mode in which the data was written, or else must apply the scale factor explicitly to all values from the file. Binary and string values, of course, are never modified, regardless of the scaling factor currently in use.

CHAPTER 15

ALPHABASIC FILE I/O SYSTEM

This chapter contains information on creating and using disk files from within your BASIC programs. Since these processes differ somewhat depending on whether you want to use sequential or random data files, we discuss sequential and random files generally before getting into the specific commands you can use to manipulate these files. Note the sample program at the end of the chapter; it demonstrates defining a logical record, computing the logical record blocking factor for a random file, allocating a random file, opening and closing a random file, searching for a file, and writing and reading data to and from a random file.

AlphaBASIC supports both sequential and random access disk files. You may write data either in ASCII or in packed binary formats. Files that AlphaBASIC programs create are compatible with all other system utility formats, and BASIC files may be interchanged with files from other languages. That is, BASIC data files can be read and manipulated by programs written in other languages. Conversely, files created by other languages and system utilities may be read and manipulated by programs written in AlphaBASIC.

Files are created and referenced by the general statements OPEN, CLOSE, INPUT, INPUT LINE, PRINT, READ, and WRITE. All file references are done by a file-channel number, which may be any integer value from 0 to 65535. You might think of the file-channel number as designating an information channel. Once a file has been associated with it, the file channel serves as a pipeline through which data can be transferred between your program and the file. Once you close that file, the file channel is no longer associated with it, and you may open another file on that file channel. You may never have two files open at the same time with the same file channel. The file channel always follows the verb in any file I/O statement and may be any numeric expression which is preceded by a pound sign (#). File channel zero is defined as your terminal, and is legal in file statements to allow you to write generalized programs which may selectively output to either a file or to the terminal at run-time.

There is no absolute limit to the number of files that may be open at any given time in a program, but since each file requires a certain amount of memory, there is a practical limit to this number based on memory available in your partition.

BASIC automatically closes all open files when the program exits or when a CHAIN statement is executed, if the files have not already been explicitly closed via a CLOSE statement. BASIC cannot open two files with the same file-channel number at the same time, but after BASIC closes a file, another file may be opened using the same file-channel number. All file statements are valid as direct statements, but BASIC closes any open files before it executes another RUN command. This prevents statements in an executing program from reading or writing to files which were opened by a direct statement. Under the current version of AlphaBASIC, each open file requires about 580 bytes of free memory for buffers and control blocks.

15.1 SEQUENTIAL ASCII FILES

Sequential disk files are the easiest to understand and implement in AlphaBASIC. BASIC writes data to a sequential file in ASCII format, and stores numeric data as ASCII string values. A sequential data file usually has the extension .DAT unless you explicitly order otherwise in the OPEN statement that opens that file. NOTE: Sequential files may not contain non-ASCII data (e.g., binary or floating point data). Therefore, you may only use PRINT, INPUT and INPUT LINE for transferring data to and from sequential files. (Remember that PRINT converts floating point and binary data to printable (ASCII) form.) The READ and WRITE statements do not convert data to ASCII form, and so are used for transferring data to and from random files.

The sequential data files are normal ASCII files in all respects, and you may manipulate them by using the system text editors, the printer spooler, or any of the other system utilities.

To open a sequential file, use the OPEN statement, specifying either INPUT or OUTPUT mode.

Use the PRINT statement (followed by a non-zero file-channel number) to write data to sequential files. The PRINT statement automatically appends a carriage return/line-feed to your data in the same manner that it does when sending data to a terminal display. (See Section 10.19 for information on using commas and semicolons to format PRINT statement output.)

Use the INPUT or INPUT LINE statements (followed by a non-zero file-channel number) to read data from a sequential file. Remember that the INPUT statement reads one piece of data for each variable specified, while the INPUT LINE statement (if you specify a string variable) reads into the specified string variable the entire line of ASCII data up to (but not including) the carriage return/line-feed at the end of the line. INPUT and INPUT LINE work exactly the same for files as they do for terminal input except that you omit a prompt string and must include a file-channel number.

Sections 15.3.7 and 15.3.8 talk about INPUT and INPUT LINE. (Also, see Sections 10.11 and 10.12 for more information on INPUT and INPUT LINE.)

15.2 RANDOM FILES

Random access, or direct access, files are more complex than sequential files, but offer a much more flexible method for storing and retrieving data in different formats. Random files are written in "unformatted" or packed data mode. Random file disk blocks are contiguously allocated on the disk. The major advantage of random files over sequential files is the flexibility with which you may access data in a random file. You may only open a sequential file for input or output, but you may open a random file for input and output simultaneously. Accessing data in a sequential file requires that you step through the file record by record. In the case of a random file, however, you may access any record without referring to any other record in that file. In addition, random files can contain data in any format supported by AlphaBASIC (unlike sequential files, which may only contain ASCII data).

15.2.1 Logical Records

All program accesses to random files are made via the "logical record" approach. A logical record is defined as a fixed number of bytes whose format is explicitly under control of the program performing the access. Physical blocks on the disk are each 512 bytes long, and each random file must be preallocated as some given number of these 512-byte blocks. Logical records may be any length from 1 byte to 512 bytes. (Logical records can never overlap physical disk blocks.) The AlphaBASIC I/O system automatically computes the number of logical records that fit into one disk block, and performs the blocking and unblocking functions for you. For example, if your logical record size is defined as 100 bytes, then each block on the disk contains 5 logical records with the last 12 bytes of each block being unused. Therefore, the most efficient use of random files comes when the logical record size is a power of 2; that is, it divides evenly into 512 bytes (32, 64, 128, etc.).

15.2.2 Blocking Factor and Record Size

Random access files are preallocated once, using the ALLOCATE statement, which gives the number of physical 512-byte blocks to allocate. It is up to you to calculate the maximum number of logical records required in the file, and then to calculate how many disk blocks are required to completely contain the number of logical records you desire. For instance, assume the logical record size is 100 and you need a maximum of 252 logical records in your file. Each disk block is 512 bytes, and therefore contains 5 logical records. You need 252 logical records, so dividing 252 by 5 gives 50 full disk blocks plus 2 logical records remaining. Since the file must be

allocated in whole disk blocks, you need 51 blocks, which gives you a maximum of 253 logical records. These logical records are referenced in your program as records 0 through 252, since the first record of any random file is record 0, unless you have used FILEBASE. (See Section 10.6, "FILEBASE.") (NOTE: When your record size does not divide evenly into 512 bytes, it is a good idea to consider expanding it so that it does. This is for two reasons: 1) you will be using the same number of physical disk blocks whether or not you expand the record size, so you're not saving anything by not doing so; and 2) this leaves you room for future expansion of the data in the record.)

When you are opening a random file, you must specify the logical record size in the OPEN statement (also specifying RANDOM mode); it is possible to get things fouled up if you do not have the record size correct. No logical record size is maintained within the file structure itself. This fact does make it nice in one respect; a file which is accessed by many programs can have its record size expanded without recompiling all the accessing programs. Here is how: Assume (as an example) that you have a file which is considered the parameter descriptor file for all other files in the entire system. This file gives the record size as 100 bytes for the vendor name and address file. All programs which reference the vendor file first read this parameter file to get the size of the vendor file logical record. The programs then set the size into a variable and use this variable in the OPEN statement for the record size. Each READ or WRITE statement then manipulates the 100 bytes of data by reading or writing to or from variables whose size totals 100 bytes. Let's say you now want to expand the file to 120 bytes and that most of the programs do not have to make use of the extra 20 bytes until some time in the future. You write a program which copies the 100-byte file into a new 120-byte file and then you update the main parameter file to indicate that the new record size for the vendor file is 120 bytes instead of 100. Each program now opens the file using the new 120-byte record size (since it is read in from the parameter file at run-time), but only READs or WRITEs the first 100 bytes of each record due to the variables used by the READ and WRITE calls.

15.3 FILE I/O STATEMENTS

Later sections in this chapter show you the general format of each of the file I/O statements and give detailed examples of their uses.

Although you will want to read each of those sections carefully, we'd like to give a summary here of how to create and use sequential and random files. Remember that the steps below are only suggestions, and you may want to omit or add steps.

USING SEQUENTIAL FILES FOR OUTPUT:

1. Use the LOOKUP command to see if the file already exists.

   When you output to a sequential file, you are creating a brand new file. If a file of the same name and extension already exists in

the account you are writing to, BASIC automatically deletes the old file for you before it opens the new output file. Therefore, if you don't want BASIC to delete an existing file, be sure to use the LOOKUP command before you open a file for output to make sure that such a file does not already exist.

2. If the file already exists, you can go ahead and open it (if you want BASIC to delete the existing file for you) or you can choose another file name and use the LOOKUP command again to see if that file already exists.

3. Use the OPEN statement to open the file for OUTPUT.

4. Begin using PRINT statements (specifying the file-channel number associated with the file by the OPEN statement) to write data to the file.

5. When finished, use the CLOSE statement to close the file.

USING SEQUENTIAL FILES FOR INPUT:

1. Use the LOOKUP command to see if the file already exists. (If it doesn't exist, you cannot input data from it.)

2. Use the OPEN statement to open the file for INPUT.

3. Begin using INPUT LINE or INPUT statements to read data from the file (specifying the file-channel number associated with the file by the OPEN statement).

4. Check the EOF function after each input to make sure you haven't read beyond the end of the file.

5. When finished, use the CLOSE statement to close the file.

USING RANDOM FILES FOR INPUT/OUTPUT:

1. Use the LOOKUP command to see if the file already exists. If it does, you can skip down to step #3.

2. If the file doesn't exist, you must create it. First, decide what size the logical records will be (in decimal bytes). Then compute the blocking factor as discussed in Section 15.2.2, "Blocking Factor and Record Size." Use the ALLOCATE command to create the file with the number of disk blocks needed.

3. Use the OPEN statement to open the file for RANDOM processing. Specify the size of the logical records in the file, and the record-number variable that will hold the number of the logical record you are currently accessing.

4. Use READ and WRITE statements (specifying the file-channel number associated with the file by the OPEN statement) to read and write data in the file. Remember to change the record-number variable to the correct record number before performing each read or write operation so that you access the logical record you want. Make sure that the record-number variable contains a valid record number before performing the file I/O.

5. When you are finished reading and writing the file, use the CLOSE statement to close the file.

15.3.1 OPEN

You must open a file before you can transfer data to or from the file. The OPEN statement assigns a unique file-channel number to a file and also specifies the name that is either to be given to an output file, or to be used in locating an input file. The general format is:

```
OPEN #file-channel, filespec, mode, {record-size, record#-variable}
```

file-channel — Any numeric expression which evaluates to an integer from 0-65535 (0 is defined as the user terminal and treated as such).

filespec — Any string expression which evaluates to a legal file description. May be a string variable or string literal. (If it is a string literal, remember to enclose it in quotation marks.)

mode — Specifies the mode for opening the file:

INPUT -	Opens an existing sequential file for input operations.
OUTPUT -	Creates a sequential file for output operations.
RANDOM -	Opens an existing random file for random read/write.
INDEXED -	Opens an ISAM data file and primary index file.
INDEXED'EXCLUSIVE -	Opens an ISAM data file and primary index file for exclusive access.

The remaining two options must be used for RANDOM, INDEXED and INDEXED'EXCLUSIVE modes only:

Record-size — An expression which dynamically specifies at run-time the logical record size for read/write operations on the file.

Record#-variable — A non-subscripted numeric variable which must contain the record number of the desired random access for READ or WRITE statements when they are executed. It must be a floating-point variable.

Any attempts your program makes to read or write to a file which has not been opened result in the error message IO to unopened file in line nnn, and the program is aborted. The filespec string may be as brief as the name of the file, in which case it is assumed to have an extension of .DAT and to reside in your disk account. The filespec string may be a complete file specification, if you desire, giving the explicit location of the file, which may be in another disk account or even on another disk drive. Some examples are:

```
OPEN #1, "DATFIL", INPUT
OPEN #15, "PAYROL.TMP", OUTPUT
OPEN #A, C$, OUTPUT
OPEN #3, "DSK1:OFILE.ASC[200,20]", OUTPUT
OPEN #1, "VENDOR.DAT", RANDOM, 100, RECNUM
OPEN #1+X, MID$(A$,2,3), OUTPUT
OPEN #25,"MASTER",INDEXED,80,RELKEY
```

The OPEN statement is one of the only statements which reference the file by its actual ASCII filespec in the standard operating system format. Most references in the program are made to the file-channel number which is assigned in the OPEN statement #file-channel.

15.3.2 CLOSE

The CLOSE statement ends the transfer of data to or from a file. Once a file has been closed, no further references are allowed to that file until another OPEN statement for that file is executed. Any files that are still open when the program exits are closed automatically. The format of the CLOSE statement is:

```
CLOSE #file-channel
```

where #file-channel specifies the file-channel number associated with the file you want to close. For example, if you have previously opened a file VENDOR.DAT:

```
OPEN #3, "VENDOR.DAT[200,1]",INPUT
```

to close that file, use the statement:

```
CLOSE #3
```

15.3.3 KILL

The KILL statement erases one file from the disk. It does not need a file-channel number and no OPEN or CLOSE need be performed to KILL a file. The format for the KILL statement is:

```
KILL filespec
```

For example:

```
KILL "NEWDAT.DAT"
```

As in the OPEN statement, the filespec is any string expression which evaluates to a legal file description. KILL assumes an extension of .DAT. If you try to erase a file that does not exist, you see the error message:

File not found

You may not erase a file that exists in an account outside of the project you are logged into. For example, if you are logged into account [110,2] and the program you are running tries to kill a file in account [200,1], you see a protection violation error message.

15.3.4 LOOKUP

The LOOKUP statement looks for a file on the disk and returns a flag which tells you if the file was found, and if so, how many disk blocks it contains. The format for the statement is:

```
LOOKUP filespec, result-variable
```

As in the OPEN statement, the filespec is any string expression which evaluates to a legal file description. The result-variable is any legal floating point variable which receives the result of the search. The LOOKUP result-variable may return:

0	File was not found
Positive #n	File was found; it is a sequential file, and contains n disk blocks.
Negative #n	File was found; it is a random file, and contains n disk blocks.

Remember that the number returned by LOOKUP is the number of physical disk blocks used by the file. You must multiply this number of 512-byte blocks by the file's blocking factor to find out how many logical records your file contains. For example, after we execute:

```
LOOKUP "CNURT.DAT",BLOCKS
```

the variable BLOCKS contains the number of disk blocks in the file CNURT.DAT, or a 0 if the file does not exist. We must multiply BLOCKS by the blocking factor of the file to see how many logical records can fit in the file.

15.3.5 ALLOCATE

The ALLOCATE statement preallocates a random file on the disk, which you may then open for random processing. An attempt to allocate a file which already exists results in an error message. A random file need only be allocated once and may then be opened for random read/write operations as many times as desired. The statement format is:

```
ALLOCATE filespec, number-of-blocks
```

As in the OPEN statement, the filespec is any string expression which evaluates to a legal file description. The number-of-blocks is a floating point expression which represents the number of physical 512-byte disk blocks to be allocated to the file. For example:

```
ALLOCATE FILE$, BLOCKS
ALLOCATE "NEW.DAT",20
```

15.3.6 FILEBASE

During normal operation, BASIC refers to the first record in a random file as record number zero (i.e., you set the record number variable to zero to access the first record in the file). In some applications you may want BASIC to refer to this first record by some number other than zero: for instance, to allow you to use zero to flag some special condition, such as a deleted record. The FILEBASE command allows you to set the number used to refer to the first record. For example:

```
FILEBASE 1
```

tells BASIC that the first record in the file is record number one, not record number zero. You may use any numeric argument with FILEBASE.

Note that FILEBASE does not associate its value with a file, but only takes effect when you execute the program it is in. If one program uses a FILEBASE command when referencing a file, all other programs which reference that file should also use a FILEBASE command with the same value

15.3.7 INPUT

Once a sequential file has been opened for input, you may use a special form of the INPUT statement to read data from the file. The INPUT statement uses a file-channel number corresponding to the file-channel assigned in the OPEN statement. The variables in the list may be either numeric or string variables, but must follow the format of the data in the file being read. (Weird results occur if you attempt to read string data into a numeric variable, or vice-versa.) The general format of the INPUT statement is:

```
INPUT #file-channel,variable1{,variable2,...variableN}
```

During the reading of the input data into the variable list, all leading spaces are bypassed unless they are enclosed within quotes, just as in the normal form of the INPUT statement. Also, all carriage-returns and line-feeds are bypassed, allowing the file created by the PRINT statements to contain formatted line data if desired. Commas, spaces and end-of-line characters all terminate numeric data strings and then are bypassed. For more information on INPUT, see Section 10.12. Also, see the section on INPUT LINE, below.

15.3.8 INPUT LINE

After a sequential file has been opened for input, the data can be read from the file by a special form of the INPUT LINE statement which uses a file-channel number corresponding to the file channel assigned in the OPEN statement. The variables in the list may be either numeric or string variables, but must follow the format of the data in the file being read. Unpredictable results occur if you attempt to read string data into a numeric variable, or vice-versa. The general format of the INPUT LINE statement is:

```
INPUT LINE #file-channel,variable1
```

The INPUT LINE statement operation is identical to that of the INPUT statement with the exception that input into a string variable accepts the entire line up to but not including the carriage-return and line-feed that ends the line. This allows commas, quotes, blanks and other special characters to be input. Also, INPUT LINE accepts blank lines as input. The INPUT LINE statement may be used in sequential file processing as well as the standard terminal INPUT statement. You will usually use INPUT LINE specifying one string variable to read in one line of the file at a time. See Section 10.13 for more information on INPUT LINE.

15.3.9 PRINT

Once you have opened a sequential file for output, you will write data to it with a special form of the PRINT statement using a file-channel number which corresponds to the file channel assigned in the OPEN statement. All the techniques available to you when you use the normal form of the PRINT statement (which outputs to the terminal) are also available for sending data to a file, including PRINT USING for formatted data. PRINT writes data to the file in the same format as it would appear if you used PRINT to send the data to a terminal display (i.e., if you left off the file-channel number). Here is the format and some examples of the PRINT statement.

```
PRINT #file-channel, expression-list

PRINT #1; A; B; C
PRINT #4, USING A$, A, SQR(A)
PRINT #Q1, USING "###.##", A1(10);
PRINT #1, "THIS IS A SINGLE LINE"
PRINT #2, "WRITE TO","PRINT ZONES"
```

For more information on PRINT, see Section 10.19

15.3.10 READ

The READ statement reads a selected logical record from a random file which has been opened for random access processing. The logical record which is transferred by the system I/O is the one whose record number is currently in the record-number variable mentioned in the OPEN statement. The format of the READ statement is:

```
READ #file-channel,variable1{,variable-2...,variableN}
```

The variables in the list may be any format, but they obviously should match that of the designated record format. The data is read into the variables as unformatted bytes, without regard to variable type. The data is transferred into each variable until the variable has been completely filled. Then the next variable in the list is filled, and so on. If the record is longer than the variable list specifies, all excess data in the record will not be transferred. An attempt to transfer more data than is in the logical record size results in an error message. The most efficient use of the random files comes when the variable or variables used are mapped by the MAP statement to the exact picture of the record format in use. (See Chapter 8, "Memory Mapping System," for information on MAP statements.) Also see the sample program at the end of this chapter for a demonstration of creating and reading a random file.)

15.3.11 WRITE

The WRITE statement is used to write a selected logical record into a random file which has been opened for random access processing. The logical record which is transferred by the system I/O is the one whose record number is currently in the record-number variable mentioned in the OPEN statement. The format of the WRITE statement is:

```
WRITE #file-channel,expression-list
```

The variables in the list may be any format, but they obviously should match that of the designated record format. The data is written into the logical record from the user variables as unformatted bytes, without regard to variable type. The data is transferred from each variable until the variable has been completely emptied. Then the next variable in the list is used, and so on. If the record is longer than the variable list specifies, all excess data in the record will not be modified. An attempt to transfer more data than is in the logical record size results in an error message. The most efficient use of random files comes when the variable or variables used are mapped by the MAP statement to the exact picture of the record format in use.

15.4 SAMPLE PROGRAM

The program below gives a very simple demonstration of limited data manipulation. Notice that you could easily write modules that would expand its functions to include deleting customer records, changing data in existing customer records, adding more customer records to a partially filled file, and so on.

Some of the file-handling commands demonstrated by the program are: LOOKUP, ALLOCATE, OPEN, CLOSE, READ, and WRITE. Notice that we also use the extended TAB functions to clear the screen and position the cursor, and use the MAP statement to define all logical records and variables used in the program.

```
5    ! SAMPLE PROGRAM TO CREATE AND ACCESS A RANDOM FILE
10   !
15   ! This program simulates a very simple information management system.
20   ! Notice that we use MAP statements to map all variables used in the
25   ! program; although this is not strictly necessary (except for the
30   ! definition of the Control record and Logical record templates), it
35   ! is handy to have all variables defined at the front of the program.
40   !
45   ! Define Control record that contains info about file
50   MAP1   HEADER'RECORD
55     MAP2   TOTAL'RECS,F              ! Total number of records in file
60     MAP2   IN'USE,F                  ! Number of records in use
65     MAP2   FILLER,S,52               ! Filler bytes needed to pad record
70                                      ! to 64 bytes
75   ! Define logical record (64 decimal bytes)
```

```
80   MAP1   CUSTOMER'INFO
85     MAP2   NAME,S,20                  ! Name and address
90     MAP2   STREET,S,10
95     MAP2   CITY,S,11
100    MAP2   STATE,S,2
105    MAP2   ID'NUM,F                   ! Customer ID number
110    MAP2   CAR'INFO
115      MAP3   MODEL,S,10               ! Information about car
120      MAP3   YEAR,S,4
125      MAP3   INSURANCE,B,1            ! Does owner have insurance?
130  !
135  ! Miscellaneous variables used by the program
140  MAP1 BLOCKS,F                       ! # of disk blocks used by file
145  MAP1 BYTES,F                        ! # of bytes used by all records
150  MAP1 REC'SIZE,F,6,64                ! # of bytes in record (64, decimal)
155  MAP1 REC'NUM,F                      ! Contains current record numbe
160  MAP1 RESULT,F                       ! LOOKUP command result variable
165  MAP1 QUERY,S,3,""                   ! Scratch variable (init to null)
170  MAP1 NVAL,F                         ! Scratch variable for user input
175  !
180  !                    BEGIN MAIN PROGRAM
185  ! Use LOOKUP command to see if file already exists.  If it does, go to
190  ! routine that will read information from the file; otherwise, create
195  ! file. First, ask user for total number of records we can write to file.
200  ! Then, see how many bytes this requires (64*TOTAL'RECS). We can fit
205  ! exactly 8 records per disk block (512=8*64). If can't fit even number
210  ! of records per block, allocate one extra block.  Now that we know how
215  ! many disk blocks to allocate, ALLOCATE and OPEN the file.
220  START:
225     LOOKUP "CUSTMR.DAT", RESULT : IF RESULT <> 0 GOTO READ'FILE
230     PRINT TAB(-1,0); TAB(10,1);      ! Clear screen; position cursor
235     INPUT "Enter total number of file records: ", TOTAL'RECS
240     BYTES = TOTAL'RECS * REC'SIZE : BLOCKS = BYTES/512
245     IF BLOCKS <> INT(BYTES/512) THEN BLOCKS = FIX(BLOCKS) + 1
250     ALLOCATE "CUSTMR.DAT", BLOCKS
255     OPEN #2, "CUSTMR.DAT", RANDOM, REC'SIZE, REC'NUM
260     ! Write initial file header to Record 0 (REC'NUM = 0).  File header is
265     ! control record that tells us how many records are in file
270     ! (TOTAL'RECS) and, of those, how many are in use (IN'USE).
275     REC'NUM = 0 : IN'USE = 0 : WRITE #2, HEADER'RECORD
280     REC'NUM = 1                      ! Get ready to write to next record
285     ! Clear screen and position cursor
290     PRINT TAB(-1,0); "Entering info..."; TAB(10,1)
295     PRINT "When you are through, enter a RETURN for Customer Name."
300     GOSUB GET'INFO                   ! Get info and write it to file
305  !
310  !           READ INFORMATION FROM EXISTING FILE
315  ! Open file for input. Get control record to see how many records are
320  ! in use.  Ask user if wants to read from file; if not, exit.
325  ! Check to see if existing file is empty; if so, exit.  Tell the
330  ! user what customers we have info for; ask which customer user wants
335  ! info on (1 customer, ALL, or none).  Check to make sure user enters
340  ! valid customer number.  Just a RETURN (=null) means user wants to quit.
```

```
345 ! Display desired info until user enters a RETURN to quit.
350 READ'FILE:
355    OPEN #3, "CUSTMR.DAT", RANDOM, REC'SIZE, REC'NUM
360    REC'NUM = 0 : READ #3, HEADER'RECORD
365    PRINT : INPUT "Do you want to read file (Y or N)? ",QUERY
370    QUERY = UCS(QUERY) : IF (QUERY = "N") GOTO READ'EXIT
375    ! Clear screen and position cursor
380    PRINT TAB(-1,0); "Reading file..."; TAB(10,1)
385    IF (IN'USE = 0) THEN PRINT "File is empty" : GOTO READ'EXIT
390    ! Show user what customers we have info on
395    PRINT "Here is a list of the customers for whom we have info:"
400    FOR REC'NUM = 1 TO IN'USE
405       READ #3, CUSTOMER'INFO : PRINT
415       PRINT "CUSTOMER #:"; ID'NUM; SPACE(5); "CUSTOMER NAME: "; NAME
420    NEXT REC'NUM
425 !
430 ! Find out what info we should display
435 GET'NUM:
440    QUERY= "" : PRINT                  ! Set initial choice to null
445    PRINT "Enter the ID number of the customer whose info you want to"
450    PRINT "see. (Enter just a carriage return to end program; enter 'ALL'"
455    INPUT "to see info for all customers.): ", QUERY
460    IF (QUERY="") GOTO READ'EXIT       ! User wants to quit
465    QUERY = USC(QUERY) : IF (QUERY = "ALL") THEN GOTO DISPLAY'ALL
470    ! Check to see that customer number is valid.  Convert string to
475    ! numeric so that we do numeric, not string, comparison.
480    NQUERY = VAL(QUERY)
485    IF (NQUERY < 1 OR NQUERY > IN'USE) THEN &
           PRINT : PRINT "Invalid number." : GOTO GET'NUM
490    ! Read desired record (set REC'NUM to customer number).
495    REC'NUM = NQUERY : READ #3, CUSTOMER'INFO
500    GOSUB DISPLAY'INFO                 ! display record
505    GOTO GET'NUM                       ! See if user wants to see another
!
510 ! User wants to display all customer records
515 DISPLAY'ALL:
520    FOR REC'NUM = 1 TO IN'USE
525       READ #3, CUSTOMER'INFO          ! Get next record
530       GOSUB DISPLAY'INFO              ! Display information in record
535   NEXT REC'NUM
540   GOTO GET'NUM                        ! See if user wants to look again
!
545 ! Time to leave program
550 READ'EXIT:
555    PRINT : PRINT "Closing display file..."
560    CLOSE #3
565    END

800 ! Subroutine to display information in record
805 !
810 DISPLAY'INFO:
815    PRINT : PRINT "CUSTOMER #:"; ID'NUM; "-- "; NAME
820    PRINT SPACE(5); "Street address:"; SPACE(7); STREET
```

```
825     PRINT SPACE(5); "City: "; CITY; SPACE(5); "State: "; STATE
830     PRINT SPACE(5); "Car model: "; MODEL; " Car year: "; YEAR
835     PRINT SPACE(5);
840     IF (INSURANCE = 0) PRINT "No insurance." ELSE PRINT "Car is insured."
845     RETURN

900 ! Subroutine to get information from user and write it to the file
905 !
910 GET'INFO: PRINT
915     ! Make sure we're not trying to add data to a full file
920     IF (IN'USE = TOTAL'RECS) THEN PRINT "File is full..." : GOTO EXIT
925     ! Clear NAME to null so we can test to see if user wants to quit
930     NAME = ""
935     ! Start entering data.  Pad it to proper length with spaces
940     ! so that complete logical record comes out to exactly 64 bytes.
945     INPUT "Customer name: ", NAME : IF NAME = "" GOTO EXIT
950     NAME = NAME + SPACE(20 - LEN(NAME))
955     INPUT "Street address: ", STREET
960     STREET = STREET + SPACE(15 - LEN(STREET))
965     INPUT "City: ", CITY
970     CITY = CITY + SPACE(12 - LEN(CITY))
975     INPUT "State: ", STATE
980     STATE = STATE + SPACE(2 - LEN(STATE))
985     INPUT "Car model: ", MODEL
990     MODEL = MODEL + SPACE(10 - LEN(MODEL))
995     INPUT "Car year: ", YEAR
1000    YEAR = YEAR + SPACE(4 - LEN(YEAR))
1005    INPUT "Car insurance? (Y or N): ", QUERY
1010    QUERY=UCS(QUERY) : IF (QUERY = "Y") INSURANCE = 1 ELSE INSURANCE = 0
1015    ID'NUM = REC'NUM                  ! Customer number is just record #
1020    ! Write whole record; increment records-in-use counter and bump
1025    ! REC'NUM so we are ready to write to next record
1030    WRITE #2, CUSTOMER'INFO : IN'USE = IN'USE + 1 : REC'NUM = REC'NUM + 1
1035    PRINT "Customer ID Number is:"; ID'NUM
1040    GOTO GET'INFO
1045 !
1050 ! We want to stop entering data
1055 EXIT:
1060    PRINT "Now closing output file."
1065    REC'NUM = 0 : WRITE #2, HEADER'RECORD
1070    PRINT "Total number of records in file:";
1075    PRINT TOTAL'RECS,"Records in use:"; IN'USE
1080    CLOSE #2
1085    RETURN
```

CHAPTER 16

CHAINING TO BASIC AND SYSTEM PROGRAMS

The CHAIN statement terminates execution of the current program and initiates the execution of a new program or system function. The new program to be executed must be named in the CHAIN statement itself; that name may be a full file specification. The file named in the statement may be another AlphaBASIC program (compiled only), or it may be a system command or command file. This allows your program to execute a command file and invoke system commands as well as execute other AlphaBASIC commands.

16.1 CHAINING TO ANOTHER ALPHABASIC PROGRAM

CHAIN assumes a default extension of .RUN, which designates a new AlphaBASIC program to be executed. If the extension of the evaluated file specification is indeed .RUN (either explicitly or by default), the specified BASIC program is loaded into memory and executed. (If you do not specify a device and account, BASIC follows the search pattern outlined in Section 3.10, "Library Searching," in looking for .RUN files. If you do specify a device and account, BASIC looks in the specified area.) All variables in the new program are first cleared to zero prior to execution. Also, all variables in the current program are set to zero (or null, if strings). The BASIC program that you specify must be a compiled (.RUN) file. Some examples of legal CHAIN statements are:

```
CHAIN "PAYROL"
CHAIN "PAYROL.RUN"
CHAIN "DSK1:PAYROL[101,13]"
```

Due to the fact that programs are compiled and not interpreted, there is no way to execute a program at any entry point other than its physical beginning. There is also no internal method for passing parameters between programs, but you can accomplish this function for yourself by using the BASIC assembly language subroutine COMMON to store data in a common memory area. COMMON allows you to store information either in system memory (where programs run by all users on the system can access the information) or an individual user's memory partition (where only programs run by that user can

access the information). For details on using COMMON, see COMMON - BASIC Subroutine to Provide Common Variable Storage, (DWM-00100-18), in the "BASIC Programmer's Information" section of the AM-100 documentation packet. In addition to sharing information, you can use the common area to pass parameters to the chained program. For example, the current program can pass a parameter to the new program which it uses in an ON-GOTO statement to begin execution at some point in the new program based on the value passed in the parameter.

Another way to make sure that chained programs can share information is the use of disk files. The current BASIC program can open a data file, write the variables it wants to share into that file, and then close the file. When the new file is chained in, it can open the file and read the necessary information.

16.2 CHAINING TO SYSTEM FUNCTIONS

It is sometimes desirable to transfer execution to a system function or a command file from a BASIC program. If the extension of the file in the CHAIN statement is not .RUN, the file is a system command program or system command file (a .PRG, .DO or .CMD file). In this case, the AlphaBASIC run-time package creates a dummy command file at the top of the current user partition and transfers control to the monitor command processor. The monitor then interprets this dummy command file as a direct command and executes it. Note that the dummy command file created by the run-time package is merely the one-line name specified in the CHAIN statement. It is not the command file itself, which is the target function desired. Some valid examples are:

```
CHAIN "SYSTAT.PRG[1,4]"
CHAIN "TEST1.CMD"
CHAIN "DSK0:BCKUP.CMD[2,2]"
CHAIN "TRANS.DO[110,0]"
```

Note that if the device and account are not specified, the action taken is the same as if you had entered the command directly from your keyboard. That is, if you omit device and account specifications, the monitor command processor searches for command files or programs in the following order:

1. System memory
2. User memory
3. The account and device you are logged into.

(NOTE: To load a file into your user memory partition, use the monitor level LOAD command. To load the file into system memory (where it may be accessed by all users on the system), the System Operator must add the appropriate SYSTEM command line to the system initialization command file.)

Note also that when you chain to a monitor command, after the command has finished executing, it returns you to the monitor level, rather than BASIC. This means that if you wish to automatically return to some AlphaBASIC program, you have to execute a command file whose final command is a RUN command which specifies that original BASIC program.

CHAPTER 17

ERROR TRAPPING

AlphaBASIC allows your program to trap errors that would normally cause the system to print an error message and abort the program run. When you are in interactive mode, an error returns you to AlphaBASIC; if you are in compiler mode, an error returns you to the monitor. Use of the ON ERROR GOTO and RESUME statements causes immediate action to be taken to recover from errors detected within the program.

17.1 ON ERROR GOTO STATEMENT

Error trapping is enabled and disabled by using the ON ERROR GOTO statement in one of two forms. The first form specifies a line number (or label) within the program. When the program encounters this ON ERROR statement, it stores the line number and sets a flag enabling error trapping. If an error occurs any time after this, BASIC transfers control to the routine specified by the line number or label. Examples of this form of the statement are:

```
ON ERROR GOTO 500
ON ERROR GOTO TRAP'ROUTINE
```

The error routine must then take appropriate action based on the type of error.

The second form of the statement disables further user error trapping by specifying a line number of zero or leaving the line number off completely.

```
ON ERROR GOTO 0
ON ERROR GOTO
```

After executing the above form, if an error occurs, the program prints the standard error message and aborts the program run.

A special case exists when the above statement is encountered within an error recovery routine (prior to executing the RESUME statement). In this instance, the user error trapping is disabled and the existing error is forced to be processed by BASIC's error handling as if no error trapping

were ever enabled. It is recommended that all error trapping routines execute the ON ERROR GOTO 0 statement for all errors which have no special recovery processing.

NOTE: If an error occurs within the error trapping routine itself, that error is processed and the error message (?Error in error trapping) occurs. There is no method to detect errors within the error recovery routine.

17.2 ERR(X) FUNCTION

The ERR function returns the following data based on conditions at the time of the error:

ERR(0) = numeric code specifying the type of error detected
ERR(1) = last line number encountered prior to the error
ERR(2) = last file number accessed (only relevant for file errors)

17.2.1 Error Codes Returned by ERR

Code	Meaning
1	Control-C interrupt
2	System error
3	Out of memory
4	Out of data
5	NEXT without FOR
6	RETURN without GOSUB
7	RESUME without ERROR
8	Subscript out of range
9	Floating point overflow
10	Divide by zero
11	Illegal function value
12	XCALL subroutine not found
13	File already open
14	IO to unopened file
15	Record size overflow
16	File specification error
17	File not found
18	Device not ready
19	Device full
20	Device error
21	Device in use
22	Illegal user code
23	Protection violation
24	Write protected
25	File type mismatch
26	Device does not exist
27	Bitmap kaput

28	Disk not mounted
29	File already exists
30	Redimensioned array
31	Illegal record number
32	Invalid filename
33	Stack overflow

For example, if PRINT ERR(0) returns a 10, you know that the program tried to divide a number by zero.

17.3 RESUME STATEMENT

The RESUME statement is used to resume execution of the program after the error recovery procedure has been performed. It also re-enables Control-C detection, which is turned off while BASIC processes the error trapping routine. The statement takes on two forms similar to the forms of the ON ERROR GOTO statement. The first form specifies a line number (or label) within the program where the execution is to be resumed:

```
RESUME 410
RESUME TRY'AGAIN
```

The second form specifies a line number of zero, or no line number at all, and causes the execution to be resumed at the statement which caused the error to occur:

```
RESUME 0
RESUME
```

Both forms cause the error condition to be cleared and error trapping to be enabled again.

NOTE: You must never use the GOTO statement to exit from an error trapping routine. You must use RESUME. This is because RESUME clears the Error stack, but GOTO does not, which causes problems for later error handling.

17.4 CONTROL-C TRAPPING

When you type a Control-C on your keyboard during the execution of an AlphaBASIC program, the program is suspended at the next statement. Action taken then depends upon the status of the error trapping flag. If no error trapping is enabled, the program is aborted and the appropriate message is printed on the terminal. If error trapping is enabled, the error trapping routine is entered with the code in ERR(0) being set to 1. This feature allows you to prevent users from inadvertently exiting programs during critical times such as file updates.

Control-C action is suspended during error recovery processing to prevent accidentally aborting the program during an error routine. The Control-C is detected immediately upon execution of the RESUME statement.

17.5 SAMPLE PROGRAMS

The simple program below contains an error trapping routine that handles "divide by zero" errors. Note that a successful error trapping routine must either resolve the error or exit the program. For example, if the program below had merely printed an error message and then RESUMEd back to the line where the error occurred, the "divide by zero" error would still exist, BASIC would again transfer control to the error trapping routine, and we would be in an eternal loop. Instead, the program resolves the error by changing the values of the problem variables to 1, and then resuming program execution; this time around, a divide-by-zero error cannot occur, and everything is all right.

```
10    ON ERROR GOTO DIVIDE'BY'ZERO
20    INPUT "Enter two numbers: ", A, B
30    PRINT "A/B ="; A/B
40    END
50 DIVIDE'BY'ZERO:
60    ! If error is not "divide by zero" exit the program.
70    IF (ERR(0) <> 10) THEN END
80    PRINT " Division by zero undefined!-- setting A and B to 1"
90    A = 1 : B = 1                ! Reset A and B so that division works.
100   RESUME                       ! Go back to line where problem occurred.
```

Two sample runs of the program look like this:

```
Enter two numbers: 2,3 (RET)
A/B = .666667

Enter two numbers: 3,0 (RET)
A/B = Division by zero undefined!-- setting A and B to 1
A/B = 1
```

The following program shows a small, uncomplicated error trap routine that handles a Control-C. Notice that we enable the error trapping routine CATCH'CTRLC: just before the user enters input. Directly afterward, we disable our routine and re-enable the regular BASIC error trapping via the ON ERROR GOTO 0 statement. This is to catch any errors other than a Control-C that might occur in the rest of the program.

```
10 !               ERROR TRAPPING SAMPLE PROGRAM
20
30 ! Define error code for Control-C, and various string variables.
40
50 MAP1 CONTROL'C,F,,1
60 MAP1 SCRATCH,S,16," "
70 MAP1 ANSWER,S,16," "
80 MAP1 QUERY,S,1
90
100  ! Begin Main Program
110
120  START: PRINT
130     PRINT "This program converts positive decimal numbers to binary."
140
150  ! Ask user for decimal number.
160
170  GET'NUMBER:
180  ! Turn on our error trap to catch Control-C on input.
190     ON ERROR GOTO CATCH'CTRLC
200     INPUT "Enter a number between 1 and 65535: ",NUMBER
210  ! If user typed a Control-C, we've already caught it, so turn off our
220  ! error trapping and turn regular BASIC error trapping back on in
230  ! case other error occurs.
240     ON ERROR GOTO 0
250     IF NUMBER < 0 GOTO GET'NUMBER
260     CURRENT=NUMBER
270
280  ! Now calculate answer.
290
300  CALCULATE:
310     IF (CURRENT/2 = FIX(CURRENT/2)) THEN &
            SCRATCH=SCRATCH+"0" ELSE SCRATCH=SCRATCH+"1"
320     IF FIX((CURRENT/2) = 0) GOTO DISPLAY     ! Done.
330     CURRENT=FIX(CURRENT/2)                   ! Get rid of remainder.
340     GOTO CALCULATE
350
360  ! Display routine.  Reverses string so that answer is in proper order.
370
380  DISPLAY:
390     FOR I = 1 TO LEN(SCRATCH)
400       ANSWER=ANSWER+SCRATCH[-I;1]
410     NEXT I
420     PRINT "     The decimal number";NUMBER;"is";ANSWER;"in binary."
430     SCRATCH= " " : ANSWER = " "              ! Initialize answer to null.
440     GOTO GET'NUMBER
450
```

```
460  ! Error trapping routine.  Just looks for Control-C.  Gives user chance
470  ! to quit or resume.
480
490  CATCH'CTRLC:
500     IF (ERR(0) <> CONTROL'C) THEN RESUME
510     INPUT "Do you wish to quit? (Y or N): ",QUERY
520     QUERY = UCS(QUERY) : IF (QUERY = "N") THEN RESUME GO'AHEAD
530     PRINT : PRINT "So long..." : PRINT
540     END
550  GO'AHEAD:- ! User wants to resume after ^C
560     PRINT : PRINT "Resuming..." : PRINT
570     GOTO GET'NUMBER
```

A sample run of the program above looks like this:

```
.RUN CNVRT (RET)

This program converts positive decimal numbers to binary.
Enter a number between 1 and 65535: 24 (RET)
     The decimal number 24 is 11000 in binary.
Enter a number between 1 and 65535: ^C        [you typed a Control-C]
Do you wish to quit? (Y or N): Y (RET)

So long...
.
```

CHAPTER 18

CALLING EXTERNAL ASSEMBLY LANGUAGE SUBROUTINES

AlphaBASIC supports the use of external assembly language subroutine programs callable from your BASIC programs. There are several good reasons why you might want to use an assembly language program to carry out a function rather than using another BASIC program. Assembly language programs are generally much smaller and faster than equivalent BASIC programs; when speed and size are important factors, you may want to code your programs into assembly language. Yet another reason for using assembly language programs is simply that some tasks are too awkward (or even impossible) to do from within a higher-level language. Assembly language programs are uniquely suitable for applications that require that you work more closely with the hardware or operating system than is convenient or possible in BASIC.

This chapter gives information on writing your own assembly language subroutines for BASIC, and on calling such routines from within a BASIC program.

Although you may want to write your own assembly language subroutines, note that we do provide a set of existing assembly language subroutines in the BASIC Library Account, DSK0:[7,6]. (For information on these subroutines, see the "BASIC Programmer's Information" section of the AM-100 documentation packet.) In addition, a set of business-oriented assembly language subroutines is available from your dealer.

To call an assembly language subroutine from an AlphaBASIC program, use the XCALL statement. The syntax for this statement is as follows:

```
XCALL routine{argument1{,argument2,...argumentN}}
```

The routine to be called is an assembly language program which has been assembled using the MACRO assembler. The resulting .PRG program file must then be renamed to give it the assumed extension .SBR, indicating that it is a subroutine and not a runnable program. When the XCALL statement is executed by the AlphaBASIC run-time system, the named subroutine is located in memory and then called as a subroutine (see Section 18.4, below, for more information on automatic subroutine loading.) AlphaBASIC first saves all registers, then sets certain parameters into those registers for use by the

external subroutine. The addresses of the arguments are calculated and entered into an argument list in memory along with their sizes and type codes. The base address of this list is then passed to the user routine in register R3.

The arguments may be one of two basic forms: 1) A variable name may be used, in which case the argument entry in the list references the selected variable within the user impure area. This variable is available to the called subroutine for both inspection and modification. 2) The argument may also be an expression (numeric or string), in which case the expression is evaluated and the result is placed on the arithmetic stack (referenced by R5). This result, instead of a single variable, is then referenced in the argument list entry. It is only available for inspection, since the stack is cleared when the subroutine exits.

The user routine is free to use and modify all six general work registers (R0-R5), and may use the stack for work space as required. When the subroutine has completed its execution, a return must be made to the run-time system by executing the RTN subroutine return instruction.

18.1 REGISTER PARAMETERS

The following registers are set up by the run-time system to be used as required by the external subroutine. They may be modified, if desired, since they have been saved before the subroutine was called:

R0 Indexes the user impure variable area. R0 is used throughout the run-time system to reference all user variables. Details on the format of this area are not available at this time. R0 may be used as a work register.

R3 Points to the base of the argument list. R3 may be used to scan the argument list for retrieval of the argument parameters.

R4 Points to the base of the free memory area that may be used by the external subroutine as work space. This is actually the address of the first word following the argument list in memory, and, if desired, may be used to store a terminator word to stop the scanning of the argument list.

R5 This is the arithmetic stack index used by the run-time system. The arithmetic stack is built at the top of the user partition and grows downward as items are added to it. When the external subroutine is called, R5 points to the current stack base. Since the arithmetic stack may contain valid data, the external subroutine must not use the word indexed by R5 or any words above it.

18.2 ARGUMENT LIST FORMAT

The list of arguments specified in the XCALL statement may range from no arguments at all to a number limited only by the space on the command line. To pass these arguments to the external subroutine, an argument list is built in memory which describes each variable named in the list and tells where it can be located in the user impure area. The variables themselves are not actually passed to the subroutine, but rather their absolute locations in memory are. In this way, the subroutine may inspect them and modify them directly in their respective locations. This does not apply to expressions which are built on the stack as described previously.

R3 points to the first word of the argument list, which is a binary count of how many arguments were contained in the XCALL statement. Following this count word comes one 3-word descriptor block for each argument specified. If there are no arguments in the XCALL statement, the argument list consists only of the single count word containing the value of zero.

The format of each 3-word block describing one argument is as follows:

Word 1 Variable type code. Bits 0-3 contain the type code for the specific variable: 0=unformated, 2=string, 4=floating point, 6=binary, 7 through 17 are currently unassigned. Bit 4 is set to indicate the variable is subscripted or cleared to indicate the variable is not subscripted. Other bits in the type code word are meaningless.

Word 2 Absolute address of variable in user impure area. This address is the first byte of the variable no matter what its type or size might be.

Word 3 Size of the variable in bytes.

Note that the above descriptions also apply to the expression arguments, except that the results are located above the address specified by R5 instead of below it.

The argument list is built in free memory directly above the currently allocated user impure area. R4 points to the word immediately following the last word in the argument list. You may scan the argument list and determine its end either by decrementing the count word at the base of the list or by scanning until the scan index reaches the address in R4.

18.3 FREE MEMORY USAGE

When the subroutine is called, indexes R4 and R5 mark the beginning and end of the free memory that is currently available for use as workspace. This area is not preserved by the run-time system, and the subroutine must not count on its security between XCALL statements.

Note that the word at @R4 may be used as the first word, but the word at @R5 is the base of the arithmetic stack and must not be destroyed. The last word of actually free memory is at -2(R5).

The run-time system has its own internal memory management system and does not conform to the AMOS operating system memory management method. Therefore, the external subroutine must not use the GETMEM monitor calls to generate a block of work space in memory. Also, if any file calls are to be done they must be done with internal buffers, since the INIT call sets up a buffer by using the GETMEM monitor call.

18.4 AUTOMATIC SUBROUTINE LOADING

When a BASIC program calls a subroutine via an XCALL statement, BASIC attempts to locate the subroutine in user or system memory. If it is unable to do so, it attempts to load the subroutine from the disk, following the search pattern outlined in Section 3.10, "Library Searching." If a BASIC program fetches a subroutine from disk, BASIC loads it into memory only for the duration of its execution. Once the subroutine has completed its execution, it is removed from memory if it was loaded via this automatic procedure. Therefore, if a subroutine is to be called a large number of times, it is wise to load it into memory (using the monitor LOAD command) to avoid the overhead of fetching the subroutine from disk. NOTE: Subroutines loaded into memory via the monitor LOAD command remain in memory until you reset the system or until you use the monitor command DEL to delete them.)

CHAPTER 19

USING ISAM FROM WITHIN BASIC

This chapter discusses the ISAM information management system and its use from within BASIC. It is important when reading the following sections that you be familiar with opening and using random data files. If you are not, refer first to Chapter 15, "AlphaBASIC File I/O System."

The ISAM program is a tool for organizing and retrieving data. The name stands for "Indexed Sequential Access Method," and refers to the manner in which the data is organized.

AlphaBASIC has the ability to process indexed sequential files by linking to the ISAM assembly language package (which must reside either in system memory or in individual user memory). ISAM supports multiple index files via some elementary ISAM statements that allow the direct control of index file and data file items. This chapter assumes that you are familiar with the Alpha Micro ISAM system. For more detailed information on ISAM files and the ISAM assembly language package, please refer to the ISAM System User's Guide (DWM-00100-06), and Important Notice for ISAM Users, (DWM-00100-36), in the AM-100 documentation packet.

19.1 FILE STRUCTURE

An indexed sequential file consists of one data file and one or more index files which link to the data file. The data file is structured in the same way as a normal random access file except that ISAM links all records which are not currently active to each other in a chain called the "free data record list." All data records reside in the data file and the data records may be any size up to the maximum of 512 bytes. As in the normal random file, data records are not split across physical 512-byte block boundaries in the file. Index files are arranged in a complex balanced tree structure and contain one symbolic key for each active data record plus a link to that data record in the data file. This link is the relative record number and is used in the same manner as its counterpart in a normal random access file. The index file also contains an array of internal links which comprise the sequential access tree structure.

Two references used in this manual may be confusing if they are not understood. When we talk about an "indexed file," we are speaking of the entire file structure in general, including the data file and one or more index files. We talk about an "index file" when specifically speaking of the portion of the structure which contains only the symbolic keys and the tree links. ISAM stores symbolic keys in an index file in ASCII collating sequence. Index files may be primary or secondary.

IMPORTANT NOTE: Both INDEXED and INDEXED'EXCLUSIVE modes require that ISAM be able to write to the disk containing the index files even if you do not plan to do anything more than read from the disk; therefore, make sure that that disk containing the index files is not write-protected.

All indexed sequential files must be created by the ISMBLD program prior to access by any AlphaBASIC program. There is no method for the creation of a new indexed file within the AlphaBASIC language since this would require a prohibitive amount of seldom-used code. You may, however, create indexed files by using the feature that allows a BASIC program to create and then execute a command file. This command file could set up parameters and then call the ISMBLD program to perform the actual creation of the files.

For compatibility with existing structures, the data file must have an extension of .IDA and all index files must have an extension of .IDX. There must be at least one index file which is called the primary index file. There may also be additional index files called secondary index files which also link to the primary data file. The primary index file must always be opened in any program in order to gain access to the data file. This is true even if you only intend to access the data file through one of the secondary index files in the current program. For information on file structures and operating the ISMBLD program, refer to the ISAM System User's Guide (DWM-00100-06).

19.2 SYMBOLIC AND RELATIVE KEYS

Indexed files are accessed by one of two specific types of keys. The relative key is already familiar to us since it is the same type of key used to access normal random files. The relative key is the floating point record-number variable specified in the OPEN statement for the indexed file. It contains the number of the logical record to be accessed. A relative key when used with an indexed file is used only to access a specific record in a data file.

The symbolic key is new to us and is used only with indexed files. Symbolic keys are ASCII strings of variable lengths and are used to access the index file (primary or secondary). Symbolic keys are specified in the ISAM statements when accessing the index file, and are used to retrieve the relative key of the associated data record in the data file. The concept of symbolic versus relative keys and their different uses is an important one, and misuse of them causes the ISAM system to malfunction in a number of ways. Symbolic keys are used with the ISAM statement; relative keys are used with the OPEN statement so that READ and WRITE statements can be

successfully performed. In most instances, the use of the relative key is transparent to you, and is merely a device automatically set up and referenced by the above calls.

19.3 THE ISAM STATEMENT

You access Indexed files by a special statement in AlphaBASIC called the ISAM statement. This statement has the general form:

```
ISAM #file-channel,code,symbolic-key
```

All ISAM statements follow the above format using a different numeric value in "code" to specify the specific function to be performed by the ISAM package. All ISAM statements directly translate into a specific type of call to the assembly language ISAM program. A symbolic key must always be specified even for those functions which do not require the use of one. (This simplifies syntax checking and execution coding.) You may use a dummy string variable if you desire. Briefly, the following codes are used by the ISAM statement:

1 - Find a record in the data file by symbolic key (i.e., return the relative record number in the variable specified by the OPEN statement that opens the data file/primary index file).

2 - Find the next data record (by the order in which the symbolic keys appear in the index file). Return the relative record number in the variable specified by the OPEN statement.

3 - Add a symbolic key to an index file.

4 - Delete a symbolic key from an index file.

5 - Locate the next free data record in the data file (returning the relative record number in the variable specified in the OPEN statement).

6 - Delete a record from a data file, and return that record to the free list.

An error results if an ISAM statement is executed with the value of "code" not equal to one of the above numbers. The "code" may be any legal numeric expression which is resolved at run-time.

19.3.1 The ISAM Statement Codes

Below is a fuller explanation of the ISAM codes. Some of these codes require a relative key as input; others return a relative key to be used when accessing the data record. This relative key is returned in the variable specified by the OPEN statement for the index file being accessed

by the ISAM statement. This then leaves the system properly set up for an immediate access to the corresponding data record via a READ or WRITE statement.

Code 1 - ISAM searches in the specified index file for the key which matches the symbolic key in the ISAM statement. If a match is found, ISAM returns the associated relative key so that your program can access the data file. If the key is not found, ISAM returns an error code 33 (see Section 19.8, "Error Processing").

Code 2 - ISAM accesses the specified index file and locates the next symbolic key. ISAM then returns the corresponding relative key in preparation for a READ or WRITE to the data file. If this is the first access to the file following the OPEN statement, ISAM locates the first symbolic key. If this statement follows a previous code 1 statement, ISAM locates the next symbolic key following the code 1 key. If there are no more keys in the index file, ISAM returns an end-of-index-file error (38), and your program should not access the data file further until ISAM returns a valid relative key.

Code 3 - ISAM adds the specified symbolic key to the index file along with the relative key. The relative key must be in the corresponding variable specified in the OPEN statement. ISAM normally sets up this relative key by a prior code 5 ISAM statement which delivers the next free data record to be used. This relative key then becomes the result of any index search which locates this specific associated symbolic key.

Code 4 - ISAM locates the specified symbolic key in the index file and deletes it. ISAM then returns the corresponding data record relative key so that the data record may be deleted and returned to the free list by using a code 6 ISAM statement. If ISAM cannot locate the symbolic key in the index file, it gives you a "record not found" error.

Code 5 - ISAM extracts the next available data record from the free list and returns the relative key in preparation for a code 3 index key addition statement. If no more data records are free in the data file, ISAM returns a "data file full" error. All free records in the data file are kept in a linked list called the "free list." This list is built initially by ISMBLD and contains all the records in the data file. As code 6 ISAM statements are executed, ISAM again returns the records to the free list for reuse. ISAM does not modify the index file and ignores the symbolic key in the statement. This call must be made only to the primary index file number.

Code 6 - the data record specified by the relative key is returned to the free list for reuse by a code 5 call. The index file is not modified and the symbolic key in the statement is ignored. This call must be made only to the primary index file number.

19.4 OPENING AN INDEXED FILE

As with other types of files, an indexed file must be opened with a specific file-channel number prior to any references to the file by other statements. The OPEN statement follows the same format as that used by the normal random files except that you specify INDEXED or INDEXED'EXCLUSIVE mode.

```
OPEN #file-channel,filespec,INDEXED,record-size,relative-key
OPEN #file-channel,filespec,INDEXED'EXCLUSIVE,record-size,relative-key
```

#file-channel	Any numeric expression that evaluates to an integer from 0-65535 (0 is defined as the user terminal).
filespec	Any string expression that evaluates to a legal AMOS file specification (optionally including account and device specifications). Specifies the data file/primary index file or the secondary index file. (The primary index file always has the same name as the data file, but has the .IDX extension; the data file has the .IDA extension.)
INDEXED	Specifies indexed sequential mode.
INDEXED'EXCLUSIVE	Specifies indexed exclusive mode. (See Section 19.7 for more information.)
record-size	Expression that specifies the logical record size for the data file.
relative-key-variable	Floating point variable that contains the record number of the logical record you want to access.

The filespec must refer to the name given to the index file during the ISMBLD creation. If this is a call to open a secondary index file, you must have already previously opened the corresponding primary index file on another file number so that the data file may be accessed.

As an example, assume that an indexed file structure consists of the primary index and data files named MASTER.IDX and MASTER.IDA respectively. The structure also has secondary index files named ADRESS.IDX and PAYROL.IDX which access the MASTER.IDA file in different sequences. If you desire to process the file structure via the sequence used by the ADRESS.IDX index file, the following two statements are required:

```
OPEN #1, "MASTER", INDEXED, RECSIZ, RELKEY
OPEN #2, "ADRESS", INDEXED, RECSIZ, RELKEY
```

The first statement opens both the data file and the primary index file. NOTE: Remember that there are now three files opened: 1) the data file,

MASTER.IDA; 2) the primary index file, MASTER.IDX; and 3) the secondary index file, ADRESS.IDX.

Note that the record size expression (RECSIZ) and the relative key variable (RELKEY) are identical in both statements. This is important since they both refer to the same data file (MASTER.IDA). ISAM statements may then be made referring to either index file (#1 or #2) but all READ and WRITE statements must be made to the data file (#1) which is associated with the primary index file. In other words, READ and WRITE statements must not be made to file #2.

19.5 READ AND WRITE STATEMENTS

The ISAM calls do not access the data records themselves but merely deliver back the relative key of the associated data record to be used. Normal READ and WRITE statements are then used to actually retrieve or write into the data record itself. These READ and WRITE statements follow the same format used when accessing a normal random access data file in AlphaBASIC. The relative key associated with the primary file (as specified in the OPEN statement) must contain a valid relative key for the operation or an error results. READ and WRITE statements as mentioned before must only be made using the primary index file-channel number. For example:

```
10 OPEN #3,"PAYROL",INDEXED,67,NUM'REC
20 ISAM #3,1,NAME        ! Get record
30 READ #3,LABEL         ! Read record
```

19.6 CLOSING AN INDEXED FILE

In order to ensure that all data records have been rewritten to the data file and that all links in the index file have been properly updated and rewritten to the disk, it is imperative that all index files (primary and secondary) be closed using the normal CLOSE statement, referencing the correct file-channel number. Failure to do so may result in destroying the link structure. NOTE: It makes no difference in which order you close the ISAM files; however, remember that you cannot access a secondary index file if you have already closed the primary index.

19.7 INDEXED'EXCLUSIVE MODE

When your program is the only program that needs to access an ISAM indexed file, you can specify INDEXED'EXCLUSIVE as the mode in which you open the file. For example:

```
OPEN #5,"PAYROL",INDEXED'EXCLUSIVE,100,REC'NUM
```

The statement above opens the data file PAYROL.IDA and the primary index file PAYROL.IDX in exclusive mode. The main advantage of INDEXED'EXCLUSIVE mode is a large increase in the speed with which your programs can access the indexed file. It also prevents other users from accessing your indexed file until you close the file. Otherwise, it works in the same way as INDEXED mode.

In INDEXED'EXCLUSIVE mode, ISAM knows that no other program is going to access the indexed file while your program is working with it. Therefore, ISAM can take full advantage of prior knowledge about the file for every access and can speed up your access time considerably.

When your program opens an indexed file in the more common INDEXED mode, you must use file-locking procedures to protect your indexed file if other programs are going to access it while you are working with the file. (For information on the file-locking subroutines XLOCK and FLOCK, see the "BASIC Programmer's Information" section of the AM-100 documentation packet.)

When your program opens an indexed file in INDEXED'EXCLUSIVE mode, ISAM will not allow another user to access the specified indexed file; if they try to do so, they see a "file not found" error message. This means that you only have to worry about file-locking at the moment in which you are opening the indexed file. You may prevent another program from accessing your indexed file at the moment that you are opening it by securing the file via the file-locking routines XLOCK or FLOCK, or just by making sure that no other user is running a program that accesses the file.

Remember: The advantage of an indexed file opened in INDEXED'EXCLUSIVE mode is that no other user can access the file while you are using it. If you need to have several programs access the file, use the INDEXED mode; in that case, remember to use file-locking procedures to prevent users from trying to access the file at the same time.

One feature of the INDEXED'EXCLUSIVE mode is that it temporarily renames the .IDX file to an .IDY extension to prevent ISAM from letting other programs access the file. If something should go wrong (such as a system crash), ISAM may not be able to rename the file to its original .IDX extension, and you will have to do so yourself.

For more information on INDEXED'EXCLUSIVE mode, see Important Notice for ISAM Users, in the "User's Information" section of the AM-100 documentation packet.

19.8 ERROR PROCESSING

Every ISAM statement may potentially return some kind of an error. These errors fall into two categories: hard or soft errors. Hard errors are those errors returned to ISAM by the monitor; such errors indicate invalid disk operations (e.g., file not found). Soft errors occur within the ISAM processor and indicate an error or condition peculiar only to ISAM files.

Hard errors cause AlphaBASIC to print an error message and abort to the monitor or (if error trapping is enabled) pass control to your own error trapping routine. (See Chapter 17, "Error Trapping," especially Section 17.1, "ON ERROR GOTO Statement," for information on writing your own error trapping routines.)

19.8.1 Soft Errors

Soft errors never result in an error message or error trap, and BASIC does not stop program execution when a soft error occurs. It is therefore up to your program to test for such errors. You must test for a soft error after every ISAM statement. Otherwise you have no way of knowing whether or not the statement was successfully executed. Use the ERF file error function. ERF is used in much the same way as the EOF function. You must specify the file number used in the ISAM statement whose success you want to test. If the ERF function returns a zero, the preceding ISAM statement was successfully executed; if ERF returns a nonzero number, some error was detected, and your program must take corrective action before accessing the file again. For example:

```
ISAM #2, 2, PART'NO
IF ERF(2) <> 0 THEN GOTO ISAM'ERROR
```

The routine ISAM'ERROR might print an error message and then exit. (See Section 11.3.2, "ERF(X)" for more information on ERF.)

The soft error codes returned by ERF are:

```
32 - Illegal ISAM statement code
33 - Record not found in index file search
34 - Duplicate key found in index file during attempted
     key addition
35 - Link structure is smashed and must be re-created
36 - Index file is full
37 - Data file is full (i.e., free list is empty)
38 - End of file during sequential key read
```

19.9 USING INDEXED SEQUENTIAL FILES

The sections below give step-by-step instructions for using indexed files. For a complete demonstration of using ISAM from within BASIC, refer to the sample BASIC program in Section 19.10 at the end of this chapter.

Remember that you must load the ISAM program into memory before using a BASIC program that uses ISAM statements. Use the monitor LOAD command:

```
.LOAD SYS:ISAM.PRG
```

(NOTE: The "SYS:" device specification is an ersatz device specification that specifies the System Library account, DSK0:[1,4]. The command above is the same as: LOAD DSK0:ISAM.PRG[1,4].)

19.9.1 Creating an Indexed File

Use the ISMBLD program to create a data/index file combination. If you want a secondary index file, use ISMBLD again to create that file. While using ISMBLD, you may either load the empty data/index file with information from an ordinary sequential file, or you may leave the file empty and let your BASIC program enter the data. For information on using ISMBLD, see the ISAM System User's Guide.

19.9.2 Adding Data to an Indexed File

From within your BASIC program:

1. Open the data/index file with an OPEN statement. For example:

```
OPEN #1, "PHONES", INDEXED, RECSIZE, RELKEY
```

   Remember to open any secondary index files that you might want to use via separate OPEN statements on different file-channel numbers:

```
OPEN #3, "IDNUM", INDEXED, RECSIZE, RELKEY
```

2. Use a code 1 ISAM statement to see if the index entry you want to add already exists. For example:

```
ISAM #1, 1, NAME
```

   Check to see if an error was returned:

```
IF ERF(1) = 0 THEN PRINT "Duplicate name" : GOTO GET'NAME
```

   (If no error occurred, the index entry already exists, and you can't add it.)

   If you are using secondary index files, also check to see that the secondary index entries don't already exist.

3. Now, use a code 5 ISAM statement to get the next free data record. For example:

```
ISAM #1,5,DUMMY
```

   Check to make sure that an error (e.g., 37 - "data file is full (free list is empty)") did not occur. For example:

```
IF ERF(1) <> 0 THEN GOTO ISAM'ERROR
```

4. If no error occurred, the record number of the next free record is in the relative key variable defined by the OPEN statement. Now you can write your data to the data file:

```
WRITE #1, INFO
```

5. Now you must add the symbolic keys for that data record to the index files, using a code 3 ISAM statement. (Those symbolic keys will then link to that data record.) Be sure to check for an ISAM error after each addition.

6. After adding all your information, close the ISAM files.

19.9.3 Reading Data Records in Symbolic Key Order

ISAM stores symbolic keys in the index file in ASCII collating sequence. To retrieve records in the order in which their keys appear in an index file:

1. Open the indexed sequential file with an OPEN statement. If you also want to open one or more secondary index files that cross-index to the primary index file, use one OPEN statement for each secondary index file.

2. Use a code 2 ISAM statement to find the next symbolic key.

3. Check to make sure that the ISAM statement didn't return an error. For example:

```
IF ERF(1) = 38 THEN PRINT "End of the file" : GOTO PROMPT
IF ERF(1) <> 0 THEN GOTO ISAM'ERROR
```

4. The proper relative key is in the relative key variable specified by the OPEN statement; now use a READ statement to read in the data record associated with that key. (Remember that the READ statement is done to the primary data/index file, even though you may have specified a symbolic key contained in a secondary index file.)

5. Repeat these procedures to step through the data records in the order of the symbolic keys in the index files. Close all files when you are done.

19.9.4 Reading Data Records Randomly by Symbolic Key

1. Open the data/index file with an OPEN statement. You must also open any secondary index file you want to use.

2. Use a code 1 ISAM statement to locate the data record you want to find. The statement must contain the symbolic key associated with the data record you want, and must contain the file-channel number associated with the index file that contains the symbolic key.

3. Check for a "record not found" error; this indicates that the symbolic key was not found in the specified index file.

4. If the record was found, the proper relative key is now in the relative key variable defined in the OPEN statement. Use a READ statement to read in the data. (The READ statement includes the file-channel number associated with the data file/primary index file even if the symbolic key used belonged to a secondary index file.)

5. Repeat steps 2 through 4 for each record you want.

6. Close all files.

19.9.5 Updating Data Records

You may sometimes want to change the data in a record in the data file. You may do so by first finding the record you want and then rewriting it:

1. Open the data/index file with an OPEN statement.

2. Locate the record you want via one of the methods above (i.e., by using a code 1 or code 2 ISAM statement).

3. Check to make sure that the record was found. (Use the ERF function.)

4. Now the correct relative key is in the relative key variable defined by the OPEN statement, so use the WRITE statement to rewrite the data record. (Remember to specify the file-channel number associated with the data/primary index file.)

5. Repeat steps 2 through 4 for all records you want to rewrite.

6. Close the files.

The steps above do not change the index files, so do not change the symbolic key in the record you rewrite.

If you need to change the symbolic key(s) in the data record, you must first delete the key in the correct index file (code 4), and then add the new key to the index file (code 3). You do not need to delete and re-create the data record during this operation unless you are entering completely new data.

19.9.6 Deleting a Data Record

Deleting a data record from an indexed sequential file entails not only deleting the record itself from the data file but also deleting all symbolic keys associated with that data record from all index files.

1. Open the data/primary index file and all secondary index files needed.

2. Locate the data record via one of the symbolic keys (a code 1 ISAM statement).

3. Check to see that the statement was executed successfully (i.e., that ERF returned a zero). For example:

```
IF ERF(2) = 33 THEN PRINT "Record not found" : GOTO PROMPT
IF ERF(2) <> 0 THEN GOTO ISAM'ERROR
```

4. Read the data record with a READ statement. (The file number must be the number associated with the data/primary index file.) Extract each symbolic key from that record.

5. Now you must delete all symbolic keys that are associated with the deleted record in each index file. Use code 4 ISAM statements to do so, specifying the symbolic keys you extracted from the data record in the step above.

6. After you delete each symbolic key, check for errors.

7. Now go ahead and delete the data record by using a code 6 ISAM statement.

8. Check to see that no error occurred.

9. Close all files.

NOTE: A good check on the file structure would be to store the relative key in another variable and then compare the relative keys returned by each ISAM code 4 statement to ensure that the symbolic keys all did indeed link to the correct data record. You should also check after each ISAM statement to see if any error occurred.

19.10 SAMPLE ISAM PROGRAM

The sample program below will make clearer the use of the commands discussed above. For more information on using ISAM from within a BASIC program, consult the ISAM System User's Guide, (DWM-00100-06).

We first create or enter our program using the text editor VUE. We'll call it SAMPLE.BAS. After the program has been entered, we compile it:

```
.COMPIL SAMPLE (RET)
```

After we compile the program, and before we run it, we first use the program ISMBLD to build the ISAM files LABELS.IDA (the data file), LABELS.IDX (the primary index file), and HASH.IDX (the secondary index file). Note that we build an empty file (i.e., we type a RETURN after the "Load from file:" prompt). We use the BASIC program below to place data into the file.

```
.ISMBLD LABELS (RET)

Size of key: 25 (RET)
Position of key: 1 (RET)
Size of data record: 67 (RET)
Number of records to allocate: 50 (RET)
Entries per index block: 10 (RET)
Empty index blocks to allocate: 20 (RET)
Primary Directory: Y (RET)
Data file device: (RET)

Load from file:
.ISMBLD HASH (RET)

Size of key: 10 (RET)
Position of key: 58 (RET)
Size of data record: 67 (RET)
Number of records to allocate: 50 (RET)
Entries per index block: 10 (RET)
Empty index blocks to allocate: 20 (RET)
Primary Directory? N (RET)

Secondary index to file: LABELS (RET)
End of primary file
No records loaded
```

Now, before we run our BASIC program, we must load ISAM into memory:

```
.LOAD DSK0:ISAM.PRG[1,4] (RET)
```

Then we run our BASIC program:

```
.RUN SAMPLE (RET)
```

SAMPLE BASIC ISAM PROGRAM

```
10 ! ISAM Sample Program.
20 !
30 ! This program is a simple example of how to handle ISAM files, both
40 ! primary and secondary.  It simulates a very simple-minded mailing
50 ! list program, with the addresses keyed by both name and user
60 ! defined hash code.
70 !
80 ! Define the Mailing List file record.
90 !
100 MAP1 LABEL
110     MAP2 NAME,S,25
120     MAP2 ADDRESS,S,25
130     MAP2 STATE,S,2
140     MAP2 ZIP,S,5
150     MAP2 HASH,S,10
160
170 ! Define record sizes.
180
190 MAP1 RECSIZE,F,6,67                 ! Size of data record.
200
210 ! Open the primary and secondary files.
220     OPEN #1, "LABELS", INDEXED, RECSIZE, RELKEY1
230     OPEN #2, "HASH", INDEXED, RECSIZE, RELKEY1
240
250 PROMPT:
260     PRINT
270     INPUT "ENTER FUNCTION &
            (1=ADD,2=DELETE,3=INQUIRE,4=DISPLAY,99=END): "; FUNCTION
280     ON FUNCTION GOTO ADD'RECORD,DELETE'RECORD,INQUIRE'RECORD,PRINT'LABELS
290     IF FUNCTION=99 THEN GOTO END'IT
300     GOTO PROMPT
310
320 ADD'RECORD:
330     INPUT "ENTER NAME: "; NAME
340     INPUT "ENTER ADDRESS: "; ADDRESS
350     INPUT "ENTER STATE: "; STATE
360     INPUT "ENTER ZIP: "; ZIP
370     INPUT "ENTER HASH: "; HASH
380 ! Add Trailing blanks to the keys.
390     NAME = NAME + SPACE(25-LEN(NAME))
400     HASH = HASH + SPACE(10-LEN(HASH))
410 ! Look up name to verify that it is not a duplicate. (If ERF(1)=0, then
415 ! ISAM found the key in the data file.)
420     ISAM #1, 1, NAME
430     IF ERF(1) = 0 THEN PRINT "DUPLICATE NAME" : GOTO ADD'RECORD
440 ! Verify that hash is not a duplicate.
450     ISAM #2, 1, HASH
460     IF ERF(2) = 0 THEN PRINT "DUPLICATE HASH" : GOTO ADD'RECORD
470 ! Get free data record from primary file and write record out.
480     ISAM #1, 5, NAME
490     IF ERF(1) <> 0 THEN GOTO ISAM'ERROR
```

```
500     WRITE #1, LABEL
510 ! Add key to primary index file.
520     ISAM #1, 3, NAME
530     IF ERF(1) <> 0 THEN GOTO ISAM'ERROR
540 ! Add key to secondary index file.
550     ISAM #2, 3, HASH
560     IF ERF(2) <> 0 THEN GOTO ISAM'ERROR
570     GOTO PROMPT
580
590 DELETE'RECORD:
600     INPUT "ENTER NAME: "; NAME
610     NAME = NAME + SPACE(25-LEN(NAME))
620 ! Verify that the key exists.
630     ISAM #1, 1, NAME
640     IF ERF(1) = 33 THEN PRINT "RECORD NOT FOUND" : GOTO PROMPT
650     IF ERF(1) <> 0 THEN GOTO ISAM'ERROR
660     READ #1, LABEL
670 ! Delete the key from the primary index.
680     ISAM #1, 4, NAME
690     IF ERF(1) <> 0 THEN GOTO ISAM'ERROR
700 ! Delete the key from the secondary index.
710     ISAM #2, 4, HASH
720     IF ERF(2) <> 0 THEN GOTO ISAM'ERROR

730 ! Delete the data record in data file.
740     ISAM #1, 6, NAME
750     IF ERF(1) <> 0 THEN GOTO ISAM'ERROR
760     GOTO PROMPT
770
780 INQUIRE'RECORD:
790     INPUT "BY NAME (1) OR HASH (2): "; FUNCTION
800     IF FUNCTION = 2 THEN GOTO BY'HASH
810     INPUT "NAME: "; NAME
820     NAME = NAME + SPACE(25-LEN(NAME))
830 ! Locate the record.
840     ISAM #1, 1, NAME
850     IF ERF(1) = 33 THEN PRINT "RECORD NOT FOUND" : GOTO PROMPT
860     IF ERF(1) <> 0 THEN GOTO ISAM'ERROR
870 ! Read the record.
880 READ'RECORD:
890     READ #1, LABEL
900     PRINT NAME, HASH
910     PRINT ADDRESS, STATE, ZIP
920     GOTO PROMPT
930 ! Locate record by hash code.
940 BY'HASH:
950     INPUT "HASH: "; HASH
960     HASH = HASH + SPACE(10-LEN(HASH))
970     ISAM #2, 1, HASH
980     IF ERF(2) = 33 THEN PRINT "RECORD NOT FOUND" : GOTO PROMPT
990     IF ERF(2) <> 0 THEN GOTO ISAM'ERROR
1000    GOTO READ'RECORD
1010
```

```
1020 PRINT'LABELS:
1030 ! Read null key to get to front of file.
1040    NAME = SPACE(25)
1050    ISAM #1, 1, NAME
1060 ! Loop thru file doing sequential reads until we hit the end.
1070 LOOP:
1080    ISAM #1, 2, NAME
1090    IF ERF(1) = 38 THEN GOTO PROMPT         ! We hit end-of-file.
1100    IF ERF(1) <> 0 THEN GOTO ISAM'ERROR
1110    READ #1, LABEL
1120    PRINT
1130    PRINT NAME, HASH
1140    PRINT ADDRESS, STATE, ZIP
1150    GOTO LOOP
1160
1170 END'IT:
1180 ! Be sure and close files before we exit.
1190    CLOSE #1
1200    CLOSE #2
1210    END
1220
1230 ISAM'ERROR:                          ! ERF(X) returned an ISAM error
1240    PRINT "?FATAL ISAM ERROR"         ! other than RECORD NOT FOUND.
1250    END
```

APPENDIX A

SUMMARY OF COMMANDS, STATEMENTS AND FUNCTIONS

The following four sections summarize the syntax of the AMOS monitor commands that invoke and control BASIC, and the AlphaBASIC commands, statements and functions.

Commands are instructions to BASIC that affect the way it handles a program. For example, the SAVE command tells BASIC to save a copy of a program on the disk. Commands are not part of the program itself, and may only be used in interactive mode.

Statements are instructions to BASIC from within the program; you might think of them as program "verbs" which tell BASIC how to operate on the program data. For example, the PRINT statement tells BASIC to display the specified data. Although most often part of a program, you can also use some statements directly in interactive mode, outside of a program.

Functions are elements of an expression which compute and return a value. For example, ABS(X) computes and returns the absolute value of X. You may also use functions (in combination with program statements) directly in interactive mode, outside of a program.

The syntax of the commands, statements and functions is illustrated in this appendix using certain conventions. The curly brackets { and } are used to enclose options available for certain commands and statements. These may be nested several deep. Certain commands and statements permit a series of optional elements. The elements are numbered 1 through N, and the variable number of elements in this available series is pictured using three dots (...). For example:

```
INPUT {"prompt-string",}variable1{,variable2...variableN}
```

indicates that your INPUT statement may request an input of a minimum of one numeric or string variable. You may also cause it to request two numeric or string variables, but if you do, the two variables must be separated by commas. And so forth to variableN, where N is some arbitrary number. You may also optionally supply a string literal prompt string.

For the AMOS monitor commands, the underlined dot represents the AMOS prompt you see at the AMOS command level. The (RET) indicates that you should type a RETURN at the point where you see the symbol, following the text.

When we use the term "filespec," we are talking about an AMOS file specification which contains the name of the file and optionally includes a device, account, and extension specification. For instance:

{Devn:}filename{.ext}{[Project,programer-number]}

A.1 AMOS MONITOR COMMANDS

These commands are used only from the AMOS command level. They are illustrated much as you would see them on your terminal.

A.1.1 BASIC

.BASIC (RET)

READY

Places you in the interactive mode of AlphaBASIC and gives you the prompt word READY. From here you may enter certain statements or statement/function combinations without line numbers. BASIC responds to valid entries with immediate results. Invalid entries cause an error message to be returned. You may also enter any valid statements, functions, constant values, variables, arithmetic operators, strings, data or expressions (meaning any combination of the above) as long as they are preceded by a line number from 1 to 65534. These lines combine to form a BASIC program. Line entries invalid due to syntactical errors or illegal formats are reported immediately via error messages. Other illegal entries which cannot be detected immediately are reported during program compilation or program run.

You exit from BASIC back to the AMOS command level via the BYE command.

A.1.2 COMPIL

.COMPIL filespec (RET)

The file specification may simply be the filename of a BASIC program in your account, or it may be a complete file specification including device name, filename and extension and account number. The default extension is .BAS. If the file you specify is not found, the system error message

?Cannot OPEN Filespec - file not found

is returned to you. When the file is found, the system begins to process the file. At the end of the compilation process, a new file has been created in your account called by the filename and with the extension .RUN. This is the compiled program.

A.1.3 Control-C

```
[Type a CONTROL and a C simultaneously]
^C
Operator interrupt in line nnnn of FILE.RUN
```

A Control-C interrupts the execution of the program currently running. Returns you to AMOS command level.

A.1.4 RUN

```
.RUN filespec (RET)

   (The program commences.)
```

At this command, the monitor loads the AlphaBASIC run-time package, RUN, into memory and executes it. RUN in turn loads the fully compiled program which is specified, having the extension .RUN, into memory and executes it. Your program begins to run from the beginning. Interruptions to the program may occur if there is an error in programming, if there is a STOP statement in the program, if you type a Control-C during execution, or if the program finishes.

A.2 ALPHABASIC COMMANDS

The commands are used in the interactive mode of BASIC to control BASIC itself.

A.2.1 BREAK

```
BREAK {{-}line#1{,{-}line#2,...{-}line#N}}
```

Lists all breakpoints set if no line number is specified. Sets a breakpoint at the specified line number if the specified number is positive, or clears a breakpoint at the specified line number if it is negative.

A.2.2 BYE

BYE

Returns you to AMOS command level.

A.2.3 COMPILE

COMPILE

Compiles the program currently in memory.

A.2.4 CONT

CONT

Program execution resumes from the last point of cessation.

A.2.5 CONTROL-C

[Press CONTROL KEY and C KEY simultaneously]

(Terminal rings and you see the message "Operator interrupt in line nnnn".)

Interrupts a running program and returns you to interactive mode.

A.2.6 DELETE

DELETE line#1{,line#2}

Deletes the program line(s) between and including those specified.

A.2.7 LIST

LIST {line#1{,line#2}}

Lists the entire program in memory, or the line(s) between and including those specified.

A.2.8 LOAD

LOAD filespec

The default file extension is .BAS. Loads the specified program into memory from the disk.

A.2.9 NEW

NEW

Clears memory of all source code, object code, user symbols and variables.

A.2.10 RUN

RUN

Checks a flag to determine if the program has been compiled. If not, the program is compiled. RUN then initiates the execution of the program in memory, starting at the lowest line number.

A.2.11 SAVE

SAVE filespec[.RUN]

Saves the program in memory on the disk with the specified name and default extension of .BAS. If the extension .RUN is specified, the object code is saved on the disk with the program name and extension .RUN.

A.2.12 SINGLE-STEP (LINEFEED)

(Press linefeed key)

Executes the current program line and returns you to interactive mode.

A.3 ALPHABASIC STATEMENTS

Statements are used within the source program. Some of them may be used as direct statements. Note that those statements that accept a file specification accept it as a string literal (for example: "DSK0:INIT.BAS") enclosed in quotation marks, as a string variable (for example: FSPEC$), or a string expression (for example: MID$(A$,1,6) which evaluates to a valid file specification.

A.3.1 ALLOCATE

ALLOCATE filespec, number-of-blocks

Allocates a random file on the disk with the specified number of disk blocks. Then you can use the OPEN statement to open the file for random processing.

A.3.2 CHAIN

CHAIN filespec

Causes the current program to be deleted from memory and the program with the specified filename, and the optional device name and extension, to be loaded into memory and executed.

A.3.3 CLOSE

CLOSE #file-channel

Closes the specified file. No further reading to or writing from that file is allowed until another OPEN statement for that file is processed. All files are automatically closed at program completion.

A.3.4 DATA

DATA data1{,data2,...dataN}

Stores numeric constants or string literals in a dedicated memory area at program execution. The DATA statement enables data to be an integral part of the program. Numeric items may not contain commas within them. Individual data strings or constants are separated by commas in the DATA statements. The data between each pair of commas is drawn consecutively from the dedicated memory area and assigned to the respective READ statement variable until either data is exhausted or no further READ statements occur. If data is exhausted, using RESTORE reinitializes the data placed in the data pool by the DATA statement. Notice the READ and RESTORE commands below.

A.3.5 DIM

DIM variable1(expr1{,expr2,...exprN}){,variableN(expr1{,expr2,...exprN})}

Defines one or more arrays which are allocated at the time of program execution. String and/or numeric variables are allowed, and any number of

subscripts may be used to define the separate levels of each array. Subscripts may be any legal numeric expression containing variables or constants.

A.3.6 END

END

Causes the program to terminate execution. It is not required unless other program lines (e.g., subroutines) follow the program end.

A.3.7 FILEBASE

FILEBASE n

Tells BASIC that the first record in the file is record number n, not record number zero. You may use any numeric argument with FILEBASE.

FILEBASE does not associate its value with a specific file, but only takes effect when the program it is in is executed.

A.3.8 FOR, TO, STEP and NEXT

FOR variable = expression TO expression {STEP {-}value}

(program statements,if any, to be affected by the loop)

NEXT {variable}

Initializes a loop during program execution. Variables may be subscripted. STEP defaults to positive 1 if not specified. If STEP is negative, the values must be specified from larger to smaller (i.e., FOR A=10 TO 1 STEP -1; FOR X=-1 TO -10 STEP -2). The statement NEXT (with the optional variable specifying the particular loop) continues the loop until the second value (following TO) is reached by incrementation or decrementation.

A.3.9 GOSUB or CALL and RETURN

GOSUB label or line#	CALL label or line#
specified subroutine	specified subroutine
RETURN	RETURN

GOSUB and CALL perform identical functions. If a label is specified, the

subroutine must be prefaced by the label name and a colon: otherwise, the first line of the subroutine must start with the specified line#. Subroutines may be nested. RETURN terminates the subroutine and returns control to the statement following the GOSUB or CALL statement.

A.3.10 GOTO

GOTO label or line#

Unconditional transfer statement transfers control to the label or line number indicated. It may also be written GO TO.

A.3.11 IF, THEN and ELSE

IF expression THEN {statement}{line#}{label} {ELSE {statement}{line#}{label}}
IF expression {statement}{line#}{label} {ELSE {statement}{line#}{label}}

The conditional processing statement with many different optional formats. Other AlphaBASIC statements are legal within the statement. Also, IF-THEN-ELSE statements may be nested to any depth.

A.3.12 INPUT

INPUT {"prompt-string",}variable1{,variable2,...variableN}

Allows data to be entered from your terminal and assigned to the specified numeric or string variable(s) during program run. Input is prompted with a question mark unless you supply a text prompt. Commas are the terminators between data items you input. A carriage return from the terminal without entering data aborts input and leaves all the following variables unchanged.

INPUT #file-channel,variable1{,variable2,...variableN}

enters data from the file associated with the specified file channel. For use with sequential files.

A.3.13 INPUT LINE

INPUT LINE {"prompt-string",}variable1

Main purpose is to read entire line of input into string variables. Acts the same as INPUT for numeric variables. For string variables, allows an entire line of data, except carriage return and linefeed, to be entered verbatim from your terminal during program execution and assigned to the specified string variable. No quotation marks for literal strings are

required. There is no prompt symbol by default, but you can define the prompt text in the statement.

INPUT LINE #file-channel,variable1

enters data from the file associated with the specified file number. For use with sequential files.

A.3.14 KILL

KILL filespec

Erases the specified file from the disk. A file can be killed without being opened or closed. Only files in your account or project can be killed.

A.3.15 LET

LET variable = expression

Assigns a value to a variable. Use of the actual word LET is optional (i.e., LET A=1 may be written A=1).

A.3.16 LOOKUP

LOOKUP filespec, variable

Looks for the specified file. If found, the specified variable assumes the number of disk blocks the file contains. If not found, the specified variable assumes 0. If the file is sequential, variable contains positive number; if file is random, variable contains negative number.

A.3.17 ON ERROR GOTO and RESUME

ON ERROR GOTO label/line#	ON ERROR GOTO {0}
	(Disables further error trapping)
RESUME label/line#	RESUME
(Branch to area of program resumption)	(Branch to line causing error)

As a result of a program error, control is transferred to the specified label or line number for processing. In the error trapping routine, the statement RESUME causes the program to resume at the statement causing the error, or at the label or line number specified.

A.3.18 ON-GOSUB or CALL

```
ON expression GOSUB label/line#1{,label/line#2,...label/line#N}
ON expression CALL label/line#1{,label/line#2,...label/line#N}
```

Enables multi-path branching to subroutines based on the positive integer value of the expression (i.e., on expression=1, branch to label/line#1, etc.).

A.3.19 ON-GOTO

```
ON expression GOTO label/line#1{,label/line#2,...label/line#N}
```

Enables multi-path transfers of program control based on the positive integer value of the expression (i.e., on expression=1, branch to label/line#1, etc.).

A.3.20 OPEN

```
OPEN #file-channel,filespec,mode{,record-size,record#-variable}
```

Assigns a specific integer file-channel number to the specified file and also specifies whether the file is being opened for input, output or random (both input and output) operations, or ISAM operations. (Mode may be: INPUT, OUTPUT, RANDOM, INDEXED, or INDEXED'EXCLUSIVE.) If the mode selected is RANDOM, record-size is an expression that specifies the logical record size, and record#-variable is a variable that maintains the current logical record number.

A.3.21 PRINT

```
PRINT {expression-list}
```

or:

```
? {expression-list}
```

Outputs a blank line, or the expression(s) specified. A semicolon or comma at the end of the list of expressions inhibits carriage return/linefeed after a PRINT output. The expressions to be printed may consist of numeric or string expressions, string or numeric variables, numeric constants, string literals, functions, or combinations of the above. String literals must be placed within quotation marks. The word PRINT may be replaced with the question mark symbol.

A.3.22 PRINT USING

```
variable = expression USING format-string
PRINT USING format-string, expression-list
PRINT expression USING format-string
```

For formatted output where the characters are specifically positioned. The string contains one or more special formatting characters to control the printed output, such as character placement, field size, leading asterisks, floating dollar signs, numeric sign, commas, exponential format and numeric string size. The list is made of the expression(s) you want printed.

A.3.23 RANDOMIZE

```
RANDOMIZE
```

Resets the random number generator seed to begin a new random number sequence starting with the next RND(X) function call.

A.3.24 READ and RESTORE

```
READ variable1{,variable2,...variableN}
```

Assigns next group(s) of data in dedicated memory to variable(s).

```
RESTORE
```

Readies data in the dedicated memory area for rereading from the beginning of the data pool.

READ and RESTORE, along with the DATA statement, enable data to be an integral part of the program. The data in the data pool is drawn consecutively from the dedicated memory area by READ and assigned to the respective READ statement variable until either data is exhausted or no further READ statements occur. If data is exhausted, using RESTORE reinitializes the data pool. See the DATA statement above.

READ has another operation within the file I/O system which has no relation to the DATA or RESTORE statements.

```
READ #file-channel,variable1{,variable2,...variableN}
```

This operation of the READ statement reads into the specified variable(s) data from the random file associated with the specified file channel. It reads from the logical record whose record number is currently in the record#-variable defined by the OPEN statement for that file.

A.3.25 SCALE

SCALE value

Sets the number of decimal places by which all floating point numbers are offset when they are calculated, to minimize error propagation.

A.3.26 SIGNIFICANCE

SIGNIFICANCE value

where the value is between 1 and 11. Sets the maximum number of printable digits in unformatted numbers. Numbers are calculated in full 11-digit accuracy, then rounded off to the value of significance just prior to printing. Not in effect when PRINT USING statements are being used.

A.3.27 STOP

STOP

Suspends program execution and returns you to interactive mode or AMOS monitor level, depending on where you were at program commencement. You see a message identifying the line of the program stop. In the compiler mode, from the AMOS monitor level, the message adds, "Enter CR to continue:". From the interactive mode of BASIC, the program may be continued by the CONT or single-step (linefeed) commands.

A.3.28 STRSIZ

STRSIZ value

Assigns the maximum size in bytes of all following strings. STRSIZ must be assigned a positive integer.

A.3.29 WRITE

WRITE #file-channel,variable1{,variable2,...variableN}

Writes the data currently assigned to the specified variable(s) into the random file associated with the specified file channel. It writes into the logical record whose record number is currently in the record#-variable defined by the OPEN statement for the file.

A.3.30 XCALL

XCALL routine,{argument1{,argument2,...argumentN}}

Calls an assembly language program as a BASIC subroutine. The argument may be a variable or an expression.

A.4 ALPHABASIC FUNCTION STATEMENTS

The following is a list of the AlphaBASIC functions. Functions almost always require an argument. Depending on the function, the argument may be a variable, a string or a fixed value. The argument is used either to control the function or as data upon which the function operates.

We have organized the AlphaBASIC functions into two categories: those that accept numeric arguments and those that accept string arguments. However, be aware that because of the mode independence of AlphaBASIC, such distinctions are often hazy. For example, although the square root function, SQR, is a numeric function, you can give it a string argument as long as the mode independence feature can convert that string to numeric data. For example:

```
PRINT SQR(16)
 4

PRINT SQR("16")
 4
```

In the same way, you can use the string function LEFT$ to excerpt characters from numeric data as if that data were a string:

```
PRINT LEFT$("123",2)
23

PRINT LEFT$(456,2)
56
```

A.4.1 NUMERIC FUNCTIONS

These functions require arguments which can be evaluated as numbers. X may be any expression, but if it contains string variables or literals, they must represent numeric values. For example: ABS("11"+2) returns 13.

A.4.1.1 ABS(X) - Returns the absolute value of the argument X.

A.4.1.2 CHR(X) - Returns a single character having the ASCII decimal value of X. Only one character is generated for each CHR function call.

A.4.1.3 EXP(X) - Returns the constant e (2.71828) raised to the power X.

A.4.1.4 FACT(X) - Returns the factorial of X.

A.4.1.5 FIX(X) - Returns the integer part of X (fractional part truncated).

A.4.1.6 INT(X) - Returns the largest integer less than or equal to the argument X.

A.4.1.7 LOG(X) - Returns the natural (base e) logarithm of the argument X.

A.4.1.8 LOG10 - Returns the decimal (base 10) logarithm of the argument X.

A.4.1.9 RND(X) - Returns a random number generated by a pseudo-random number generator based on the seed. The argument X controls the number to be returned. If X is negative, it is used as the seed to start a new sequence of numbers. If X is zero or positive, the next number in the sequence is returned, depending on the current value of the seed (this is the normal mode).

A.4.1.10 SGN(X) - Returns a value of -1, 0 or 1 depending on the sign of the argument X. Gives -1 if X is negative, 0 if X is 0 and 1 if X is positive.

A.4.1.11 SQR(X) - Returns the square root of the argument X.

A.4.1.12 STR(X) or STR$(X) - Returns a string which is the character representation of the numeric expression X. No leading space is returned for positive numbers.

A.4.2 TRIGONOMETRIC FUNCTIONS

The following trig functions are implemented in full 11-digit accuracy:

SIN(X)	Sine of X
COS(X)	Cosine of X
TAN(X)	Tangent of X
ATN(X)	Arctangent of X
ASN(X)	Arcsine of X
ACS(X)	Arccosine of X
DATN(X,Y)	Double arctangent of X,Y

A.4.3 CONTROL FUNCTIONS

The following control functions test certain file conditions and control and return information about certain system operations.

A.4.3.1 DATE - The DATE system function sets and returns the two-word system date.

```
DATE = expression       !sets system date to expr
   A = DATE             !returns system date into A
```

A.4.3.2 TIME - The TIME system function requires no argument and is used to set and retrieve the time of day as stored in the system monitor communications area. The time is stored as a two-word integer representing the number of clock ticks since midnight. One clock tick represents one interrupt from the CPU line clock (usually 60 or 50 Hz). Dividing the time by the clock rate gives the number of seconds since midnight. Converting this to current time is then accomplished by successive divisions by 60 to get minutes, and again by 60 to get hours.

```
TIME = exprpression     !sets time-of-day in system to expr
   A = TIME             !returns time-of-day in clock ticks into A
```

A.4.3.3 BYTE and WORD - The BYTE and WORD system functions allow you to inspect and alter any memory locations within the 64K memory addressing range of the machine. The BYTE functions deal with 8 bits of data in the range of 0-255, and the WORD functions deal with 16 bits of data in the range of 0-65535, inclusive. Any unused bits are ignored, with no error message.

```
BYTE(X) = expr    !writes the low byte of expr into decimal memory loc X
WORD(X) = expr    !writes the low word of expr into decimal memory loc X
      A = BYTE(X) !reads decimal memory loc X and places the byte into A
      A = WORD(X) !reads decimal memory loc X and places the word into A
```

A.4.3.4 EOF(X) - The EOF (end-of-file) function returns a value giving the status of a sequential file open for input whose file number is X. The values returned by the EOF function are:

```
-1  if the file is not open or the file number X is zero
 0  if the file is not yet at end-of-file during inputs
 1  if the file has reached the end-of-file condition
```

EOF should only be tested for sequential input files.

A.4.3.5 ERF(X) - Returns a file error-condition code. If the returned value of X is not zero, an error or abnormal condition exists as a result of the preceding file operation. (See Chapter 19 for a list of the error codes returned by ERF.)

A.4.3.6 ERR(X) - Returns a status code for X which refers to program status during error trapping. (See Chapter 17 for a list of the error codes returned by ERR.)

A.4.3.7 MEM(X) - Returns a positive integer which specifies the number of bytes currently in use for various memory areas used by the compiler system.

A.4.3.8 SPACE(X) or SPACE$(X) - Returns a string of X spaces in length.

A.4.4 STRING FUNCTIONS

The arguments of these functions are literal strings or string variables. For example, if A$ is "Now is the time", the LEN function (which computes the number of characters in a string) returns 15 in both of these cases:

```
PRINT LEN("Now is the time")

PRINT LEN(A$)
```

A.4.4.1 ASC(A$) - Returns the ASCII decimal value of the first character in string A$. The function ASC("C") returns the ASCII decimal value of the character C, 67.

A.4.4.2 INSTR(X,A$,B$) - Performs a search for the substring B$ within the string A$, beginning at the Xth character position. It returns a value of zero if B$ is not in A$, or the character position if B$ is found within A$. Character position is measured from the start of the string, with the first character position represented as one.

A.4.4.3 LCS(A$) - Returns a string which is identical to the argument string (A$), with all characters translated to lower case.

A.4.4.4 LEFT(A$,X) or LEFT$(A$,X) - LEFT$(A$,X) Returns the leftmost X characters of the string expression A$.

A.4.4.5 LEN(A$) - Returns the number of characters in the string expression A$.

A.4.4.6 MID(A$,X,Y) or MID$(A$,X,Y) - Returns the substring composed of the characters of the string expression A$ starting at the Xth character and extending for Y characters. A null string is returned if X > LEN(A$).

A.4.4.7 RIGHT(A$,X) or RIGHT$(A$,X) - Returns the rightmost X characters of the string expression A$.

A.4.4.8 UCS(A$) - Returns a string which is identical to the argument string (A$), with all characters translated to upper case.

A.4.4.9 VAL(A$) - Returns the numeric value of the string expression A$ converted under normal BASIC format rules.

APPENDIX B

MESSAGES OUTPUT BY ALPHABASIC

Below is a complete list of all messages output by the AlphaBASIC system (i.e., BASIC, RUN, and COMPIL), along with a brief explanation of each message.

Bitmap kaput
: Your program attempted a file operation (OPEN, ALLOCATE, etc.) on a device with a bad bitmap.

Break at line n
: The program reached the breakpoint that was set at line n.

COMPILE
: BASIC is telling you that it is compiling your program.

Can't continue
: You have attempted to continue a program which is not stopped at a breakpoint, or which has reached a point where it can go no further (e.g., it has reached an END statement).

Cannot find xxxxxxx
: The program xxxxxxx was not found.

Compile time was x.x seconds.
: BASIC is telling you how long (in elapsed time, not computing time) it took to compile your program.

DELETE what?
: You have specified a DELETE command without specifying what line(s) are to be deleted.

Device does not exist
: The device you specified in a file operation (OPEN< LOOKUP, etc.) does not exist.

?Device driver must be loaded into user or system memory
: If you are accessing a non-DSK device, the appropriate device driver must be loaded into user or system memory.

Device error
An error has occurred on the referenced device.

Device full
The specified device has run out of room during a WRITE, CLOSE, or ALLOCATE operation. Remember that an ALLOCATE requires contiguous disk space, so that a Device full error may occur when there are still a number of non-contiguous blocks available.

Device in use
The specified device is currently assigned to another user.

Device not ready
The specified disk is not ready for use.

Disk not mounted
The specified disk has not been mounted. Mount it via the MOUNT monitor command or via the XMOUNT subroutine.

Divide by zero
Your program attempted to perform a division by zero.

Duplicate label
Your program has defined the same label name more than once.

*** End of Program ***
You have reached the end of the program during single-stepping.

Enter <CR> to continue:
You have reached a STOP statement in your program. You may continue from the STOP statement via a carriage-return, or may abort the run via a Control-C.

?Error in Error Trapping
An error occurred while you were in the error trapping routine.

File already exists
Your program tried to create a file which already exists.

File already open
You have attempted to open a file that is already open on the same file number.

File not found
BASIC was unable to locate the specified file.

Filespec error
The file specification you gave in a file operation (OPEN, LOOKUP, etc.) is in error. All file specifications must conform to the system standard (i.e., Devn:Filename.Extension[p,pn]).

File type mismatch
: Your program tried to perform a sequential operation on a random file or vice-versa.

Floating point overflow
: A floating point overflow occurred during a calculation.

IO to unopened file
: Your program tried to perform input or output to a file that was not open.

Illegal GOTO or GOSUB
: The format of the GOTO or GOSUB statement is invalid.

Illegal NEXT variable
: The variable used in the NEXT statement is not valid (e.g., not floating point).

Illegal PRINT USING format
: The edit format used in a PRINT USING statement is invalid.

Illegal SCALE argument
: The argument given in a SCALE statement is invalid (the argument must range between -30 and +30).

Illegal STRSIZ argument
: The argument given in a STRSIZ statement is invalid.

Illegal TAB format
: Your program has incorrectly specified a TAB function.

Illegal expression
: The specified expression is not valid.

Illegal function value
: The specified function value is not valid for the particular function.

Illegal line number
: The specified line number is invalid (e.g., not between 1 and 65534).

Illegal or undefined variable in overlay
: The variable specified in a MAP statement overlay (via @) has not been previously defined, or is not a mapped variable.

Illegal record number
: The relative record number specified in a random file processing statement (i.e., READ or WRITE) is either less than the current FILEBASE or outside of the file.

Illegal size for variable type
: The specified variable size is not valid for the particular variable type. Floating point variables must be size 6, and binary variables must have sizes 1 through 5.

Illegal subroutine name
: The name specified as a subroutine is not valid.

Illegal subscript
: The subscript expression is not valid.

Illegal type code
: The variable type code specified in a MAP statement is not one of the valid types.

Illegal user code
: The specified PPN was not found on the specified device, or is not in a valid format.

Insufficient memory to load program xxxxxxx
: The RUN program did not find enough free memory to be able to load the specified program.

Invalid filename
: The specified filename was not a legal filename.

[Invalid syntax code]
: An internal error has occurred in BASIC. Please notify Alpha Micro of this error. Provide an example of what caused it.

Line number must be from 1-65534
: The line number entered was not in the range of legal line numbers.

Line x not found
: The specified line was not found for a DELETE, LIST, etc., operation.

NEXT without FOR
: A NEXT statement was encountered without a matching FOR statement.

No breakpoints set
: BASIC is telling you that there are currently no breakpoints set in your program.

No source program in text buffer
: You tried to compile when there was no program in memory.

Operator interrupt
: You typed a Control-C to interrupt program execution.

Out of data
: A READ statement was encountered after the data in all DATA statements had been used.

Out of memory
: BASIC is telling you that it has run out of memory in which to execute your program.

Out of memory - Compilation aborted
: COMPIL is telling you that it does not have enough free memory to finish compiling your program.

Program name:
: You tried to SAVE or LOAD a program without providing a filename. Enter the filename at this point.

Protection violation
: Your program tried to write into another account where you do not have write privileges.

RESUME without error
: A RESUME statement was encountered, but no error has occurred.

RETURN without GOSUB
: A return statement was encountered, but not corresponding GOSUB has been executed.

Record size overflow
: Your program tried to read a file rcord into a variable larger than the file record size.

Redimensioned array
: You tried to redimension an array.

Runtime was x.x seconds
: BASIC is telling you how long it took to run your program.

?Runtime package (RUN.PRG) not found
: BASIC or COMPIL was unable to locate the run-time package, or did not have sufficient memory in which to load it.

Source line overflow
: A line in the source program, including continuation lines, exceeds 500 characters.

Stack overflow
: BASIC's internal stack has overflowed. This is most often caused by such operations as nesting GOSUBs too deep, or branching out of FOR-NEXT loops.

Subroutine not found
: The specified subroutine could not be found.

Subscript out of range
: The specified subscript is outside the range specified in the DIM or MAP statement for the subscripted variable.

Syntax error
: The syntax of the specified line is invalid.

System commands are illegal within the source program
: BASIC system commands (LOAD, DELETE, LIST, etc.) are not valid within a BASIC source program.

System error
: A system error has occurred during the execution of the specified line. System error is used as a catch-all error message for a variety of unlikely occurrences.

Temporarily all arrays must be less than 32K
: The array size you specified is larger than 32K bytes.

Undefined line number or label
: The line number or label specified in a GOTO or GOSUB statement is not defined within the program.

Write protected
: Your program tried to write on a write-protected device

Wrong number of subscripts
: The number of subscripts specified is not the same as the number defined in the DIM or MAP statement for the subscripted variable.

APPENDIX C

RESERVED WORDS

Below is a list of the reserved words used by the BASIC compiler. Some of these reserved words designate routines that have not been implemented at this time. However, you must not use any of these reserved words as variable names or labels. NOTE: This restriction applies to string variables as well as numeric variables. (For instance, END$ and END are both illegal variable names.)

ABS	absolute value
ACS	arccosine
ALLOCATE	allocate file
AND	logical AND
ASC	ASCII value
ASN	arcsine
ATN	arctangent
BREAK	set breakpoint
BYE	exit to monitor
BYTE	memory byte
CALL	call subroutine
CHAIN	chain next program
CHR	character value
CHR$	character value
CLOSE	close file
COMPILE	compile program
CONT	continue execution
COS	cosine
DATA	data statement
DATE	system date
DATN	double arctangent
DEF	define function
DELETE	delete lines
DIM	dimension
ELSE	else
END	end of program
EOF	end of file
EQV	logical equivalence
ERF	file error
ERR	error status

ERROR	error
EXP	exponentiation
EXPAND	expand mode on
FACT	factorial
FILEBASE	file base offset
FIX	fix
FOR	loop initiation
GO	program jump
GOSUB	call subroutine
GOTO	program jump
IF	conditional test
INDEXED	indexed
INPUT	input data
INSTR	search string
INT	integer
IO	input/output
ISAM	ISAM control
KILL	kill file
LCS	lower case string
LEFT	left string
LEFT$	left string
LEN	length string
LET	variable assignment
LINE	line
LIST	list text
LOAD	load program
LOG	natural logarithm
LOG10	base 10 logarithm
LOOKUP	lookup file
MAP	map variable
MAX	maximum value
MEM	memory size
MID	mid string
MID$	mid string
MIN	minimum value
NEW	new program
NEXT	loop termination
NOEXPAND	expand mode off
NOT	logical complement
ON	on (GOTO, GOSUB, ERROR)
OPEN	open file
OR	logical OR
OUTPUT	output
PRINT	print on terminal/file
RANDOM	random
RANDOMIZE	randomize RND function
READ	read data
REM	remark line
RESTORE	restore data
RESUME	resume after error
RETURN	subroutine exit
RIGHT	right string
RIGHT$	right string

RND	random number
RUN	run program
SAVE	save program
SCALE	set scale factor
SGN	sign
SIGNIFICANCE	set significance
SIN	sine
SPACE	spaces
SPACE$	spaces
SQR	square root
STEP	step
STOP	stop program
STR	numeric to string conversion
STR$	numeric to string conversion
STRSIZ	set string size
SUB	sub (GOSUB)
TAB	tab
TAN	tangent
THEN	optional statement verb
TIME	system time
TO	to
UCS	upper case string
USING	using
VAL	string to numeric conversion
WORD	memory word
WRITE	write file
XCALL	external subroutine call
XOR	logical XOR

APPENDIX D

THE ASCII CHARACTER SET

The next few pages contain charts that list the complete ASCII character set. We provide the octal, decimal and hexadecimal representations of the ASCII values.

Note that the first 32 characters are non-printing Control-characters.

THE CONTROL CHARACTERS

?CHR(,)

CHARACTER	OCTAL	DECIMAL	HEX	MEANING
NULL	000	0	00	Null (fill character)
SOH	001	1	01	Start of Heading
STX	002	2	02	Start of Text
ETX	003	3	03	End of Text
ECT	004	4	04	End of Transmission
ENQ	005	5	05	Enquiry
ACK	006	6	06	Acknowledge
BEL	007	7	07	Bell code
BS	010	8	08	Back Space
HT	011	9	09	Horizontal Tab
LF	012	10	0A	Line Feed
VT	013	11	0B	Vertical Tab
FF	014	12	0C	Form Feed
CR	015	13	0D	Carriage Return
SO	016	14	0E	Shift Out
SI	017	15	0F	Shift In
DLE	020	16	10	Data Link Escape
DC1	021	17	11	Device Control 1
DC2	022	18	12	Device Control 2
DC3	023	19	13	Device Control 3
DC4	024	20	14	Device Control 4
NAK	025	21	15	Negative Acknowledge
SYN	026	22	16	Synchronous Idle
ETB	027	23	17	End of Transmission Blocks
CAN	030	24	18	Cancel
EM	031	25	19	End of Medium
SS	032	26	1A	Special Sequence
ESC	033	27	1B	Escape
FS	034	28	1C	File Separator
GS	035	29	1D	Group Separator
RS	036	30	1E	Record Separator
US	037	31	1F	Unit Separator

PRINTING CHARACTERS

CHARACTER	OCTAL	DECIMAL	HEX	MEANING
SP	040	32	20	Space
!	041	33	21	Exclamation Mark
"	042	34	22	Quotation Mark
#	043	35	23	Number Sign
$	044	36	24	Dollar Sign
%	045	37	25	Percent Sign
&	046	38	26	Ampersand
'	047	39	27	Apostrophe
(	050	40	28	Opening Parenthesis
)	051	41	29	Closing Parenthesis
*	052	42	2A	Asterisk
+	053	43	2B	Plus
,	054	44	2C	Comma
-	055	45	2D	Hyphen or Minus
.	056	46	2E	Period
/	057	47	2F	Slash
0	060	48	30	Zero
1	061	49	31	One
2	062	50	32	Two
3	063	51	33	Three
4	064	52	34	Four
5	065	53	35	Five
6	066	54	36	Six
7	067	55	37	Seven
8	070	56	38	Eight
9	071	57	39	Nine
:	072	58	3A	Colon
;	073	59	3B	Semicolon
<	074	60	3C	Less Than
=	075	61	3D	Sign
>	076	62	3E	Than
?	077	63	3F	Question Mark
@	100	64	40	Commercial At

CHARACTER	OCTAL	DECIMAL	HEX	MEANING
A	101	65	41	Upper Case Letter
B	102	66	42	Upper Case Letter
C	103	67	43	Upper Case Letter
D	104	68	44	Upper Case Letter
E	105	69	45	Upper Case Letter
F	106	70	46	Upper Case Letter
G	107	71	47	Upper Case Letter
H	110	72	48	Upper Case Letter
I	111	73	49	Upper Case Letter
J	112	74	4A	Upper Case Letter
K	113	75	4B	Upper Case Letter
L	114	76	4C	Upper Case Letter
M	115	77	4D	Upper Case Letter
N	116	78	4E	Upper Case Letter
O	117	79	4F	Upper Case Letter
P	120	80	50	Upper Case Letter
Q	121	81	51	Upper Case Letter
R	122	82	52	Upper Case Letter
S	123	83	53	Upper Case Letter
T	124	84	54	Upper Case Letter
U	125	85	55	Upper Case Letter
V	126	86	56	Upper Case Letter
W	127	87	57	Upper Case Letter
X	130	88	58	Upper Case Letter
Y	131	89	59	Upper Case Letter
Z	132	90	5A	Upper Case Letter
[	133	91	5B	Opening Bracket
\	134	92	5C	Back Slash
]	135	93	5D	Closing Bracket
^	136	94	5E	Circumflex
←	137	95	5F	Underline
`	140	96	60	Grave Accent
a	141	97	61	Lower Case Letter
b	142	98	62	Lower Case Letter
c	143	99	63	Lower Case Letter
d	144	100	64	Lower Case Letter
e	145	101	65	Lower Case Letter
f	146	102	66	Lower Case Letter
g	147	103	67	Lower Case Letter
h	150	104	68	Lower Case Letter
i	151	105	69	Lower Case Letter
j	152	106	6A	Lower Case Letter
k	153	107	6B	Lower Case Letter
l	154	108	6C	Lower Case Letter
m	155	109	6D	Lower Case Letter
n	156	110	6E	Lower Case Letter
o	157	111	6F	Lower Case Letter

CHARACTER	OCTAL	DECIMAL	HEX	MEANING
p	160	112	70	Lower Case Letter
q	161	113	71	Lower Case Letter
r	162	114	72	Lower Case Letter
s	163	115	73	Lower Case Letter
t	164	116	74	Lower Case Letter
u	165	117	75	Lower Case Letter
v	166	118	76	Lower Case Letter
w	167	119	77	Lower Case Letter
x	170	120	78	Lower Case Letter
y	171	121	79	Lower Case Letter
z	172	122	7A	Lower Case Letter
{	173	123	7B	Opening Brace
|	174	124	7C	Vertical Line
}	175	125	7D	Closing Brace
~	176	126	7E	Tilde
DEL	177	127	7F	Delete

APPENDIX E

SAMPLE PROGRAM - NUMERIC CONVERSION FOR BASES 2 - 16.

This appendix contains a sample AlphaBASIC program that converts a number between one number base and another. You may convert numbers from the binary through hexadecimal (2-16) number bases to another number base in the same range. For example, you can translate an octal number to its hexadecimal form, or vice versa. Below is a sample run of the program:

```
"CONVRT"--CONVERT BETWEEN NUMBER BASES
          Enter positive numbers, any base from 2 to 16
          (Enter a zero to FROM BASE? to end the program)

FROM BASE? 10 (RET)
  TO BASE? 2 (RET)
BASE 10 NUMBER? 364 (RET)
BASE 2 NUMBER = 101101100

FROM BASE? ^C
Operator interrupt in line 2015
```

The program:

```
10 ! "CONVRT" -- PROGRAM TO CONVERT BETWEEN NUMBER BASES

100 MAP1 IN'VARIABLES                          !INPUT BASE VARIABLES
105   MAP2 IN'NUMBER,S,50                      !input number string
110   MAP2 IN'BASE,F                           !base of input number,
                                               !   2 through 16 valid
200 MAP1 OUT'VARIABLES                         !OUTPUT BASE VARIABLES
205   MAP2 OUT'NUMBER,S,50                     !output number string
210   MAP2 OUT'BASE,F                          !base of output number,
                                               !   2 through 16 valid
300 MAP1 VALID'DIGIT,S,16,"0123456789ABCDEF" !VALID DIGITS
                                               !base x contains x leftmost
                                               !   digits
```

```
 900 MAP1 MISC'VARIABLES                          !MISCELLANEOUS VARIABLES
 905   MAP2 BASE10'NUMBER,F                       !input string converted to
                                                  !   base 10
 910   MAP2 ERROR'FLAG,F                          !flag set if invalid digit
                                                  !   found
 915   MAP2 LEADING'BLANK,F                       !flag reset when first non-
                                                  !   blank character found
 920   MAP2 LOOP'1,F                              !FOR-NEXT index #1
 925   MAP2 LOOP'2,F                              !FOR-NEXT index #2
 930   MAP2 WORK'1,F                              !scratch variable used in
                                                  !   conversion to output
                                                  !   base
                                                  !START OF PROGRAM
1000 DISPLAY'BANNER:
1005   PRINT CHR$(34);"CONVRT";CHR$(34);"--CONVERT BETWEEN NUMBER BASES"
1010   PRINT TAB(10);"Enter positive numbers, any base from 2 to 16"
1015   PRINT TAB(10);"(Enter a zero to FROM BASE? to end the program.)"

2000 ENTER'IN'BASE:                               !ENTER INPUT BASE
2005   PRINT                                      !blank line between header
                                                  !   or previous conversion
2010   IN'BASE=0                                  !set to zero in case of
                                                  !   carriage return
2015   INPUT "FROM BASE? ",IN'BASE                !enter input base
2020   IF IN'BASE=0 GOTO END'OF'PROGRAM           !if zero/carriage return, end
2025   IF IN'BASE>=2 AND IN'BASE<=16 GOTO ENTER'OUT'BASE
2030   PRINT CHR$(7);"INVALID BASE!"              !bases 2 to 16 only
2035   GOTO ENTER'IN'BASE                         !re-enter base

2200 ENTER'OUT'BASE:                              !ENTER OUTPUT BASE
2205   OUT'BASE=0                                 !set to zero in case of
                                                  !   carriage return
2210   INPUT "  TO BASE? ",OUT'BASE               !enter output base
2215   IF OUT'BASE=0 GOTO ENTER'IN'BASE           !if zero/carriage return,
                                                  !   re-enter input base
2220   IF OUT'BASE>=2 AND OUT'BASE<=16 GOTO ENTER'IN'NUMBER
2225   PRINT CHR$(7);"INVALID BASE!"              !bases 2 to 16 only
2230   GOTO ENTER'OUT'BASE                        !re-enter output base

2400 ENTER'IN'NUMBER:                             !ENTER INPUT NUMBER
2405   IN'NUMBER=""                               !set to null string in case
                                                  !   of carriage return
2410   PRINT "BASE";IN'BASE;"NUMBER? ";           !prompt for input number
2415   INPUT LINE "",IN'NUMBER                    !enter input number

2420 VALIDATE'NUMBER:                             !CHECK/CONVERT INPUT NUMBER
2425   LEADING'BLANK=1 : ERROR'FLAG=0             !initialize flags
2430   BASE10'NUMBER=0                            !initialize base 10 number
2435   FOR LOOP'1=1 TO LEN(IN'NUMBER)             !check one character at a time
2440     IF IN'NUMBER[LOOP'1;1]<>" " GOTO NON'BLANK     !skip if non-blank
2445     IF LEADING'BLANK=0 LOOP'1=LEN(IN'NUMBER)       !if not leading
2450     GOTO END'LOOP'1                                !   blank, end
                                                        !   conversion of
```

```
                                                        !   input number;
                                                        ! otherwise skip it
2455 NON'BLANK:                                 !PROCESS NON-BLANK CHARACTERS
2460     LEADING'BLANK=0                        !reset leading blank flag,
                                                !   non-blank found
2465     ERROR'FLAG=1                           !assume invalid character
                                                !   until valid one found
2470     FOR LOOP'2=1 TO IN'BASE                !CHECK FOR VALID DIGIT using
                                                !   valid character list
2475       IF IN'NUMBER[LOOP'1;1]<>VALID'DIGIT[LOOP'2;1] GOTO END'LOOP'2
                                                        !invalid character,
                                                        !   try next
2480       BASE10'NUMBER=BASE10'NUMBER*IN'BASE+LOOP'2-1 !convert and shift
2485       ERROR'FLAG=0                           !reset--valid found
2490       LOOP'2=IN'BASE                         !no need to check
                                                        !   more digits
2495 END'LOOP'2:
2500     NEXT LOOP'2                                    !next valid or end
2505     IF ERROR'FLAG<>0 LOOP'1=LEN(IN'NUMBER)         !if bad character
                                                        !   found, check
                                                        !   no further
2510 END'LOOP'1:
2515   NEXT LOOP'1                                      !next character in
                                                        !   input string
                                                        !   or end
2520   IF ERROR'FLAG=0 GOTO CALCULATE'OUT'NUMBER
2525   PRINT CHR$(7);"INVALID BASE";IN'BASE;"NUMBER!"   !bad character
2530   GOTO ENTER'IN'BASE                               !found, display
                                                        !message and
                                                        !start over

2600 CALCULATE'OUT'NUMBER:                      !CONVERT TO OUTPUT BASE
2605   OUT'NUMBER=""                            !start with null string
2610 CONTINUE'CALCULATION:
2615   WORK'1=INT(BASE10'NUMBER/OUT'BASE)       !remainder of number/base is
                                                !   base 10 value of next
                                                !   digit going from right
                                                !   to left
2620   OUT'NUMBER=VALID'DIGIT[1+BASE10'NUMBER-WORK'1*OUT'BASE;1]+OUT'NUMBER
                                                !"1+" to adjust for position
                                                !   in valid digits string
2625   BASE10'NUMBER=WORK'1                     !new number is integer part
                                                !   of number/base
2630   IF BASE10'NUMBER<>0 GOTO CONTINUE'CALCULATION    !done when new
                                                        !   number = 0
2800 PRINT'OUT'NUMBER:                          !PRINT OUTPUT NUMBER
2805   PRINT "BASE";OUT'BASE;"NUMBER = ";OUT'NUMBER

2900 GET'NEXT:
2905   GOTO ENTER'IN'BASE                       !start over from the top

9000 END'OF'PROGRAM:                            !END OF PROGRAM
9010   END                                      !go through the formalities
```

Index