OLYMPIA SYSTEM BOSS

BAL LANGUAGE AND FILE SYSTEM

REFERENCE MANUAL

B. 1006 A

PRELIMINARY ISSUE - OCTOBER 1980

PREFACE

PURPOSE

This manual provides you with the descriptions and data you need to use the BOSS Business BASIC Language (BAL) to develop your applications programs. This document provides a description of the simple, powerful BAL commands; describes rules for the proper usage of the language; and includes programming examples.

THE LANGUAGE

One of the important characteristics of a program is that it must be written in a language that the computer can understand. All computers have a very elementary language called machine language which requires the use of long lists of primitive instructions. The machine language differs with each manufacturer and can even differ from one line of computers to another within the same manufacturer's products.

Darmouth College developed a language having very simple grammatical rules that could be learned quickly. This language has come to be known as BASIC. However, BASIC does not have effective instructions for the entry of business-oriented information and control of display formats for the results. This extended version of BASIC (called BAL) makes more effective use of the entire computer system and provides the necessary tools to program in a commercial environment.

Features of BAL

- o Designed to be used easily by inexperienced programmers.
- o Includes enhanced instructions for input/output which are better adapted to the problems posed in a business environment.
- o Performs arithmetic operations to fourteen-place accuracy.
- o Has a flexible way of handling data files.

Virtual Memory

The information handled by the computer can be stored in the central memory or in peripheral memory units such as mini-floppy disks or large disks. Central memory units are characterized by very fast access times, while peripheral units can contain much larger volumes of data but have a slower access time. A recent advance in computer technology is the concept of virtual memory. Virtual memory systems allow you to write programs without regard for the amount of central memory available. You can manipulate data variables stored on peripheral units as easily as if they were stored in central memory -- only the access time changes. Program segments themselves may be kept on a peripheral unit and recalled when needed.

Programming Methods

There are many possible methods for entering your program into a computer and outputting the results. The BAL system provides an easy-to-use conversational entry method for your programs and data, via the keyboard with the information displayed on a CRT screen. At the same time, the information is recorded on either a minifloppy or a large disk.

Your data can be output to a line printer, the CRT display, or to one of the disks.

Conventions

Notation conventions used in this document are:

Brackets [] surround optional fields in instruction descriptions. The brackets are not to be typed when entering the instruction into the computer.

The symbol b denotes a required blank.

The symbol cr denotes a carriage return.

OLYMPIA SYSTEM BOSS

BAL LANGUAGE AND FILE SYSTEM REFERENCE MANUAL

TABLE OF CONTENTS

TABLE OF CONTENTS, CONT'D

TABLE OF CONTENTS, CONT'D

TABLE OF CONTENTS, CONT'D

TABLE OF CONTENTS, CONT'D

TABLE OF CONTENTS, CONT'D

CHAPTER 1. INTRODUCTION

1.1 INTRODUCTION TO THE BAL LANGUAGE

In order for the computer to perform the desired calculations, you must provide it with the calculation method to be used. It must be in a form the computer can understand, and this form has come to be known as the BAL language.

The "sentences" in this language consist of a command (key) word of 1-6 letters, followed by any other required or optional information. BAL is easy to learn because a small number of command words is sufficient for most programming tasks.

One of the most important features of any programming language is its input/output structure. The objectives of input/output (I/O) are:

Input:	The supplying of data to the computer for program execution.
Output:	The results of the calculations presented in a format desired by the user.

The BAL I/O structure has been designed to facilitate the conversational writing of management applications programs by relatively inexperienced personnel. In general, data is entered on the keyboard and output to the CRT display or printer.

To illustrate BAL, consider the following examples. Their purpose is to illustrate some of the instructions, which are discussed beginning in Chapter 2.

Note that Appendix C is a list of all ~~MICRAL~~ BAL commands indexed to the page where they are described.

1.1.1 Example 1

The first example consists of dividing two numbers. An executable program can be written as follows:

INSTRUCTIONS	FUNCTION	CHAPTER
050 PROGRAM "DIVISION"	Program name	1
100 DCL N,D,Q	Declaration of variables	3
150 SEGMENT 0	Segment number	12
200 REM DIVISION	Remark	1
250 ASK=1: "NUMERATOR"=N	Conversational Input	8
300 ASK=1: "DENOMINATOR"=D	Instructions	8
350 REM CALCULATE QUOTIENT	Remark	1

INSTRUCTIONS	FUNCTION	CHAPTER
400 Q = N/D	Assignment instruction	4
500 PRINT=1: "QUOTIENT",Q	Output of Data	10
600 GOTO 250	Branch Instruction	5
1000 ESEG 0	End Segment	12
1110 END	End of Program	1

Note that the instructions of this program are numbered in ascending order. This numbering is optional.

For the time being, skip the first three instructions and look at instructions 200, 250 and 300. Instruction 200 is a remark (comment) which has no effect on the program, but permits you to add comments to explain various program operations.

Instructions 250 and 300 are conversational instructions to input numerical information. The instruction ASK=1: takes the information which is input and stores it in memory. For example, in executing instruction 250, the computer will display on the CRT screen the data in quotation marks, i.e., it will print NUMERATOR. Then, it will wait for you to enter a value for the numerator. The computer controls each entry. If the character isn't a number, the computer will not accept it. At that moment, you will hear a light "beep" to indicate that there has been an error. Similarly, in executing instruction 300, DENOMINATOR will be displayed on the screen and the computer waits for your input.

Instruction 400 takes the value which was read in by instruction 200 and assigned the variable N, then divides it by the value read in at 300 which was assigned the variable, D. The result of the division of the variable N by the variable D is placed in Q.

Notice that the character used to indicate division is / (the slash). If the instruction involved addition, it would be + (plus), subtraction - (minus), or multiplication * (asterisk). For example, LET C = B * D means multiply the variable B by the variable D giving the result C.

Instruction 500 (PRINT=1:...) enables the computer to print the results. PRINT followed by the names of the variables that one desires to print; here the data enclosed in quotes is printed, then the value of Q will be printed. Thus if Q were computed to be 2000, the printout would be:

```
QUOTIENT 2000
```

The 1 after PRINT indicates that the data is to be written on the video screen. If, in place of the 1, you had put 2, the output would be on a line printer. In the instruction ASK=1:, the number indicates that it will use the video screen. BAL does not allow the user to change theses device numbers.

Instruction 600 is interpreted as a transfer to instruction 250. The computer is going to branch to instruction 250 and execute it. Then, it is going to continue normally with the instructions that follow instruction 250; i.e. 300, then 400, etc.

Instruction 600 could be changed by:

```
600 IF Q < 100 GOTO 250
700 ........
```

The program would then be:

```
 050 PROGRAM "DIVISION"
 100 DCL N,D,Q
 150 SEGMENT 0
 200 REM DIVISION
 250 ASK=1: "NUMERATOR"=N
 300 ASK=1: "DENOMINATOR"=D
 350 REM CALCULATE QUOTIENT
 400 LET Q = N/D
 500 PRINT=1: "QUOTIENT",Q
 600 IF Q < 100 GOTO 250
 700 STOP
1000 ESEG 0
1110 END
```

In this case, if the value of the variable Q, (that is, the result of the division) is less than 100, the program is going to continue from 250. Otherwise, (i.e., if Q is greater than or equal to 100), it will continue execution at instruction 700.

The IF-type command is called a conditional instruction. It also exists under another form:

```
600 IF Q < 100 THEN 250 ELSE 700
```

This is interpreted as: if (IF) the value of Q is less than 100, then (THEN) the program continues from instruction 250. However, if the value of Q is greater than or equal to 100, the program continues with instruction 700 (ELSE).

Modification of the previous program then gives:

```
050 PROGRAM "DIVISION"
060 FIELD=M
100 DCL N,D,Q
```

```
 150 SEGMENT 0
 200 REM DIVISION
 250 ASK=1: "NUMERATOR"=N
 300 ASK=1: "DENOMINATOR"=D
 350 REM CALCULATE QUOTIENT
 400 LET Q = N/D
 500 PRINT=1: TABV(2), "QUOTIENT N/D", Q
 600 IF Q < 100 THEN 200 ELSE 700
 700 STOP
1000 ESEG 0
1100 END
```

Note that instruction 500 has also been changed. TABV(2) instructs the display to tab down two lines prior to printing the quotient.

1.1.2 Example 2 - Using a Subroutine

Suppose that a customer, C, wants to buy something. The initial price is P. If the order is 10 or more units, he is given a 10% discount on the purchase price, M. In the case where the number of units ordered is 25 or greater, the discount is 20%.

To establish the price for client C, the salesman has to carry out a certain number of elementary operations -- one after another:

1. Ask for the name of the customer.
2. Ask for the quantity ordered -- Q.
3. Ask for the unit price -- P.
4. Calculate the price as
   M = Q * P
5. See if the quantity ordered is 25 or more. If yes, then calculate the reduction of 20% as
   R = M * 0.20
   and subtract the reduction from the price as
   M = M - R
   and then go to the next client.
6. Otherwise, see if the quantity ordered is 10 or more. If yes, then calculate the 10% reduction as
   R = M * 0.10
   and subtract the reduction from the price as
   M = M - R
   and go to the next client.

If we write a BAL program for this sequence of operations we have:

PROGRAM   STATEMENT EXPLAINED IN CHAPTER

```
100 PROGRAM "PRICE"
105 DCL C$=20,Q,P,R,M
106 SEGMENT 0
110 ASK=1: TABV(2), "CUSTOMER NAME",=C
120 ASK=1: TABV(2), "QUANTITY"=Q
130 ASK=1: "UNIT PRICE"=P
140 LET M = Q*P
150 IF Q < 25 GOTO 160
152 LET R = M * 0.20
154 LET M = M-R
156 PRINT=1: TABV(2), C,Q,P,M
158 GOTO 110
160 IF Q < 10 GOTO 170
162 LET R = M * 0.10
164 LET M = M-R
166 PRINT=1: TABV(2), C,Q,P,M
168 GOTO 110
170 PRINT=1: TABV(2), C,Q,P,M
180 GOTO 110
190 STOP
400 ESEG 0
410 END
```

Note that we simplify the program by writing the PRINT statement once and branching to it after statements 154 and 164. This is shown below.

```
................
140 LET M = Q * P
150 IF Q < 25 GOTO 160
152 LET R = M * 0.20
154 LET M = M - R
156 GOTO 170
160 IF Q < 10 GOTO 170
162 LET R= M * 0.10
164 LET M = M - R
170 PRINT=1: TABV(2), C,Q,P,M
180 GOTO 110
190 ................
```

Note that instructions 152, 154 and 162, 164 are identical if you put a variable, for example R1, to designate the reduction. Then the instruction sequence becomes

```
152 LET R = M * R1
162 LET R = M * R1
```

To avoid rewriting the identical set of instructions, BAL lets you create a sub-program or subroutine. Consider the example:

```
           100 PROGRAM "PRICE"
           105 DCL C$=20,Q,P,R,R1,M
           106 SEGMENT 0
           110 ASK=1: TABV(2), "CUSTOMER NAME",=C
           120 ASK=1: TABV(2), "QUANTITY"=Q
           130 ASK=1: "UNIT PRICE"=P
           140 LET M = Q*P
           150 IF Q < 25 GOTO 160
           152 LET R1 = 0.20
           154 GOSUB 300
           156 GOTO 170
           160 IF Q < 10 GOTO 170
           162 LET R1 = 0.10
           164 GOSUB 300
           170 PRINT=1: TABV(2), C,Q,P,M
           180 GOTO 110
           190 STOP
          (300 LET R = M * R1
Subroutine{310 LET M = M - R
          (320 RETURN
           400 ESEG 0
           410 END
```

Observe that the subroutine was composed of instructions that were identical to those in the preceding program. After writing these instructions, we put RETURN which indicates the end of the subroutine and orders the computer to return to the first executable instruction after the one which called the subroutine.

To call a subroutine, use the instructions GOSUB, followed by the number of the line which began the subroutine. Notice that the price reduction in our program example was determined before the subroutine was called.

CHAPTER 2. THE STRUCTURE OF A BAL PROGRAM

2.1 REQUIRED STRUCTURE

As illustrated in the examples in Chapter 1, every BAL program requires the following structure:

Elements Of The Program	Instructions
1. First instruction must be the program name.	PROGRAM "NAME"
2. Declarations -- a group of instructions naming and declaring all variables used in the program.	FIELD=..... DCL........ .... FIELD=..... DCL........
3. The body of the program, which consists of one or more segments, each of which begins with SEGMENT n instruction, ends with ESEG n.	SEGMENT 0 . . . ESEG 0
Each segment contains a body of instructions which perform various program functions. The variables are <u>common</u> to all segments.	SEGMENT 4 . . ESEG 4
Segment 0 is required, others are optional (up to 16 total) and can appear in any order. One segment can call another, like a subroutine, but one segment cannot refer to instructions inside another segment.	SEGMENT k . . . . ESEG K
4. The END statement.	END

2.2 PROGRAM NAME

The first statement of every BAL program must begin with the keyword PROGRAM, followed by the user-assigned program name of 1 to 15 characters, enclosed in parentheses.

Example: PROGRAM "SAMPLE"

2.3 DECLARATIONS

The instructions which name and declare the program variables follow the program name. All variables used in a program must be declared prior to the appearence of a SEGMENT n statement. The variable declaration instructions are described in detail in Chapter 3.

The declared variables are common to all segments -- and cannot be declared local to a single segment. Also appearing prior to the SEGMENT 0 statement, are the FIELD statements which define the types of memory that support the variables being declared. BAL allows three types of memory to be used: the central memory and the virtual memory of the minifloppy disks and the large hard disks.

2.4 BODY OF THE PROGRAM

The body of a BAL program consists of one or more segments, each of which includes a group of instructions. Every program must include a Segment 0; other segments are optional. See Chapter 10 for complete details on segmentation, including instructions which allow one segment to transfer control to another.

Instructions are constructed as detailed in following chapters. The maximum length of an instruction line is 255 characters. Blanks may be used freely within instructions to improve readability, except for the equal (=) sign which may not be preceded by a blank.

2.4.1 Line Numbers

Within a segment the user may assign line numbers to the BAL instructions. Line numbers are optional, but if they are used, must be in ascending order within the same program segment, and be in the range from 1 to 9999. Note that line numbers in one segment are independent of line numbers in any other segment.

2.4.2 Comments

The importance of comments in a program cannot be over-estimated. The extra time required to document a program as it is written is repaid whenever questions arise, or changes are contemplated.

BAL provides four ways to include comments within a program:

1. The REM Instruction - Allows the user to enter an entire line of comments. When REM is typed as the keyword of an instruction line, everything following (to the next carriage return) is considered to be a comment by BAL.

2. * Instruction - Has the same function as REM, but may be used to produce a neater listing, especially when blank lines are needed in the listing for clarity.

3. . Instruction - Functions the same as the asterisk, but also forces an advance to top-of-form operation, so that the comment will always appear at the top of a printed listing page. This is useful in placing page headings on your listings.

4. ; construct - The semicolon can be typed on a line to the right of a statement, then followed by a comment. Everything entered on a line after the semicolon is considered by BAL to be a comment.

EXAMPLES:

```
              PROGRAM "EXAMPLE"
              REM VARIABLE DECLARATIONS FOR INDEXES
              DCL A,B,C,D, D4#,D5#,D6#
              *
              *
              .  THIS COMMENT WILL APPEAR AS THE TOP
                 LINE OF A PAGE.
              DCL.................
              SEGMENT 0
        100   A = 2.0       ;Initialize A
                  .
                  .
```

CHAPTER 3. DESCRIPTION AND DECLARATION OF VARIABLES

3.1 INTRODUCTION

BAL, like all programming languages, uses an assortment of characters to communicate with the system.

To communicate the characters to the computer, one uses a type-writer-like keyboard. After processing, the computer directly transmits the information back to the CRT screen, the printer, or to the disk.

The character set used for BAL is:

- o 26 characters of the alphabet:
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- o 10 numeric characters:
  0 1 2 3 4 5 6 7 8 9
- o special characters which are subdivided into:
  - o symbols of operation
    - - subtraction
    - + addition
    - * multiplication
    - / division
  - o parentheses
    - (left parenthesis
    -) right parenthesis
  - o comparison symbols
    - = equal
    - < less than
    - > greater than
    - <= less than or equal
    - >= greater than or equal
    - <> not equal
  - o punctuation characters
    - . period
    - , comma
    - ; semi-colon
    - ? question mark
    - : colon

- other special characters
  - ' apostrophe
  - " quotation
  - $ dollar sign
  - % percent
  - @ at
  - & and
  - # number

The user should take care to distinguish:

- The number zero and the letter O; usually the zero will have a slash through it, but there is no strict rule about it.

- The number one and the letter I.

The blank character (or space) will be represented by a blank or by the symbol b̸.

3.2 THE VARIABLES

3.2.1 General

A variable is capable of taking on different values during the execution of a program. In the computer, the variable is assigned a memory location. The contents of the location can change as various operations use it.

In the case of a constant, it too is assigned a memory location but the contents remain unchanged throughout the lifetime of the program.

The instruction

```
            100 LET Y = X
```

places the numeric value located in the X location in the Y location as well. The value will still remain in the X location, it has merely been copied.

There was to be a name for each location; and in the program, you will refer to the contents by the variable name.

All variables must be declared at the beginning of the program. Detailed information on instructions for declaring variables is found in Chapter 4.

3.2.2 Variable Names

The name of the variable can be:

- o A single alphabetic character (A through Z, O excluded).
- o An alphabetic character (A through Z, O excluded) followed by a single digit number (Ø to 9).

This allows 25 x 10 or 250 variable names. Note that A and AØ, B and BØ, etc. are the same variable names; whereas A and A1, B and B1, etc. are different variables. The following variable names are valid:

```
B or BØ (actually the same variable name)
C4
D9
K
W3
```

The names of the following variables are invalid:

```
A01  - two digits only
B12  - two digits only
01   - alphabetic O not allowed
04   - variable name must begin with alphabetic
       character
1A   - name may not begin with a numeral
```

3.2.3 Types of Variables

BAL uses three kinds of variables:

- o Short numeric variable (type 1 or type 2)
- o Long numeric variable
- o Character string variables

3.2.3.1 Short Numeric Variables

These are integer variables of two types:

- o Type 1, whose range is $-128 < x < 127$
- o Type 2, whose range is $-32768 < x < 32767$

These variables allow for rapid calculations. The space occupied in memory is 1 character or byte for type 1 and two characters or bytes for type 2. To define a variable of type 1, the variable name must be appended with the character # in the DCL statement; for type 2, with the character %. Note that this suffix is used with the variable only when it is defined. (See 3.5 for an explanation of the DCL statement.)

```
EXAMPLE:  R3# - Range -128 to 127
          A1% - Range -32768 to 32767
```

These variables are initialized to zero before execution of the program.

3.2.3.2 Long Numeric Variables

These variables are floating point variables, allowing operations with numbers to 14 significant digits. Long variables do not have an appending identifier. They can be of varying length.

These variables are initialized to zero before the beginning of the program. Note, however, that you must re-initialize these variables if restarts of your program are allowed.

3.2.3.3 Character String Variables

These variables can represent strings of characters, ranging from 1 to 256 characters. The variable name, at the time of declaration, must be followed by a dollar sign ($) and the length, in number of characters.

```
EXAMPLE: 20 DCL A1$=125
         30 DCL B$=10
```

String variable A1 will be 125 characters in length; string variable B will be 10 characters.

If the length specification is omitted, the length is implicitly declared to be 15 characters.

Prior to program execution, all string character locations are initialized as the character "blank". However, you must re-initialize these variables (as necessary) if restarts of your program are allowed.

Note that the $ suffix is used <u>only</u> when the variable is defined.

3.3 CONSTANTS

Constants may be used in the program, expressed as follows:

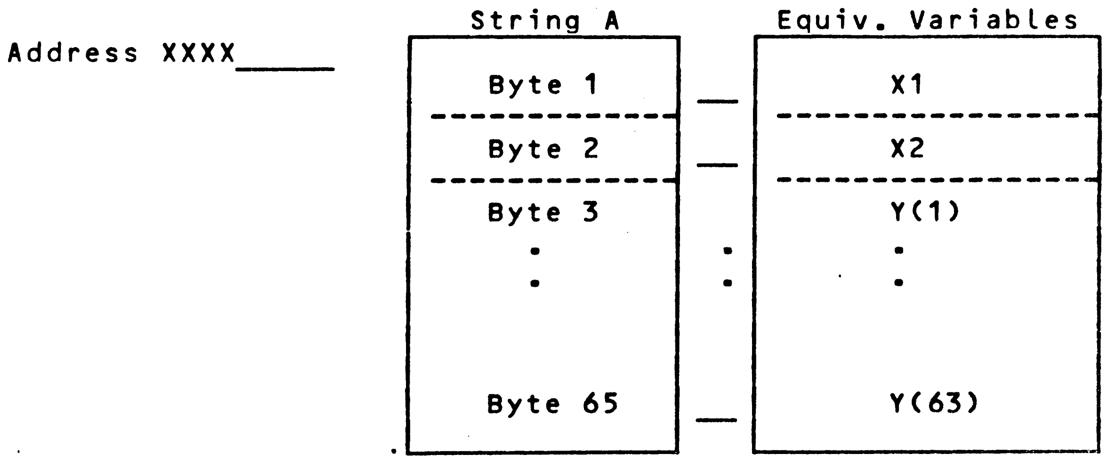
Decimal Constants	- Expressed as a decimal number, as 4 or 4.3.
Hexadecimal Constants	- Expressed as a short numeric, preceded by a slash (/), as /3F.
String Constants (string literal)	- Expressed in quotation marks, as "STRING".

3.4.1 Equivalencing Variables

As each variable is placed in memory, the system notes its address. It is therefore possible to equivalence variables - that is, have 2 or more named variables occupy the same physical memory location. The variables must be defined in the same type of memory.

Example:
```
10 FIELD=M
20 DCL A$=128
30 FIELD=M,A
40 DCL X1#,X2#,Y%(63)
```

The variable X1 occupies the same memory locations as the first byte of string A. Variable X2 occupies the same address as the next byte of A; and array Y occupies the same memory locations as bytes 3 through 65 of string A.



This allows for easy breakdown (or construction) of a string -- such as a disk record -- into its component parts.

3.5 DCL INSTRUCTION

This instruction declares the variables which will be stored in memory as specified by the last FIELD instruction which was encountered in the program. The variables are noted and their addresses are assigned in the sequential order of definition in the DCL instruction.

The general format of a DCL instruction is:

DCL V1, ..., Vn	where a series of variables can be declared by a single DCL instruction.

For each variable, the DCL instruction specifies:

- o The variable name, A through Z (O excluded), optionally followed by a single digit, 0 through 9. Example, A3.
- o The variable type, as:

numeric:	a short variable (type-1 or type-2) or a long (14 significant digits) floating point variable.
alphanumeric:	a character string

o The length specification (for long variables and string variables).

o The associated dimensions (vector or table), if necessary. The variable type is defined by an identifier which is:

# -	short variable (occupying 1 byte)
% -	short variable (occupying 2 bytes)
$ -	string variables (occupying a maximum of 256 bytes)

The identifier is used <u>only</u> when the variable is declared in the DCL instruction.

Example: `100 DCL A#, A1%(10), B$=12(128), T=7(12,8)`

Long variables operate differently from short integer variables and do not require identifiers. A variable cannot have more than one identifier--i.e., it cannot be declared several times under different identifiers.

Long variables and character strings have an implicit length. However, the user can specify a length by following the variable with an equal sign (=) and its length in bytes.

Examples:	T=4	Can be interpreted as a long variable four bytes in length.
	T=4(17,18)	Can be interpreted as a table of dimension 17 x 18; each element is defined as four bytes in length.
	T1$=10(15)	Can be interpreted as a vector of 15 values; each element is a string 10 bytes in length.

3.5.1 Remarks

o The maximum length for character strings is 256.

o The maximum length for long variables is 8 bytes - this corresponds to 14 decimal digits plus the decimal point and the sign.

- o The length specification (for long variables and string variables).
- o The associated dimensions (vector or table), if necessary. The variable type is defined by an identifier which is:

#	- short variable (occupying 1 byte)
%	- short variable (occupying 2 bytes)
$	- string variables (occupying a maximum of 256 bytes)

The identifier is used only when the variable is declared in the DCL instruction.

Example: `100 DCL A#, A1%(10), B$=12(128), T=7(12,8)`

Long variables operate differently from short integer variables and do not require identifiers. A variable cannot have more than one identifier--i.e., it cannot be declared several times under different identifiers.

Long variables and character strings have an implicit length. However, the user can specify a length by following the variable with an equal sign (=) and its length in bytes.

Examples:	T=4	Can be interpreted as a long variable four bytes in length.
	T=4(17,18)	Can be interpreted as a table of dimension 17 x 18; each element is defined as four bytes in length.
	T1$=10(15)	Can be interpreted as a vector of 15 values; each element is a string 10 bytes in length.

3.5.1 Remarks

- o The maximum length for character strings is 256.
- o The maximum length for long variables is 8 bytes - this corresponds to 14 decimal digits plus the decimal point and the sign.

If m is the length of the variable, the number of decimal digits -- that is, the precision of the information is: (m-1) * 2. If n is the desired precision--that is, the number of decimal digits, the length of the variable must be (n/2 + 1). Thus T=6 has a precision of (6-1) * 2 or 10 decimal places.

If one wants a precision of 5 decimal places, a length of 4 would have to be specified.

(5/2 + 1)__ (2.5 + 1)__ 3.5_ 4

Example:

```
  5 PROGRAM "DIVISION"
 10 FIELD=M
 15 DCL N,D,Q
 50 SEGMENT 0
200 ASK=1: "NUMERATOR" = N
300 ASK=1: "DENOMINATOR" = D
400 LET Q = N/D
500 PRINT=1: N,D,Q
600 GOTO 200
700 ESEG 0
800 END
```

The variable are assumed to be long variables with a maximum precision of 14 digits. All variables are located in central memory.

If we change instruction 15 to

```
15 DCL N=6, D=6, Q
```

the program will use the variables N and D with a precision of 10 decimal places; and a precision of 14 for the variable Q.

3.5.2 Sequence of Declaration Instructions

```
100 PROGRAM "Name"
... FIELD........
... DCL..........
... DCL..........
... .............
... FIELD........
... DCL..........
... .............
... FIELD........
... DCL..........
... .............
... .............
... SEGMENT 0....
```

For each FIELD, there must be as many DCL instructions as necessary to list the desired variables. The number of FIELDS is unlimited.

3.6 HANDLING ARRAYS AND MATRICES

3.6.1 Arrays

Occasionally, it is desirable to group information together when there is a common relationship among the data. One example would be the grades of a class on a particular exam. It would be possible to represent each grade with a separate variable, but the relationship among the data would be lost. BAL allows the information to be expressed as a group by declaring a variable name to be a group variable name, specifying how many items are in the group. The declaration would be as follows for a group of 50 items called J8.

```
200 DCL J8(50)
```

Throughout the life of the program, variable J8 would be the name of the group. To express the value of a single item within the group, subscripting is performed. Thus to refer to the fifth item, one would specify J8(5). Since it is not possible to write $J8_5$ to indicate a subscript in parentheses as a convention. The following example assigns $J8_5$ to another variable.

```
250 LET B = J8(5)
```

Such a group as J8 is commonly called a single dimension array.

3.6.2 Matrices

There are occasions when there are two relationships among a group of data. This can be expressed with a matrix. To declare such a group:

```
300 DCL J(50,10)
```

To refer to an individual item within the group--as with an assignment statement--requires two subscripts. The subscripts are used to refer to the row and column location of an individual data item within the structure. Hence, to specify the item located in the fourth row, ninth column of the group J would require a specification of J(4,9). A matrix declaration of K(50,10) means that there are 50 x 10 or 500 data items in the group.

Consider an example of a matrix. We have a high school containing four grades of students--9th, 10th, 11th, and 12th. There are boys and girls in each grade. It would be possible to define the school with a matrix as S(2,4) or S(4,2). That is, two types of students and four types of grades. To refer to the girls in the 9th grade would mean a specification of one of the eight matrix positions.

3.6.3 Declaring Arrays and Matrices

Arrays and matrices may be declared using any type variables:

A3#(5)	B#(5,6)	short variables, type-1
C%(10)	D%(7,5)	short variables, type-2
J1(15)	L(4,9)	long variables
K$(25)	M2$(8,8)	string variables

The length of each data item is declared by default. One byte for short variables, type-1; two bytes for type-2; eight bytes for long variables; and 15 bytes for string variables.

The user can specify the lengths of group variables of the long or string type by using the following convention:

```
R4=3(9)                      N=3(8,7)
W$=9(25)                     X9$=256(5,70)
```

The number before the parentheses is the length specification. The numbers within parentheses specify the list or matrix size. The minimum length for a long variable is 1 byte; the maximum is 8 bytes. For a string variable, the minimum is 1; the maximum is 256 bytes.

Note that, once a variable has been declared as to type, the symbol (#,%,$) is not used within the body of the program to specify the variable type. Thus you write:

```
    100 LET Y = K(15)

not 100 LET Y = K$(15)
```

to specify location 15 within the string list K.

3.6.4 Examples

The obvious advantage of arrays and matrices is the use of one variable name for many data items. But there is a second advantage. Since the elements may be selected using a variable as a subscript, the entire group can be operated on systematically. Consider a program having as a declaration:

```
100 DCL A(50),B,I#
```

The following code would initialize list A to 5's:

```
500 FOR I=1 TO 50
510 LET A(I)=5
520 NEXT I
```

Finding the average of all the elements of A is easily done:

```
600 LET B=0
610 FOR I=1 TO 50
620 LET B=B+A(I)
630 NEXT I
640 LET B=B/50
```

By extension, it is possible to perform other statistical operations as well as matrix operations. However, these techniques are equally applicable to business problems. Consider an application requiring the retention of an account number, an account name, and a balance.

The declaration might be as follows:

```
200 DCL A(50); account number
210 DCL A1$=25(50); account name
220 DCL A2(50); account balance
```

In this case, elements with the same subscript in each of the lists refer to the same account. If the balance is to be zeroed at the beginning of the year, this is easily done (see lines 500-520 above, substituting A2 for A).

To search for a matching account number, assuming B contains the account sought:

```
800 FOR I=1 TO 50
810 IF A(1)=B GOTO 850
820 NEXT I
830 PRINT=1:TABV(1),"NO SUCH ACCOUNT"
840 STOP
850 PRINT=1:TABV(1),"ACCOUNT",B,"IS",A1(I)
```

Note that, once a variable has been declared as to type, the symbol (#,%,$) is not used within the body of the program to specify the variable type. Thus you write:

```
          100 LET Y = K(15)

      not 100 LET Y = K$(15)
```

to specify location 15 within the string list K.

3.6.4 Examples

The obvious advantage of arrays and matrices is the use of one variable name for many data items. But there is a second advantage. Since the elements may be selected using a variable as a subscript, the entire group can be operated on systematically. Consider a program having as a declaration:

```
100 DCL A(50),B,I#
```

The following code would initialize list A to 5's:

```
500 FOR I=1 TO 50
510 LET A(I)=5
520 NEXT I
```

Finding the average of all the elements of A is easily done:

```
600 LET B=0
610 FOR I=1 TO 50
620 LET B=B+A(I)
630 NEXT I
640 LET B=B/50
```

By extension, it is possible to perform other statistical operations as well as matrix operations. However, these techniques are equally applicable to business problems. Consider an application requiring the retention of an account number, an account name, and a balance.

The declaration might be as follows:

```
200 DCL A(50); account number
210 DCL A1$=25(50); account name
220 DCL A2(50); account balance
```

In this case, elements with the same subscript in each of the lists refer to the same account. If the balance is to be zeroed at the beginning of the year, this is easily done (see lines 500-520 above, substituting A2 for A).

To search for a matching account number, assuming B contains the account sought:

```
800 FOR I=1 TO 50
810 IF A(1)=B GOTO 850
820 NEXT I
830 PRINT=1:TABV(1),"NO SUCH ACCOUNT"
840 STOP
850 PRINT=1:TABV(1),"ACCOUNT",B,"IS",A1(I)
```

CHAPTER 4. ASSIGNMENT INSTRUCTION

4.1 INTRODUCTION

The object of assignment instructions is to assign a value (number or character) to a variable (number or character). This is the most frequent method of calculation used in BAL.

4.2 SYNTAX

The general form of the assignment instruction is:

```
[LET]V = z
```

LET - Optional keyboard

V - a numeric or character string variable that could be simple or indicative.

z - a numeric or character expression

The effect of this instruction on the program is that the expression z will be calculated if necessary and its value will be assigned to the variable V.

Example:

```
*LET X=4
     A="GIRL"
     X1=-A/B
     C=A2
     D(I,J)=A(M,N)+B
```

Note the following:

1. No more than one variable is allowed to the left of the equal sign of the instruction.

   ```
   A=-3 or A2=-3*A is valid
   A+2=-3 is not valid
   ```

   The effect of the instruction is to assign the value (numeric or character) of the expression on the right to the variable on the left.

2. A frequently used application of this instruction is reassignment.

* The keyword LET is optional.

X=X+1 means:

Take the numeric value of X (10, for example); add one to it (making it 11); assign the result to X. The memory location assigned to X is thus increased by 1.

3. The expression z may be a constant. The assignment instruction is often used to assign an initial value to a variable. Constants may be decimal or hexadecimal.

4. Variables should be initialized before they are used.

4.3 NUMERIC EXPRESSIONS

A complete numeric expression is made up of one logical or arithmetic operator and two variables. It could be represented as:

```
                     constant              constant
                        or                    or
Expression=sign.variable 1.operator.variable 2

Example:     -          3          *         A2
```

4.4 LOGICAL EXPRESSIONS

BAL uses the following logical operators:

AND	-- logical AND
OR	-- logical OR
OX	-- exclusive OR

The logical operators are used in numeric expressions just as the arithmetic operators. However, the variables in logical expressions must be of the short type.

4.5 ARITHMETIC EXPRESSIONS

BAL uses the following arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division

For example, the expression A * B means multiply A times B; while A + B(1) means add the array value B(I) to A. If I were equal to 3, A equal to 100 and the array positions of B were respectively: B(1)=10, B(2)=45, B(3)=30 AND B(4)=18, then the above expression A + B (I) would be equal to 130.

If the first variable of an arithmetic expression is positive, the user can place a plus sign before it. The plus sign is optional. If the first value is to be expressed as negative, the negative sign is mandatory.

The following numeric expressions are correct:

```
A - B1
-A - B(I,J)
-10.3 * B(I,J)
C1/12.4
-14/13
B
```

The following arithmetic expressions are incorrect:

```
A * B * C        -- only one operator allowed
-10.5 ++ B(125)  -- too many operators
A * (B/C)        -- only one operator allowed per
                    line; also parentheses are not
                    allowed in an arithmetic ex-
                    pression.
```

The usual form of a numeric expression is in the LET statement. For example:

```
[LET] A = -B + C
```

which will calculate a value and assign it to variable A.

NOTE: In arithmetic expressions, the variables must be either all long or all short. Mixing long and short modes is not allowed.

4.6 FUNCTION EVALUATION

The assignment instruction is used in the evaluation of mathematical and string functions. The general form is:

```
[LET]V = F(v1)
```

Where V is a destination variable of the proper type, as required by the function; and F(v1) is one of the mathematical, string, or miscellaneous functions described in Chapter 12.

Example:

```
N=MOD(N1,N2)
B=DATE(4)
```

CHAPTER 5. PROGRAM FLOW AND CONTROL INSTRUCTIONS

5.1 GENERAL

In principle, the execution of program instructions is sequential. However, there are certain instructions which can control program execution, permitting the testing of some condition and branching by the program to any desired instruction. These instructions are:

```
GOTO
OF...GOTO
ON...GOTO
IF...GOTO
IF...THEN...ELSE
FOR...NEXT
GOSUB
OF...GOSUB
```

5.2 GOTO INSTRUCTION

Syntax:

```
GOTO n
```

When this instruction is executed, the program will unconditionally branch to and execute the instruction at line n. Line n could be anywhere in the segment--before or after the instruction GOTO n.

5.3 OF...GOTO INSTRUCTION

Syntax:

```
(OF v GOTO n1[, n2, ... ,np]
```

where the n_x is a line number of the instruction to be branched to. The value v is the index variable. The value of the variable v at the time the program executes the multiple GOTO instruction will determine which n_x is the line number of the next instruction. Variable v may have a maximum value of 124; i.e., there may be up to 124 branch points listed in a multiple GOTO instruction.

If v=1, branch to instruction n1
If v=2, branch to instruction n2
.
.
If v=p, branch to instruction n_p

If the value of v is less than or equal to zero or greater than p, the OF instruction will be ignored and the computer will execute the next sequential instruction. If v is not an integer, it will be truncated and only the integer portion will be used.

Example:

```
          200 OF L GOTO 230, 250, 270
          210 GOTO 300
          220 REM 10% REDUCTION
          230 LET T = T * 0.85
          260 GOTO 300
          265 REM 17% REDUCTION
          270 LET T = T * 0.83
          300 ......
```

Depending on the value of L (assigned earlier in the program), one systematically executes 0%, 10%, 15%, or 17% reduction on the total T. Instruction 210 is for the case where L is out of range (less than 1 or greater then 3); i.e., there is no applicable reduction. If L=1, there is a branch to instruction 250; for L=3, a branch to instruction 270.

Note that the program of this example could be written more efficiently using an array and a FOR...NEXT loop.

5.4 ON...GOTO INSTRUCTION

Syntax:

ON v GOTO n_1,n_2,n_3

where n_1, n_2, n_3 designate line numbers of instructions. v is a numeric index variable.

The execution of this instruction proceeds as follows:

- o If v has a negative value, the instruction a line n_1 is executed.
- O If v is positive, the instruction at n_3 is executed.

For example, in the program at the beginning of the chapter, we can replace instructions 130 and 140 by one instruction:

```
     120 ASK=1:  "VALUE" = V
     130 ON V GOTO 150, 200, 170
     150 LET N = N + V
```

5.5 IF...GOTO INSTRUCTION

The If instruction permits a conditional branch. The syntax is:

IF v_1 <u>operator</u> v_2 GOTO n

where v_1 and v_2 are variables (and/or constants) of the same type, n is a line number to be executed if the condition is true, and the operator is a relational operator. We say that v_1 <u>operator</u> V_2 expresses a condition. If that condition is true, there is a branch to instruction, if the condition is not true, the next executable instruction is sequence is executed.

Example: IF Z = 5.2 GOTO 151

If the value of Z is equal to 5.2, the instruction at line 151 is executed, If the value of Z is not equal to 5.2, the next instruction in sequence is executed.

5.6 IF...THEN...ELSE INSTRUCTION

The compound IF instruction permits alternative branch points.

IF v_1 <u>operator</u> v_2 THEN n_1 ELSE n_2

where v_1 and v_2 are variables (and/or constants) of the same type, n_1 and n_2 are line numbers, and the operator is a relational operator. If the condition expressed by "v_1 <u>operator</u> v_2" is true, there is a branch to instruction n_1; if not, there is a branch to instruction n_2.

Example: IF N < 0 THEN 170 ELSE 150

Note the following conditions for the use of the compound IF:

1. The variables v_1 and v_2 must be of the same type. They could be:

   short variables (1 or 2 bytes)
   long variables
   character string variables

2. The relational operators used BAL are:

<	less than
=	equal to (or included in - alphanumeric)
>	greater than
>=	greater than or equal to
<=	less than or equal to
<>	different from or not equal to

The signs are found on the keyboard of the CRT terminal. In the case of <=, >=, or <>, you must type two characters.

3. When v_1 and v_2 are numbers, the computer executes a simple comparison of the values v_1 and v_2 at the moment of execution.

4. Comparisons of strings for equality -- When two character strings are compared for equality, the first string is considered equal to the second if it is contained within that second string. A single space (blank) within a character string is considered significant, but multiple spaces are considered the same as a single space. Leading spaces are ignored, but trailing spaces are significant.

   Examples:

   a. String C1 = "ƀƀYESƀƀƀ -- String to be compared
      String C2 = "YESƀƀOUIƀƀY" -- Reference string

      If C1 = C2 THEN n_1 ELSE n_2 -- For rhz above strings, this condition is true, so instruction n_1 is executed.

   b. For string C1 = "YES", the relation is again true, C1 is contained in C2.

   c. For string C1 = "ƀƀƀƀƀƀYES", the relation is true. The leading blanks are ignored.

   d. For C1 = "Yƀƀ, the relation is false (trailing blanks are significant and are not found in C2), so the branch to n_2 is taken.

   e. For C1 = "YESOUI" the relation is false. Note that C2 includes a blank between YES and OUI. If C1 were "YESƀOUI" or even "YESƀƀƀOUI", the relation would be true.

5. String comparison for other than equality -- In comparing two character strings, they do not necessarily have to be of the same length. The computer compares the strings from left to right, character by character. Comparison stops as soon as a decision is possible or at the end of the shortest string.

   The relative value of the characters follows the ASCII character set:

ƀ	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[	/	]	^	-

For example, compare:

C1 = "bFORDbbb" versus C2 = "FORDbb"

In comparing the first character of C1, which is a blank, b, and the first character of C2, which is F, note that the blank is less than F (blank precedes F in the character set).

Then the following relations are as shown:

```
IF C1 < C2 ...,   or
IF C1 <=C2 ...    the result is true; for
IF C1 > C2 ...    the result is false.
```

If C2 were "bFORDb", then:

```
IF C1 < C2 ... the result is false
IF C1 <=C2 ... the result is true
IF C1 <>C2 ... the result is false
```

Note that we'd get the same results as above for C2 = "bFO".

Note that the line numbers specified in the IF instruction must be for some instructions within the same segment and cannot be for the IF instruction itself.

5.7 FOR...NEXT INSTRUCTIONS

The FOR...NEXT instructions are used to construct a program loop to perform a repetitive series of similar calculations.

The general form of these instructions is :

```
FOR v = v1 TO v2 [STEP v3]
.
.
NEXT v
```

The key words are FOR, to, STEP, and NEXT

The value v is a numeric variable, the index variable for the loop. The variable must be the same in both instruction FOR and NEXT. THis name identity assures the correspondence between FOR and NEXT of a given loop. This is expecially important when loop are placed within loops. Note that v cannot be a constant. However, v_1, v_2, and v_3 can be numeric variables or constants. If "STEP v_3" is omitted, the value is assumed for the increment.

5.7.1 Simple Loops

There are two types of instructions in loops:

- o The control instructions that define the loops: FOR and NEXT
- o The calculation instructions that are placed between the instructions FOR and NEXT.

Calculation instructions are those that are executed each time the program executes the loop. The control instructions (FOR and NEXT) satisfy the functions of:

- o Initializing the index variable,
- o Incrementing the index variable, and
- o Testing for the end of the loop.

5.7.2 Effect of the Instruction FOR/NEXT

The instructions would be executed in a repetious manner with the following values of v:

$v=v_1$ at the first execution

$v=v_1+v_3$ at the second execution, i.e., step v_3;

$v=(v_1+v_3)+v_3$ at the third execution and so on until v takes a value of v_2.

Then there is a normal exit of the loop and the program continues with the next executable instruction following the NEXT statement for the loop.

5.7.3 Step of the Loop (STEP)

The step of the loop v_3 can be a positive or negative, allowing v (the index) to take increasing or decreasing values. If, by programming error, the step were zero, the program would go into an "infinite loop". The loop would repeat itself indefinitely and would need outside intervention to stop the program. If, by programming error, v_3 has a value contradictory with those of v_1 and v_2, the loop is rejected. For example:

```
FOR L = 1 TO 5 STEP -2
FOR L = 1 TO -20 STEP 1
```

5.7.4 Index Value v

When a loop is executed, the index v possesses a value during each execution of the loop. This value is available as a usable variable for calculations, either as an ordinary variable or as a subscript for an array.

When the program exits the loop (normally or abnormally), the index can still be used in calculations. The value that it has would be the one at the time of the last pass through the loop. Note that if a loop is terminated normally, the final value of the index variable is always greater than the value v_2.

5.7.5 Examples

Program loops simplify many operations requiring subscripts or indices. For example, to set to zero all the elements of a list of L elements, one could write:

```
200 FOR L = 1 TO L STEP 1
210 LET A (J) = 0
220 NEXT J
```

To calculate the sum of the elements of L, one could write:

```
200 LET S = 0
210 FOR J = 1 TO L STEP 1
220 LET S = S + A(J)
230 NEXT J
240 LET M = S/L
```

Instruction 240 gives the average of the values of L. Also, loops often are useful for non-indexed variables. In this case, the role of the loop is to compute the number of times the calculations is executed.

5.7.6 Abnormal Conditions

Normal conditions for using a loop were defined above. However, one can vary these conditions, varying what can or cannot be accepted by the system. Abnormal conditions are:

- o Abnormal exit provoked by a condition included in the instructions calculating the loop.
- o Re-entering the loop without executing the instruction FOR: this is prohibited by the system and can cause unforseen errors of calculation.

The variables v, v_1, v_2, v_3 must be of the same numeric type. That is, either short (type 1 or 2) or long numeric variables.

5.7.7 Successive Loops

These are separate loops placed one before the other:

```
          FOR I = ...
           .
          NEXT I
          FOR I = ...
           .
          NEXT I
```

In this case, the first loop will be terminated (normally or abnormally) before the second is started. It is possible here to adopt the same name for the index of the two loops, because there is re-initialization at the beginning of each loop.

5.7.8 Nested Loops

Loops can be nested to two or more levels. THe BAL system limits the number of nested loops to 15. Two loops are said to be nested if one is executed entirely within the range of the other.

Example:

```
┌→FOR I = ........
│     ┌→FOR J = ...
│     │ ..........
│     │ ..........
│     └─Next J
│ ...............
└─NEXT I
```

For each single step of the index variable of the exterior loop, the interior loop will be executed (completely in the case of a normal exit; incompletely in the case of an abnormal exit). Observe that the indexes of the nested loops have diffenent names;

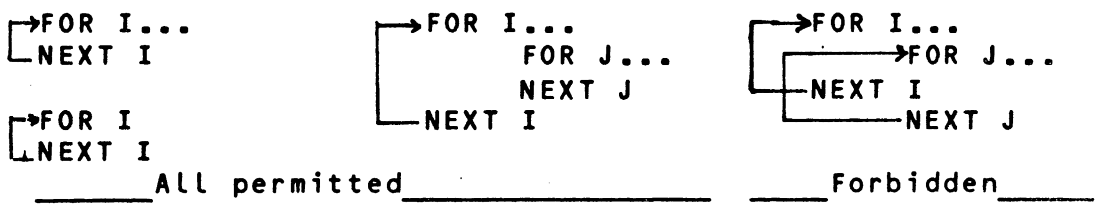
We would use nested loops in manipulating a 2-dimensional table.

Example:

```
100 LET S = 0
110 FOR I = 1 TO N1 STEP 1
120     FOR J = 1 TO N2 STEP 1
130     LET S = S + A(I,J)
140     NEXT J
150 NEXT I
```

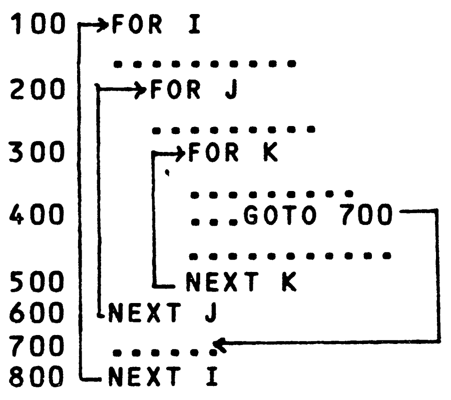
This calculates the sum of the values of a table with two dimensions.

The following diagrams summarize the possible cases of loops that are permitted or forbidden. The case of overlapping loops is logically forbidden because that demands that the computer execute the NEXT 1, continuing the J loop and simultaneously re-starting the I loop.



```
FOR I...                FOR I...                 FOR I...
NEXT I                      FOR J...                     FOR J...
                            NEXT J                 NEXT I
FOR I                   NEXT I                           NEXT J
NEXT I
        All permitted                               Forbidden
```

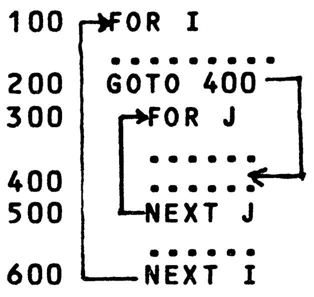
Nesting with an abnormal exit of the loop by testing is permitted wuth the condition that the return is placed outside the exited loop, for example:



```
100  FOR I
     ..........
200    FOR J
       .........
300      FOR K
         .........
400      ...GOTO 700
         ..........
500      NEXT K
600    NEXT J
700    .....
800  NEXT I
```

Permitted

As opposed to:



```
100  FOR I
     .........
200  GOTO 400
300   FOR J
      ......
400   ......
500   NEXT J
      ......
600   NEXT I
```

Forbidden

which is forbidden. The error is a run-time error and would not be detected as an error during compilation. Unpredictable results would occur during execution.

5.8 SUBROUTINES

The concept of sub-programs or subroutines is an important one in programming. It allows you to write a section of a program once, and then to invoke it when it is needed, rather than repeating the coding each time it is used. Programs which utilize subroutines tend to be easier to read and have the added advantage of requiring less memory space.

5.8.1 General Structure

A subroutine is a collection of instructions within a BAL program which generally will be used several times, invoked from different areas in the main program. Consequently, these instructions may refer to the same variables and line numbers as the main program (the calling program). Note that there are no local variables within subroutines in BAL. Thus, when using arrays and indexes with subroutines, you must be careful so that no confusion occurs with indexes and arrays used in the calling program.

The difference between a main program and a subroutine is the way in which execution of the BAL instructions is started. There are two instructions which govern the execution of sub-routines: GOSUB and RETURN. The GOSUB instruction is coded in a calling program when it is desired to pass control to a subroutine. The RETURN instruction is coded in a subroutine when control is to be passed back to the program which called the subroutine.

5.8.2 GOSUB Instruction

Syntax :

```
GOSUB n
```

where n is the first line number of the subroutine. This instruction calls subroutine n. When the call occurs, the system performs the following actions:

- o It saves the line number of the GOSUB instruction itself. This is necessary so that the subroutine can return control to the next instruction in the calling program.
- o It passes control to the subroutine beginning at line n (within the same segment as the calling until a RETURN instruction is encountered.

Example:

```
                    100 A = X
                    110 B = Y
                    120 GOSUB 500
                        ..........
                        ..........
                    200 A = 4.1
                    210 B = A + 0.7
                    220 GOSUB 500
                        .........
                        .........
                    500 C = A + B
                    510 D = A * B
                    520 PRINT=1: "THE ANSWERS ARE",C,D
                    530 RETURN
                    ..........
                    ..........
```

In this example, the variables A and B are set by the calling program, and then the subroutine is called. Lines 500 through 530 are executed once due to the GOSUB instruction on line 120, and a second time due to the GOSUB instruction on line 220. Notice that the subroutine may refer to the same variables and line numbers as the main program.

5.8.3 (OF...GOSUB Instruction

Syntax:

OF v GOSUB n_1, n_2, ..., n_p

v is variable of the short or long type and n_1, n_2, ..., n_p are line numbers referencing the beginning lines of the various subroutines.

When v = 1, the program will branch to the subroutine whose first line begins on line n_1. When v = 2, the program will branch to line n_2; and so on. The maximum number of subroutine references that may occur in an OF ... GOSUB instruction is 124.

If the value of v is less than 1 or greater than p, the system will ignore the subroutine request and transfer to the next executable instruction following the OF...GOSUB instruction. No error message will be generated.

The value of v should be an integer. If v is a floating point variable instead, the system will truncate the value to an integer and use it as the index. For example:

```
... .........
... LET J = 2.9
... .........
... .........
... OF J GOSUB 910, 920, 930, 940, 950
... .........
... .........
```

will generate a value of 2 for J and the branch to 920 will occur.

The characteristics of the OF ... GOSUB instructions thus are similar to the OF ... GOTO.

5.8.4 RETURN Instruction

within the section of code making up the subroutine, there must exist one or more RETURN instruction. The Syntax of the instruction is:

```
RETURN
```

When the RETURN is encountered, the subroutine initiated by the GOSUB becomes inactive. Hence, the instruction sequence of the subroutine halts. The next instruction to be executed by the program is the first executable instruction following the GOSUB request. Thus, there is a branch from the subroutine back to the calling program.

5.8.5 Remarks

1. The subroutines must be in the same segment as the instructions which call them.

2. There could be several RETURN instructions in the same subroutine that correspond to different points of exit. For example, the following subroutine will assign values to I, depending upon the values of X found:

```
... .....................
410 IF X >= 1000 GOTO 420
415 I = 1
416 RETURN
420 IF X >= 5000 GOTO 430
425 I = 2
426 RETURN
430 IF X >= 10000 GOTO 440
435 I = 3
536 RETURN
440 I = 4
446 RETURN
... .....................
```

Subroutine results:

X	I
X<1000	1
1000<= X<5000	2
5000<= X<100000	3
X >= 10000	4

The subroutine contains four RETURN instructions, but naturally in the execution of the subroutine, only one is executed depending on the value of X

3. Notice that the RETURN instruction does not always return to the exact same line of the calling program. It depends upon which GOSUB instruction called the subroutine.

4. If a RETURN instruction is encountered during program execution when a subroutine has not been called by a GOSUB instruction, an error message will be generated.

5.8.6 Nested Subroutines

A subroutine can call another which can call a third subroutine, etc. In each case the program will correctly save the return points. BAL allows a maximum of 16 levels of subroutine nesting.

Important Note

Successive or nested subroutines must be in the same SEGMENT as the GOSUB instruction which calls them.

CHAPTER 6. CONVERSATIONAL DATA ENTRY

6.1 GENERAL

Entering data according to a controlled format is an important aspect of all interactive computer systems. BAL uses two instructions for this purpose: ASK and MASK.

The ASK instruction allows you to display items on the CRT screen and to enter a new value for a single variable. The MASK instruction allows you to specify alternative actions to be taken when invalid data entry is attempted.

6.2 ASK INSTRUCTION

The ASK instruction is very powerful. Like any BASIC input instruction, it allows you to input a value for a variable. However, it also gives you the option to: 1) specify branch conditions if invalid data is entered, 2) specify the format of and display the contents of one or more variables, 3) control the CRT display format, positioning the data anywhere on the screen, and 4) specify the format for the input variable.

Paragraph 6.2.1 describes the simplest form of the ASK instruction, but this form is seldom used. Paragraph 6.2.2 describes the more general form of the instruction.

6.2.1 Simple ASK instruction

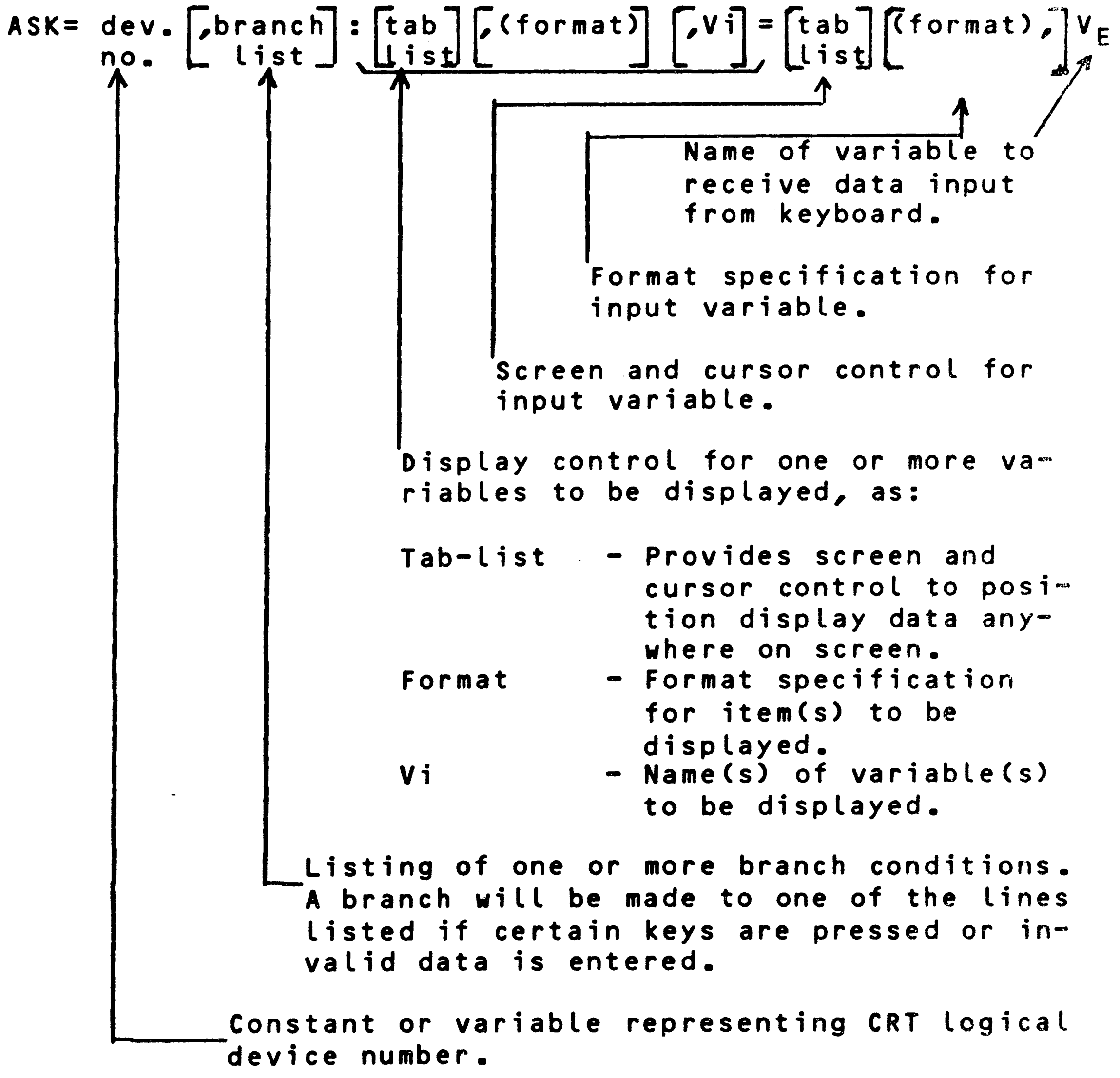
The syntax of the simplest form of the ASK instruction is:

```
ASK=1: = V
|   || | |
|   || | |_____ Name of the variable to receive data input
|   || |        from the keyboard.
|   || |
|   || |_______ Required equal sign.
|   ||
|   ||_________ Required colon.
|   |
|   |__________ CRT logical device number. This can be a
|               constant or a variable with a value of 1.
|
|______________ Keyword. Note that the keyword must be writ-
                ten ASK= , with no blanks between ASK and the
                equal sign.
```

When this instruction is executed, the computer waits for input from the keyboard. When data is entered, it will be displayed on the CRT display starting from the current position of the cursor, and will be entered into variable V in memory. The type of data which can be entered is controlled by the declaration of the variable. (Note that data entry and error conditions can be affected by the MASK instruction. See 6.3.) If an error is made while typing, pressing the @ key deletes the data already typed and starts the data entry process over.

6.2.2 ASK Instruction - General Format

The general format for the ASK instruction is:



```
ASK= dev. [,branch] : [tab] [,(format)] [,Vi] = [tab] [(format),] V_E
     no.  [ list  ]   [list]                      [list]
```

Name of variable to receive data input from keyboard.

Format specification for input variable.

Screen and cursor control for input variable.

Display control for one or more variables to be displayed, as:

Tab-list	- Provides screen and cursor control to position display data anywhere on screen.
Format	- Format specification for item(s) to be displayed.
Vi	- Name(s) of variable(s) to be displayed.

Listing of one or more branch conditions. A branch will be made to one of the lines listed if certain keys are pressed or invalid data is entered.

Constant or variable representing CRT logical device number.

Brackets surround optional fields and the brackets themselves are not to be entered in the BAL program.

Device Number

The device number may be an integer constant or a short integer variable with a value of one, specifying the CRT display screen.

Branch List

The elements of the branch list are:

```
[,E=line, I=line, U=line, D=line, /XX=line]
```

The line numbers represent the line numbers of BAL statements, one of which will be given control when invalid data is entered or when certain keys are pressed. Several branch conditions are available, provided they are enabled by the appropriate MASK instruction. Any, all, or none of these can be specified in an ASK statement, in any order. The conditions are:

E (Error)	-- Branch to the specified line if an error is made, such as incorrect format, non-numeric input where only numeric required, etc. (Requires MASK 16.)

The branch conditions below are valid only if you press the appropriate key <u>prior</u> to entering any data or after pressing annulation combination CTRL-/C.

I (Interrupt)	-- Branch to specified line if ESCAPE key is pressed. (MASK 128 may not be set.)
U (Up)	-- Branch to specified line if Up Arrow is pressed.
D (Down)	-- Branch to specified line if Line Feed (equivalent to Down Arrow) is pressed.
/XX	-- Branch to specified line if the key which generates ASCII code/XX is pressed. The code can be expressed in hexadecimal or decimal. (Note that the sign bit is unused so /41 is equivalent to /C1.) For example, if /41 is specified and the letter A is pressed, the branch is taken. This provides literally dozens of branching possibilities, limited only by the maximum length of an ASK instruction, 256 characters. See Appendix A for a list of standard ASCII codes.

Note 1: If you have typed input data, then decide to press a branch key, the branch will not occur. However, entering the annulation code erases the input field and restarts the data entry process. The branch keys are then valid.

Note 2: Annulation Code is normally parametrized to CTR-L (pressing C whole holding down CTRL key).

Tab List

This list of optional items describes the information which can be displayed on the CRT screen before you are able to enter a new value for a variable. Any or all of these items can be used in an ASK instruction, in any order.

HOME	Moves the cursor to the home position (upper left corner of screen -- first character of first line).
CLEAR	Clears the CRT screen and moves the cursor to the home position.
PAGE	Same as clear.
TABV(n)	Moves the cursor to the beginning of the line and down n lines, where n is a non-negative short integer variable or constant. TABV(0) returns the cursor to the beginning of the current line. The TABV function is the equivalent of a line feed.
TAB(n)	Moves the cursor horizontally and positions the cursor at column n. Note that this is an absolute positioning, beginning from column 1, no matter where the cursor is in the line at the moment.
BELL	Causes a beeping tone.

Note that these tab items are sent to the CRT in the order in which they appear in the ASK instruction.

Format

The format item preceding a variable controls the display or entry of that variable. See paragraph 9.4 for a complete description of formats. If no format is specified, the declaration of the variable (DCL) controls the characters which may be entered.

Variables Vi and V_E

- Variable Vi represents one or more variables to be displayed prior to data entry. This variable can be an explicit character string. All variables in the list will be displayed using the last format preceding them in the list. The occurence of one format item overrides any prior format item in the list. If no format item is given, the variable is displayed free-format.

- Variable V_E is the variable which accepts the data to be entered from the keyboard. Data will be entered using the format specified for input. If no format is specified, data entry is governed by the declaration of the variable in the DCL instruction.

6.2.3 Examples of ASK Instruction

1.
```
100 ASK=1: "ENTER A VALUE" = A
```

On execution of this instruction, the computer displays ENTER A VALUE, starting from the current position of the cursor. It then waits for data input from the keyboard. The data you type is entered into variable A in memory and displayed on the screen, immediately after VALUE, with no blanks, as: ENTER A VALUE10

2.
```
100 ASK=1:  "ENTER A VALUE" = TAB(30), A
```

This is similar to the above example. The only difference is that, after ENTER A VALUE is displayed, the cursor is tabbed to column 30 of the current line (counting from the left). The data you type then appears on the screen starting at column 30.

```
ENTER A VALUE                      10
↑                                  ↑
└─Col 1                            └─ Col 30
```

3.
```
100 ASK=1: TAB(5), "ENTER A VALUE" = TAB(30),A
```

Note that the only difference between this instruction and that of example 2 is that the system tabs to column 5 before displaying ENTER A VALUE.

4.
```
 99 T = "ENTER A VALUE"
100 ASK=1: TAB(5), T = TAB(30), A
```

These two instructions perform the identical function of example 3. Note that a character string can be specified as a variable, as well as explicitly specified.

5.
```
 99 T = "ENTER A VALUE"
100 ASK=1: CLEAR, TABV(3), TAB(5), T = TAB(30), A
```

This example is similar to example 4. The differences are that the screen is first cleared, then the cursor is tabbed down 3 lines (vertical tab). The remainder of the example is identical to example 4.

```
.....
.....
  ENTER A VALUE              10
  ↑                          ↑
  └─Col 5                    └─Col 30
```

6.
```
 98 T = "ENTER A VALUE"
 99 T1 = "TOTAL"
100 ASK=1: CLEAR, TABV(3), TAB(5), T = TAB(30), A
101 ASK=1: TABV(0), TAB(35), T1 = TAB(45), A1
```

The execution of instruction 100 is identical with that of example 5. After you enter the data for A, instruction 101 is executed. TABV(0) returns the cursor to the beginning of the current line (no line advance). The cursor is then tabbed to column 35 and TOTAL is displayed. The cursor is then tabbed to column 45 for the input of data for A1.

```
.....
.....
  ENTER A VALUE          10    TOTAL         12.7
  ↑                       ↑    ↑             ↑
  └─Col 5          Col 30─┘    └─Col 35      └─Col 45
```

7.
```
 98 I=5
 99 J=30
100 ASK=1, E=1200, D=150: CLEAR, TAB(I), T = TAB(J), A
```

When these instructions are executed, the following occurs:

a. Variable 1 is set to 5; variable J is set to 30.

b. The CRT screen is cleared and the cursor is moved to the home position.

c. The cursor is tabbed to column 5 (TAB(1)).

d. The variable, T, is displayed.

e. The cursor is tabbed to column 30 (TAB(J)).

f. At this point, you can press the LINE FEED key (Down Arrow) to cause a branch to line 150. The remainder of the ASK instruction will be ignored. This type of branch would be useful if the instruction were allowing you to modify the value of A, and you chose to continue without changing the current value of A.

g. If LINE FEED was not pressed, the computer waits for data entry. If an error is made (such as entering an alphabetic character where a numeric is expected) the system takes the E=line branch to instruction 1200.

   Note that the other branch conditions could also have been used in this example.

These examples illustrate the great flexibility of the ASK instruction. You can use this instruction to display a variable, request new data for that same or another variable, and specify alternate actions in case of an error or the entry of a cursor control function. Examples of the ASK instruction using format specifications are found in paragraph 6.4.

6.3 MASK INSTRUCTION

The MASK instruction is used to control the entry of information using the ASK instruction and to determine the action to be taken if an error occurs. The syntax is:

```
MASK n
```

where n is a short integer variable or constant. The value of n specifies which combination of 8 options is to be established for error handling. Any one or all options can be selected simultaneously. The options are:

1	Delete all invalid characters typed.
2	Cause the terminal to sound a beeping tone once for each invalid character.
4	Erase the entire field when any single invalid character is entered and require all data to be re-entered.

8	Do not require the user to enter a carriage return if the variable has been completly filled.
16	Enable the E=line transfer (as specified in the ASK instruction) when an invalid character is entered. Any existing value in the variable is not changed.
32	Do not modify the value of the variable v if the first character entered is a carriage return.
64	Do not display the character(s) entered. This permits the entry of invisible system passwords, etc.
128	Inhibit the I= branch, which is caused by pressing ESCAPE.

Options may be combined by using the sum of individual options desired. Thus, MASK 21 is equivalent to masks of 16, 4 and 1.

Once a MASK instruction has been executed, those options specified remain in force as the program is executed until a new MASK instruction is encountered.

If no MASK instruction is included in a program, MASK 3 is assumed (MASK 1, Delete invalid character & MASK 2, Beep for each invalid character). You may include multiple MASK instructions in your program.

6.4 FORMATS

Each format is composed of a list of field controls, separated by commas. The entire format must be enclosed in parentheses. Each field control is one of the characters listed below, optionally followed by an integer to indicate the number of repetitions of that character. A character string (within quotes) may also be used as a field control, and it will be displayed in free-format.

Code	Explanation
U	Enter any character (including all punctuation characters) except cursor control functions. Input of all characters specified is required; an error will occur if you do not enter data exactly as the format specifies.

W	Enter any character except cursor control functions. Entry of data is optional; input of exact number of characters is not required.
A	Alphabetic, A to Z or space (input of all specified characters is required)
D	Alphabetic, A to Z or space (input of exact number of characters specified is not required)
Z	Numeric, 0-9 Input is required, thus non-significant zeros must be entered if the number to be input is less than the number of digits specified.
N	Numeric, 0-9 (input not required, so non-significant zeros are not needed).
B	Alphanumeric (A-Z, 0-9) (input of all specified characters is required).
C	Alphanumeric (input of exact number of specified characters is not required).
+	You must enter a '+' or '-' before the first significant digit.
-	You must enter a '-' before the first significant digit, if negative. Entry of a '+' sign is optional.
E	Any character string may be entered (length is as specified by the DCL statement).
L	Not used for ASK statement (PRINT only).
V	Implicit decimal point (computer places decimal point in input data). Used to convert units in which input specified. For example, allow input of percentage as whole numbers, not fractions. See examples for illustration.
*	Replaces all non-significant zero in an output with asterisks.
.	Explicit decimal point (User must enter decimal point).
X	Skip one position (space)
/	Skip one line vertically.
"TITLE"	Literal string.

Each field control is matched on a one-for-one basis with the variables in an ASK or PRINT statement. If there are more field controls than variables, the extra controls are not used. If there are more variables than controls, then the format is recirculated; that is, the next variable will use the first field control again. The appearance of a new format cancels any previous format, even if some field controls were not used.

6.4.1 Examples

1. `100 ASK=1: "REFERENCE" = (ZZZZZ), R1`

   The computer requests the user to input R1, the value of the Reference. Data must be 5 required digits without a sign.

2. `100 ASK=1: "TICKET OFFICE CODE" = (NNNZZ), C(K)`

   The computer requests the user to input the Ticket Office Code, C(K). The code is composed of at least two required digits (ZZ) and up to three additional digits (NNN), without a sign.

3. `100 ASK=1: "TOTAL" = (+++++), T`

   The computer requests the input of the total, T, which must be entered as a leading + or - sign followed by 0 to 4 digits.

4. `100 ASK=1: "VALUE" = (-----), V`

   The computer requests the input of the value, V, which must be entered as 0 to 5 digits. For a positive number, the typing of the + sign is optional. For a negative number, the typing of a minus sign as the left-most character is required. If a sign is entered, it counts as one of the five input digits.

5. `100 ASK=1: "UNIT PRICE" = (NNNVNN), P`

   The computer requests the input of the Unit Price, P, which must be entered as five digits without a sign or decimal point. The price is typed in cents, but the computer converts it to dollars with the implicit point.

6. `100 ASK=1: "UNIT PRICE" = (NNN.ZZ), P`

   The computer requests the input of the Unit Price, P, in the floating point format. You must type three optional digits representing whole dollars, then the decimal point, followed by two required digits.

7. `100 ASK=1: "WEIGHT IN GRAMS" = (NNVNNN.NN), P1`

   The computer requests the input of the Weight in Grams, P1, which must be entered in the floating point format. The number is entered as up to 7 digits, with a required decimal point. Because the V format is used, the computer will convert the value of grams to kilograms before internal usage.

8. `100 ASK=1: "ARTICLE CODE" = (ZZAAAUZ), C(L,J)`

   The computer requests the input of a string of 7 required characters: 2 digits, then 3 letters, then any character, then a digit.

9. `100 ASK=1: "ARTICLE CODE" = (NNNAAZ), C`

   The computer requests the input of the Article Code, C. This code must be entered as 0 to 3 digits, followed by 2 letters (required), then 1 digit. The following entries would be valid for this format:

   ```
     XT3
   121T3
   82TT3
   ```

10. `100 ASK=1: "CHECK DIGIT FOR", (Z9), S = (Z), C`

    The computer displays the message CHECK DIGIT FOR and the value of variable S. The format (Z9) controls the display of S. Then the variable C, which must be a single digit, will be accepted.

Note: In any format specification, several identical field control elements can be replaced by that element, followed by a number to represent the repeat factor. For example:

```
   NNN.ZZ can be N3.Z2
NNVNNN.NN can be N2VN3.N2
```

CHAPTER 7. DATA FILES WITHIN THE PROGRAM

7.1 INTRODUCTION

Three instructions permit establishing and accessing data within the program:

- o DATA to establish a data set (a set of numeric or string constants)
- o READ=0: to retrieve the data and assign it to a group of variables.
- o RESTORE to be able to reuse the data after the first usage.

7.2 DATA INSTRUCTION

The elements of the DATA set are placed in the segment by using one or more consecutive DATA instructions of the form:

DATA c_1, c_2,.........., c_n

The user enters numeric or string constants separated by commas (no comma before c_1 or after c_n). The DATA instructions must be grouped together in each segment, and must be placed at the end of the segment. The effect of several DATA instructions in the same segment is cumulative. This means that the values that are included are considered as though they were in a single DATA statement in the corresponding order. For example:

```
200 DATA 1.25, 3., "VALUE"
210 DATA 14, 12
220 DATA "NO"
```

is equivalent to one DATA instruction:

```
200 DATA 1.25, 3., "VALUE", 14, 12, "NO"
```

This permits entering several DATA instructions as a single line.

The DATA instructions can only be read in the segment where they have been defined. They are ignored by the other segments. Each segment can contain DATA instructions.

The constants, C, are written according to the rules enumerated below. The maximum number of constants in any one segment is 32,767.

7.2.1 Decimal Constants

A decimal constant is composed of:

- o A sign (required if value negative; optional if positive).
- o The integer portion of the constant.
- o The decimal point. } optional
- o The fractional portion of the constant. } optional

Examples: 12 123.5 .145

7.2.2 Character Strings

Character strings consist of some number of alphanumeric characters enclosed in quotes.

Examples: "VALUEb̸b̸" "b̸b̸b̸b̸NOb̸b̸b̸"

7.2.3 Hexadecimal Constants

A hexadecimal constant consists of the two hex digits preceded by a slash, /.

Examples: /FF /4E

Note that the data incorporated in the DATA statement in a segment can be modified by editing the instructions and recompiling the program.

7.3 READ INSTRUCTION

The data in the DATA instruction in any segment is used in the program by READ instructions in that same segment.

Syntax:

READ=0: $v_1, v_2, \ldots\ldots, v_n$

Where v_1 through v_n are short numeric variables or simple characters or indexes, separated by commas.

The 0 indicates that the file is defined in the program. The READ=0 instruction refers only to DATA files in the same segment.

Upon execution of the READ instruction, each variable in the instruction is assigned the next DATA value in sequence. The program checks to be sure that the variable type corresponds with the data value to be assigned to that variable. That is, a character string cannot be assigned to a numeric variable, etc. If this correspondence is not satisfied, an error message is output and the program aborts.

Example:

```
        140 READ=0: A, B(1), L, C(L), L2, C1
        ..................
        0150 READ=0:  D. H
        ..................
        200 DATA 12, 14, 1, "NO"
        210 DATA 15
        215 DATA "VALUE", 17, 3.05
        250 ESEG 3
```

After the execution of instruction 140, the values associated with variables requested are:

```
           A  = 12
         B(1) = 14
           L  = 1
         C(1) = "NO"
          L2  = 15
          C1  = "VALUE"
```

and after the execution of instruction 150, the variables associated with D and H are:

```
           D = 17
           H = 3.05
```

Summarizing:

Although there may be more than one DATA statement in a given segment, the net effect is the same as if all pieces of data were in the same DATA statement.

The DATA statements must immediately precede the ESEG statement with no other executable instructions intervening.

Several READ=0: statements may read successive pieces of data from the same or multiple DATA lines.

After the DATA is all exhausted, it is not automatically restored upon return to the beginning of the file. A RESTORE instruction is needed.

7.4 RESTORE INSTRUCTION

The effect of reading data from a DATA statement is as if a pointer was being moved to the next available piece of data after each READ of a variable. Once a piece of data has been READ, it cannot be used again (the pointer has passed it by) - unless the RESTORE command is used to re-activate all the data of the segment. The effect of RESTORE is to move the pointer back to the first constant in the data list.

The syntax of the instruction is:

```
            RESTORE
```

If (by error) a segment does not contain a DATA statement and a RESTORE instruction is issued, it is ignored and execution of the program continues at the next statement.

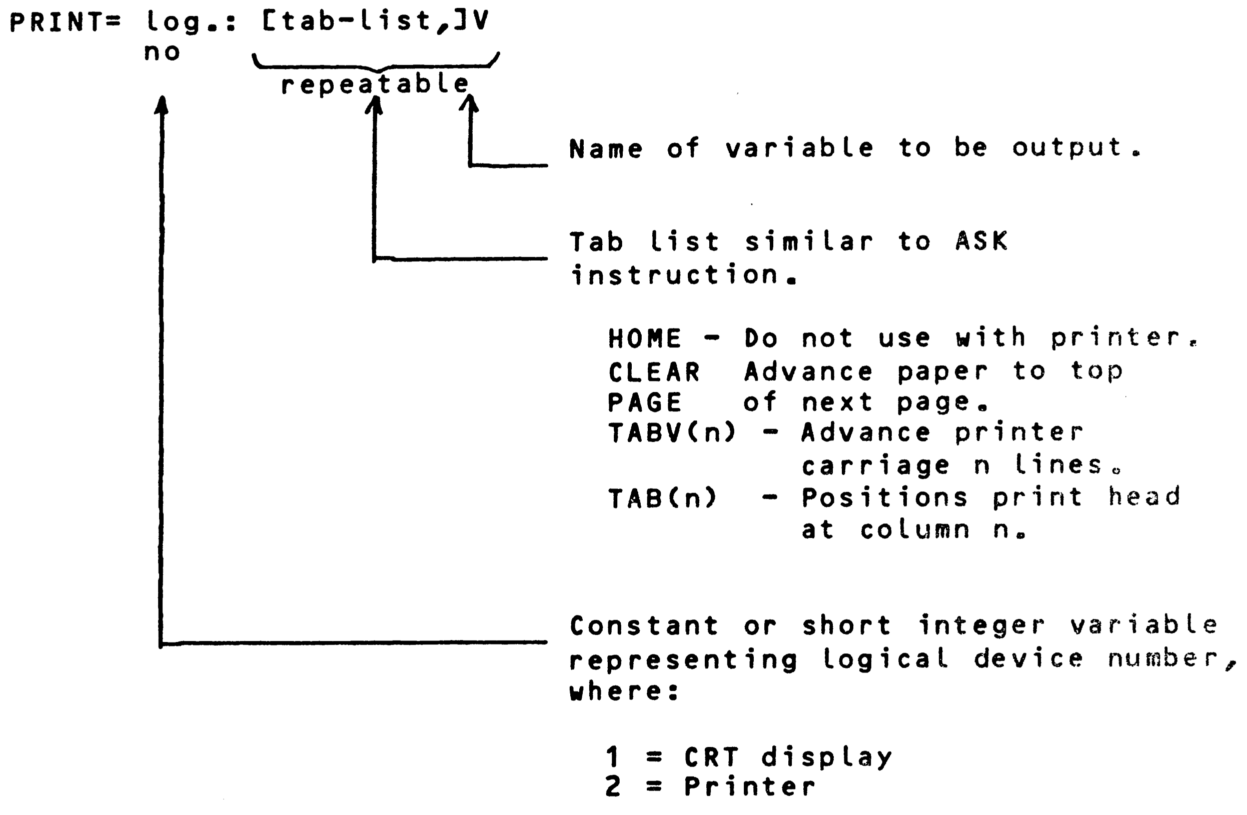
CHAPTER 8. PRINT INSTRUCTION

8.1 GENERAL

BAL uses the PRINT instruction to provide display information and program results on the printer or the CRT screen. There are two forms of this instruction: The simple form, and a more general form complete with formatting specifications.

8.2 SIMPLE PRINT INSTRUCTION

The format is:



Name of variable to be output.

Tab list similar to ASK instruction.

HOME -	Do not use with printer.
CLEAR PAGE	Advance paper to top of next page.
TABV(n) -	Advance printer carriage n lines.
TAB(n) -	Positions print head at column n.

Constant or short integer variable representing logical device number, where:

1 = CRT display
2 = Printer

When this instruction is executed, the computer performs the specified tab functions, then outputs to the specified logical device in free-format.

Examples:

1. 100 PRINT=2: A,B

   This instruction prints the values of A and B on the printer.

2. 200 PRINT=2: TAB(20), V1, TAB(50), V2

The system will first move the printing head to the 20th position; then print the value of variable VI. The printing head will then move to position 50 and print variable V2. The horizontal locations 20 and 50 are counted from the first column position (of the screen or the printer)--just as in the ASK statement.

3. 300 PRINT=1: CLEAR, TAB(20), H7, TABV(3), TAB(20),Y

The CRT screen will be erased; the cursor will move to column 20 and display H7. Then three lines will be skipped, the cursor will move to column 20, and Y will be displayed.

8.3 PRINT INSTRUCTION - GENERAL FORM

Syntax:

PRINT= log.[,format line no.]:$\underbrace{\text{[Tab list,][(Format),][Vi],}}_{\text{repeatable}}$

Note that the difference from the simpler form of PRINT is the use of formatting to establish a specific format for output of data, rather than allowing data output in free format. BAL offers comprehensive format control, as explained below.

8.4 FORMAT CONTROL

A format is composed of elements which control: spacing and tabbing, vertical spacing, and the printing of numeric and string variables and string literals.

Three types of format control can be used in printing, as described in following paragraphs:

8.4.1 Imbedded Format Control

The simplest method of format control is to imbed the format control characters in the list of variables to be printed:

```
PRINT=n:[Tab list,](format), Vi
```

For example: PRINT=2: TABV(1), TAB(5), (ZZ), V(J)

In this example TABV and TAB control spacing, (ZZ) specifies the format of the output variable as two required numeric digits. Note that only two digits may be printed, no matter what the actual size of V(J) may be. (See 8.4.4 for a complete description of format control characters.)

8.4.2 Fixed Format (FMT) Instruction

The FMT instruction is used to specify a format external to the Print instruction. This format control statement can then be used to control the output format of more thn one Print instruction.

```
The general format is:    PRINT=n, format line no: V1,...Vn
                      Line No. FMT(item, item, ....,item)
```

Note that all items in the FMT instruction must be separated by commas and the entire list of items must be in parentheses.

```
Example:    PRINT=2, 90: V1, V(J)
                  .
                  .
        90 FMT(/1, X5, ZZ, X5, N.ZZ)
```

In this example, the printer will skip one line (/1), space 5 positions (X5), print variable V1 in the ZZ format (two digits must be printed, even if zeros), space 5, print variable V(J) as one optional digit, decimal point, two required digits. (See 11.4.4 for complete details on format control characters.)

As previously mentioned, this FMT statement can be used by several Print instructions. As the line is printed, each item to be printed is matched with the next format control character in the list and printed in the specified format. Note that an error will occur if you try to print a string with a variable format control, or vice versa.

If more format control items occur in the list than variables to be printed, those format controls left over are ignored. If there are more variables than format controls, the list of format controls is re-circulated until all variables are printed.

The FMT statement is fixed and cannot be varied during the execution of a program. Note that FMT is a non-executable statement and provision must be made to branch around it. If a FMT is sequentially encountered during program execution, there will be a run-time error. One simple solution is to group all of the FMT statements together and place them below the last STOP instruction of a segment.

8.4.3 Variable Format (FM) Instruction

This instruction has a similar effect to the FMT instruction, but has an important difference. It is an executable instruction and can be changed during program execution. Thus, you can change the format used by a Print instruction as the program is executed.

The general form of this statement is:

```
    S= FM(item, item,....,item)
```

then,

```
    PRINT=n:((S)), V1,...Vn
```

Note that the item list in the FM statement has the same form as the item list in the FMT statement. It must consist of blanks, control items or constants. Variables are not allowed. In this case, the FM statement is assigned to a variable (string type only). This variable is then specified in the Print instruction within two sets of parentheses (()).

Using the FM statement in our previous example, the printout would be identical:

```
    S=FM(/1, X5, ZZ, X5, N.ZZ)
    PRINT=2:((S)), V1, V(J)
```

Note, however, that you are not restricted to one FM format per instruction. This provides a great deal of power.

Example:

```
    S=FM(/1, X5, "ACCOUNT NUMBER:", X5, ZZZZZ)
    A=FM(/1, X5, "SUBTOTAL: $", X11, ZZZZZ)
    PRINT=2:((S)), V1, ((A)), V(J)
```

This example prints the value of two variables, each controlled by a variable format. One of the variable formats could be changed, and the instruction used again, this time to print a total.

8.4.4 Format Control Characters

The format control characters, their effect on your program, and examples are presented below. Note that these formats control output (printing) on both the printer and CRT display, as selected by the Print instruction.

The format control characters are the same for ASK and PRINT instructions, but their effects are different as you will see in the following descriptions.

Format Control Characters - Numeric

Char.	Description	Example Format	Example Data	Example Printout
N	Print a number 0-9. Leading zeros are not printed, either before or after a decimal point.	(NNNNN) or (N5)	123	ƀƀ123
Z	Print a number, 0-9. Leading zeros will be printed, either before or after a decimal point.	(ZZZZZ) or (Z5)	123	00123
.	External decimal point, which will be printed in the specified output data.	(NNN.NN)	123	ƀƀ1.23
		(ZZZZ.Z)	12.3	0012.3
V	Implied decimal point. Specifies an internal representation of a decimal point, which is considered in the calculations. This will not print a decimal point in the output data if none is specified in the format. But it will force the specified decimal point to the correct position in the data.	(ZVZZ.ZZ)	3.125	0312.50
+	Requires the printing of the appropriate sign, plus or minus. If this control is represented as a single character (+NNNN) the sign will always be printed in the same position, to the left of the number. If the sign is represented as two or more characters (+++NN), the sign will float and be printed immediately to the left of the most significant digit.	(+NNNNN)	1234	+ 1234
		(+NNNNN)	12	+ 12
		(++++++)	1234	+1234
		(++++++)	-12	-12
		(+NNNNN)	123456	+23456
		Note that you can print only the number of digits, plus sign specified in the format. Be careful specifying formats to avoid truncating your output!		
-	Similar to the above, but requires the printing only of the minus sign. Plus is assumed by the absence of a sign in the printed output.	(-NNNNN)	-123	- 123
		(----NN)	-123	-123
		(------)	+123	123

L	Left justification of the printed data. All spaces in front of the number are suppressed, and an equal number of spaces are generated following the data. The sign is positioned to the right of the number.	(L--ZZZ) (L--ZZZ)	+12 -12	012b̸b̸ 012b̸-
*	Replaces all non-significant zeros in an output with asterisks.	(*ZZZZZ)	12	****12

<u>IMPORTANT NOTE</u>: When using a format, you can output only the number of digits specified in that format; as opposed to free format where all digits in the number print wherever the system places them on a line. If your format specifies fewer digits than actually exist, the output will be truncated and only the least significant digits will be printed. If a sign is specified in a format, it requires one of the allowable digit positions, when printed.

Examples: Actual value in memory =-123456.789 (will be in floating point format)

Format	Actual Printout
(N6.N3)	123456.789
(N9)	123456789
(N5)	56789
(-N9)	-12345678
(-N10)	-123456789

<u>Format Control Characters - Strings</u>

		Example		
Char.	Description	Format	String	Printout
U	Print any printable character. Unprintable characters, such as cursor control characters, will be mapped to blanks.	(UUUUUU) or (U6)	FORMA$	FORMA$
	NOTE the following special cases: 1. String <u>shorter</u> than format specified--Available characters printed left justified, remainder of format field filled with blanks. Note that blanks are not truncated, they take up space on the listing or screen	(UUUUUU) or (U6)	FOR	FORb̸b̸b̸

	2. String longer than format specified --Characters printed from beginning of string to number of positions specified. Remaining characters are lost.	(UUU)	FORMAT	FOR
	NOTE: W, B, C, A, and D formats may be used. They have exactly the same effect as U. Note that this is not true when they are used in the ASK instruction.			
E	Prints a character string of any length, where length of the specified string is as defined in its DCL instruction.	(E)	Any Str.	Any Str.

Miscellaneous Format Control Characters

Xn	Advances print head (or cursor) n spaces from its present position, where n is a decimal constant.	(X10)	Moves print head 10 spaces to the right.
/c	Advances the form (or display cursor) c lines, where c is a decimal constant.	(/2)	Advances two lines
"TEXT"	A string literal to be printed. Can be any printable ASCII characters and must be enclosed in quotation marks within the format.	("HELLO")	HELLO

8.4.5 Examples Illustrating the Output Format Elements For Numeric Variables

1. (ZZZZZ) This format involves printing with all positions displayed; including non-significant zeros. The format assumes that the number is positive. A floating point number is truncated and only the integer is printed.

   Examples:

   23 is printed as 00023
   246.60 is printed as 00246

2. (NNNNN) This format involves printing only the integer portion of the variable. Non-significant zeros are not printed. This format assumes that the number is positive. As in the above example, a floating point number is truncated.

Example:

23 is printed as 23 - i.e., right-justified in the field.

3. (+ZZZZ) Specifies printing only the integer portion of the variable, preceded by an obligatory plus (+) or minus (-) sign. Non-significant zeros are printed.

Examples:

-21.72 is printed as -0021
21 is printed as +0021

4. (-ZZZZ) Specifies printing only the integer portion of the variable preceded by a minus (-) sign, if the number is negative. Non-significant zeros are printed.

Examples:

17 is printed as ƀ0017
-34 is printed as -0034
0 is printed as ƀ0000

5. (+NNNN) Specifies printing only the integer portion of the variable preceded by a plus sign or minus sign. The non-significant zeros are not printed; they are replaced by spaces.

Examples:

17 is printed as +ƀƀ17
-3.8 is printed as -ƀƀƀ3
0 is printed as +ƀƀƀ0

6. (-NNNN) Specifies the printing of only the integer portion of the variable preceded by a minus sign (if negative). Non-significant zeros are not printed.

Examples:

17 is printed as ƀƀƀ17
-3.8 is printed as -ƀƀƀ3
0 is printed as ƀƀƀƀ0

7. (+++++) Specifies the printing of only the integer portion of the variable with the number preceded by a plus or minus sign.

Examples:

17 is printed as b̸b̸+17
-64 is printed as b̸b̸-64
0 is printed as b̸b̸b̸b̸b̸

8. (-----) Specifies the printing of only the integer portion of the variable, with the number following the minus sign (if negative).

Examples:

13 is printed as b̸b̸b̸13
-6 is printed as b̸b̸b̸-6
0 is printed as b̸b̸b̸b̸b̸

9. (+++NNNN) Specifies the printing of only the integer portion of the variable with the number following the plus or minus sign.

Examples:

14 is printed as b̸b̸+b̸b̸14
23635 is printed as b̸+23635

10. (+++ZZZZ) Examples:

14 is printed as b̸b̸+0014
0 is printed as b̸b̸+0000

11. (---ZZ) Examples:

3 is printed as b̸b̸b̸03
-1 is printed as b̸b̸-01
0 is printed as b̸b̸b̸00

12. (NNNVNN) The V aligns with the decimal point in the internal representation. This format does not print a decimal point.

Examples:

3.18 is printed as b̸b̸318
25.648 is printed as b̸2564
0.0 is printed as b̸b̸b̸b̸

13. (NNNV.NN) The V aligns with the decimal point in the internal representation and the . in the external representation.

Examples:

3.18 is printed as ƀƀ3.18
25.648 is printed as ƀ25.64
0.0 is printed as ƀƀƀƀƀ

14. (NNVNNN.NN) In this format, the V of the internal representation is not in the same position as the . of the external representation. This type of format is used in the conversion of unit. Suppose that the calculations are done in kilograms; this format is then used to print the result in grams.

Example:

The value 3.125 kg is printed as b3125.00 in the printout.

When the V is absent, it is assumed at the same position as the decimal point.

15. (L---ZZZ) The L permits a left justification of the information. The value 12 is given as 012ƀƀ in the field. The value -12 would be given as 012ƀƀ-.

8.4.6 Examples Illustrating Formats for Character String Variables

1. (UUU) Specifies a 3 character string.

Examples:

String "ABCDE" is printed as ABC
String "A" is printed as Aƀƀ.

2. (E) Specifies the printing of all the characters of the string. The computer left justifies them on output. The string length is taken from the declaration.

Examples:

String "ABCDE" is printed as ABCDE.
String "A" is printed as A.

CHAPTER 9. INPUT/OUTPUT PORT INSTRUCTIONS

9.1 GENERAL

Input/Output Port instructions allow you to transfer a byte of data to/from the CPU input/output ports.

9.2 INPUT PORT

The format of this instruction is:

```
INP N1,N2
```

This instruction transfers the one byte contents of input port N2 into variable N1.

9.3 OUTPUT PORT

The format of this instruction is:

```
OUT N1,N2
```

This instruction transfers the one byte contents of variable N1 to output port N2.

N1 and N2 are short variables.

Note: Refer to your Hardware System Reference Manual for a discussion of I/O ports utilized, and applicable bit assignments.

CHAPTER 10. SEGMENTATION

10.1 GENERAL

All BAL programs can be written in segments, where each segment consists of an independent program which works with variables that are common to all segments. Advantages of this scheme are:

1. Individual segments can be translated and debugged without the need to translate the entire program. This saves time for large programs.

2. Large programs that cannot be contained in main memory at one time can be segmented. The BAL operating system will load all segments into memory at once if space allows. If not, BAL will automatically swap segments from disk to memory as the program is executed. This is done as the LDGO.SEG instruction is used and is transparent to the user.

10.2 SEGMENT DECLARATION INSTRUCTIONS

A segment begins with the instruction:

```
      SEGMENT c[,NOLIST]
```

where c is a decimal constant between 0 and 15.

A segment ends with the instruction:

```
      ESEG c
```

where c has the same value as its associated SEGMENT statement.

Example:

```
        1 PROGRAM "TEST"
        2 FIELD=M
       10 DCL...........
      100 SEGMENT 0
      ... .............         SEGMENT 0
      ... .............
      900 ESEG 0
```

```
 600 SEGMENT 2             \
 ... ..............         >SEGMENT 2
 ... ..............        /
1200 ESEG 2

 100 SEGMENT 5
 ... ..............          SEGMENT 5
 ... ..............
1100 ESEG 5
1999 END
```

This program is composed of 3 segments, numbered 0, 2, and 5.

NOLIST (optional) indicates that the current segment is not to be included on a program listing. However, if an error is detected in the segment, a listing of the segment begins with the instruction in error.

Note that:

- o Segment 0 is the principal segment and is resident in memory. It must always be present.
- o Segments need not appear in the program in sequential order (with the exception of Segment 0).
- o A maximum of 16 segments may be in a program.
- o The line numbering of the BAL instructions within a segment must always be sequentially increasing. However, each segment has separate line numbering.
- o The last instruction of any program is always END, whether there is one or many segments. The line number (if used) of the END statement must be higher than the immediately preceding instruction (as is necessary for the instruction that terminates a segment: ESEG).

10.3 CALLING SEGMENTS

Each segment can call one or several segments with the instruction:

```
LDGO.SEG v
```

whose role is to load segment v into main memory and begin execution with the first instruction of segment v. The variable v is a short integer variable or a constant.

Example:

```
     100 SEGMENT 0
     ... ............
     500 LDGO SEG 3
     ... ............
     600 ESEG 0
     ... ............
     100 SEGMENT 3
     ... ............
     ... ............
     900 ESEG 3
```

Suppose that segment 0 is active. At the execution of instruction 500, the computer will transfer control to segment 3, loading it from disk into central memory if it is not already there. Control is then passed to the first instruction of segment 3.

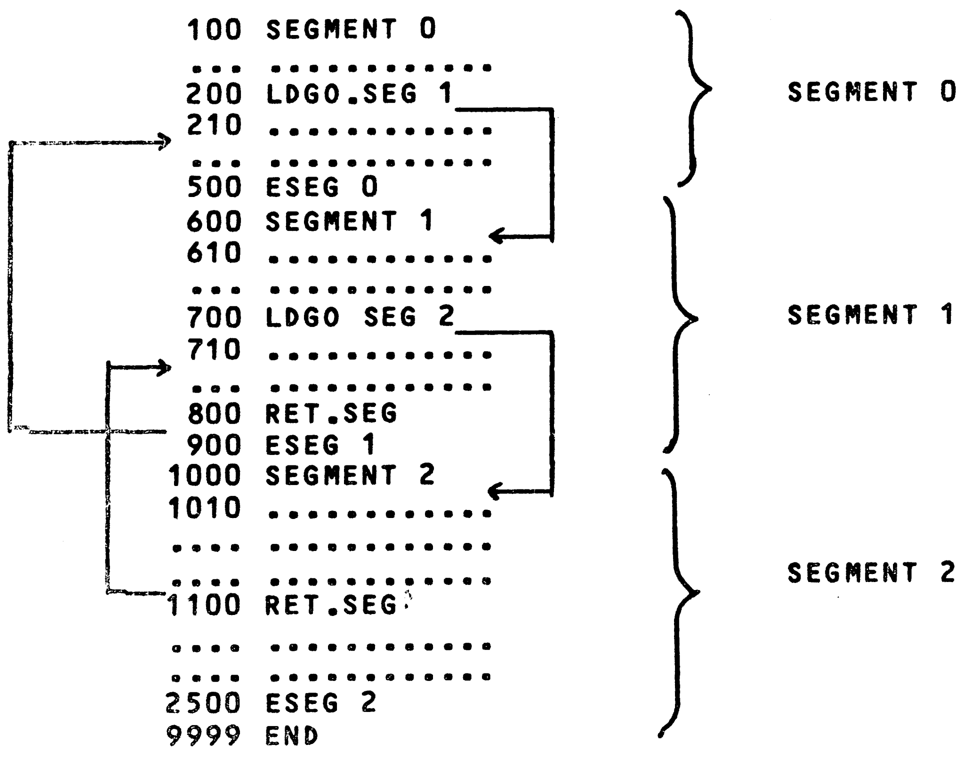
10.4 RETURN FROM A SEGMENT

When a computer encounters the instruction:

```
          RET.SEG
```

it passes control to the instruction following the last segment call (LDGO.SEG). The segment left by the return is deactivated.

10.4.1 Example 1



```
      100 SEGMENT 0
      ... ............
      200 LDGO.SEG 1
      210 ............
      ... ............
      500 ESEG 0
      600 SEGMENT 1
      610 ............
      ... ............
      700 LDGO SEG 2
      710 ............
      ... ............
      800 RET.SEG
      900 ESEG 1
     1000 SEGMENT 2
     1010 ............
     .... ............
     .... ............
     1100 RET.SEG
     .... ............
     .... ............
     2500 ESEG 2
     9999 END
```

At instruction 200 of segment 0, segment 1 is loaded (if not already in memory) and control transferred to instruction 610 of this segment. At instruction 700 of segment 1, LDGO.SEG 2 ends execution of segment 1, loads segment 2 (if necessary) and passes control to, instruction 1010 of segment 2.

In segment 2, instruction 1100, a RET.SEG, is encountered. Segment 2 is ended, segment 1 is reloaded if not in memory, and control is returned to the instruction following the last segment call, in this case instruction 710 of segment 1.

The return from segment 1 to segment 0 is a similar operation. Note that a STOP statement could be used in any of these segments to halt the program.

Note:

- o A segment attempting to call itself results in an execution time diagnostic error.
- o The number of segments that can be used in central memory depends on main memory capacity and the length of the segments.
- o The variables are common to all the segments. Local variables do not exist.

You can compare the function of calling a segment to that of calling a subroutine.

CHAPTER 11. MISCELLANEOUS INSTRUCTIONS

There are three instructions that can be used to halt a program. Two cause a temporary halt; one causes a return to the operating system when encountered.

11.1 WAIT INSTRUCTION

Syntax:

```
WAIT v
```

where v is a short integer variable or constant. It causes a halt of v seconds, after which the program will continue execution. Any characters typed during this waiting period are ignored, with a beeping tone sounded.

Example:

```
258 WAIT 15
```

will cause a halt of 15 seconds.

11.2 PAUSE INSTRUCTION

Syntax:

```
PAUSE v
```

where v may be any variable or literal. The variable v will be displayed on the screen and the system will halt. To continue execution, enter a carriage return (RETURN).

Examples:

300 PAUSE 33 will display a 33 on the screen.

500 PAUSE "HALT 5" will display "HALT 5" on the screen.

600 PAUSE I will display the current value of I.

11.3 STOP INSTRUCTION

Syntax:

```
STOP
```

The program will halt and return control to the operating system.

11.4 OP ADR

Syntax:

```
            OP ADR
```

If any character was typed on the keyboard, the program branches to address ADR; otherwise, sequential program execution continues. The typed character is not displayed, and of itself has no effect on the program.

CHAPTER 12. FUNCTIONS

12.1 GENERAL

The BAL language uses internal functions that are classed in the following categories:

1. Mathematical functions
2. String functions
3. Miscellaneous functions

The functions are used with the assignment instruction.

Example:

```
100 LET X = ABS(B) or 100 X = ABS(B)
```

12.2 MATHEMATICAL FUNCTIONS

12.2.1 Function ABS(v)

This function calculates the absolute value of a number.

Examples:

```
ABS(14.3) gives 14.3
ABS(-14.3) gives 14.3
```

12.2.2 Function INT(v)

This function provides the largest integer less than or equal to v.

Examples:

```
INT(-3.9) is given as -4
INT(-3.1) is given as -4
INT(-3)   is given as -3
INT(0)    is given as 0
INT(3.9)  is given as 3
```

12.2.3 Function MOD(B,C)

This function computes the absolute value of the remainder of B divided by C. B and C must be variables of the same type.

Example:

```
A = MOD(B,C)
A = MOD(7,3)
A = 1
```

Note: A and B must be different variables

12.2.4 Function ROUN(vl,N)

This function rounds off the number in v_1 to the number of decimal places indicated by N.

Example:

```
X = ROUN(Y,2)
```

If Y = 1289.864, X takes the value 1289.86
If Y = 1289.8681, X takes the value 1289.87
If Y =-1289.8681, X takes the value -1289.87

v_1 must be a long variable.
N must be a short integer variable.

12.2.5 Functions FIX(x) and FP(x)

The two functions permit the decomposition of a number into its integer part and its fractional part.

Example:

A=FIX(3.9) is given as 3
B=FP(3.9) is given as 0.9

The variable v must be a long variable.

12.2.6 Function SGN(v)

This function gives the sign of v.

Example:

```
Y = SGN(x)
```

The value of Y is 1 if X > 0
The value of Y is 0 if X = 0
The value of Y is -1 if X < 0

12.2.7 Function CONV(v)

This function permits converting a short variable to a long variable; a long variable to a short variable (with truncation); or a string to a numeric (and vice versa).

Note: When a numeric is converted to a string, the resulting string is right justified. Because all strings are normally left justified, you may wish to use the SHL function to left justify the string.

A similar situation occurs when converting a string to a numeric. The resulting number will be left justified in its field, where the normal number is right justified. The SHR function will right justify the numeric.

Example:

```
Numeric F = 12345
String S = ABCƀƀƀƀƀ
After S=CONV(F), S=ƀƀƀ12345
After S=SHL(S), S=12345ƀƀƀ
```

12.3 STRING FUNCTIONS

12.3.1 Function LEFT(A,N)

This function yields a sub-string containing the first N characters of A.

Example:

```
X = "ABCDEF"
Y = LEFT(X,3)
```

Y will be "ABC"
N must be a short integer variable.

12.3.2 Function RIGHT (A,N)

This function yields a sub-string containing the last N characters of A.

Example:

```
N = 3
X = "ABCDEF"
Y = RIGHT(X,N)
```

Y will be "DEF"
N must be a short variable

12.3.3 Function LEN(A)

This function returns the length of string A as declared in a DCL statement.

If A was declared as DCL A$=20, then for:

```
N = LEN(A)

N will be 20.
N must be a short variable.
```

12.3.4 Function INDEX(A,B)

This function returns the position of the first character of string B in string A.

Note: When a numeric is converted to a string, the resulting string is right justified. Because all normal strings are left justified, you may wish to use the SHL function to left justify A similar.

Example:

```
A = "ABCDEF"
B = "CD"
N = INDEX(A,B)

N will be 3
N must be a short variable.
```

12.3.5 Function INSTR(A,B,N)

This function returns 0 or 1, depending upon whether or not string B exists in string A, starting from the Nth character of A.

Example:

```
A = "ABCDEF"
B = "CDE"
N = INSTR(A,B,3)

N will be 1
N must be a short variable.
```

12.3.6 Function SUBSTR(A,N1,N2)

This function returns a sub-string of length N2 extracted from A, starting from the N1th character.

Example:

```
A = "ABCDEF"
B = SUBSTR(A,3,2)

B will be "CD"
N1 and N2 must be short variables.
```

12.3.7 Function INCLUD(N,B,[N1])

Replaces characters in specified string, starting with the Nth character of that string. N1 characters are replaced by the first N1 characters of string B. If N1 is not specified, the entire string B is used.

Example:

```
A = "1234567"
B = "ABC"
A = INCLUD(3,B,2)

A will be: 12AB567
Variable N must be a short variable.
```

12.3.8 Function VAL(A,N)

This function returns a number containing the BCD numeric value of the string expressed using ASCII code. The conversion starts from the Nth character and stops at the first character that is not a number or a decimal point.

Example:

```
A = "ABC12.3DEF"
V = VAL(A,4)

V will be equal to 12.3
Variable N must be a short variable. The resulting variable (here V) must be a long variable.
```

12.3.9 Function STRN(X)

This function returns a string of characters containing the value of X expressed in ASCII code. The characters in the new string are left justified. A plus sign will be omitted, a minus sign will be included.

Example:

```
X = 123.45
A = STRN(X)
```

Variable A will be "123.45b̸b̸b̸b̸"
X must be a long variable.

12.3.10 Function TRAN(A,B,N,C)

This function translates the characters of string C, using string A as the identifier characters and string B as substitution characters. The characters of string B map one to one for string A for a length N of string B.

Example:

```
A = "ABCDEF"
B = "1234"
C = "ABCDEFAGBHC"
D = TRAN(A,B,3,C)
```

String D will be "123DEF1G2H3".
Variable N must be declared as a short variable.

12.3.11 Function INV(A)

This function returns the inverted form of the specified string, A. Note that A and B must be different variables.

Example:

```
A = "12345"
B = INV(A)
```

B would then be "54321".

12.3.12 Function GENER(N,A)

This function generates the first character of string A exactly N times.

Example:

```
A = "DEF"
B = "BBBB"
B = GENER(3,A)

B will be "DDDB"
N must be a short variable.
```

12.3.13 Function SPACE(B)

This function generates a string of B blanks.

Example:

```
A = ABCDEF
A = SPACE(6)
A = ƀƀƀƀƀƀ
```

12.3.14 Function DATE(N)

This function initializes a character string with the date or time according to the value of N, where N is:

```
1 year
2 month
3 day of the month
4 day of the year
5 hour
6 minute
7 second
8 tenth of a second
```

Examples:

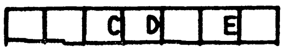
```
A = DATE(1)    A = 1980
A = DATE(2)    A = 2ƀƀƀ (February)
```

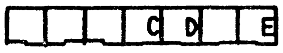
12.3.15 Function SHR(A)

This function right shifts character string A until the right-most character is non-blank.

Example:

```
A="ƀƀCDƀEƀ     or
B=SHR(A)
B now is "ƀƀƀCDƀE" or
```





12.3.16 Function SHL(A)

This function left shifts the character string A until the leftmost character is non-blank.

Example:

Taking the original value of A from the above example, B=SHL(A) would give:

B= CDbEbbb   or

C	D		E			

12.4 MISCELLANEOUS FUNCTIONS

12.4.1 Function PEEK

PEEK is used to return the contents of any memory location.

```
I=PEEK(M)     Where I is type 1 short variable (one byte)
                    M is type 2 short variable.
```

When this instruction is executed, I will contain the contents in decimal of the memory location specified by M.

Example:

```
I=PEEK(/EBCO)
```

I will contain the one byte contents of memory location/EBCO, expressed as a decimal single byte variable (-128 to +127).

12.4.2 Function POKE

The format of this function is:

```
M=POKE(I)     Where: M is a type 2 short variable.
                     I is a type 1 short variable (one
                     byte, decimal), or a one byte
                     hexadecimal constant.
```

This instruction loads the byte specified by I into the memory address specified by M (in decimal or hex). This can be useful in a program where you wish to modify certain operating system parameters.

12.4.3 Function VPTR(5) - Variable Pointer

This function is used to determine the memory address of a variable. The format is:

```
            M=VPTR(S)
```

This instruction will return the memory address (in decimal) of the variable S. M must be a short numeric variable. S can be any variable name.

CHAPTER 13. DISK ACCESS FEATURES

13.1 GENERAL

This chapter contains descriptions of a number of instructions which provide access to various disk features. These instructions are often used with the File Management System described in Chapters 14 and 15, but are described in this separate chapter because they are available in BAL whether or not your PROLOGUE system has been configured with the File Management System options.

These instructions are:

ASSIGN	- Assigns a logical number to a device and establishes various I/O characteristics.
IO	- Used for direct I/O, on a sector-addressed basis, with a disk.
LOAD	- Used to load an object program (usually a subroutine) into memory under BAL control.
CALL	- Calls a program loaded by LOAD.
CHAIN	- Used by one BAL program to load and transfer control to another executable BAL program.

13.2. ASSIGN

This instruction must be executed in order to access a support device for I/O. Note that ASSIGN has options in addition to those discussed here, which are used with File Management System instructions. This is discussed in paragaph 14.2.1.

Syntax:

```
ASSIGN = N.LOG, Device [,Options][:ERROR]
```

Where:

N.LOG	- Logical number from 1 to 15. (See Chapter 14 for a detailed definition of logical numbers.)
Device	- Must be a string variable or literal, expressing the unique name of a support device, such as FLØ.

Options	- Code	Definition	Default
	WR	Open for Writing	Read only

:ERROR - Optional error branch parameter, consisting of two parts, :ADDR, E.
- ADDR is any program line number, which will be branched to if a non-zero status/-error code is returned.
- E is the variable in which the code will be returned.

(See Chapter 14 for a detailed discussion of the status/error code.)

This instruction performs the following functions:

1. If this instruction references a logical number currently assigned to a file, that file is closed.

2. The syntax of the support device name is analyzed, and an error message is returned if incorrect.

3. A descriptor is prepared for the file routines, containing the designation of the specified support device and the WR option, if chosen.

This instruction assigns a logical number to the specified device, allowing you to read and write on the device using the IO instruction. Note that file management instructions, as described in Chapter 15, and the IO instruction are mutually exclusive. The IO instruction requires the programmer to manage file allocation, where the file management system handles that automatically.

13.3 IO Instruction -- Direct Access To A Support Device

Sometimes it may be useful to short-circuit the file system organization and put the files at the disposal of the user, or the file system may not be configured into your version of PROLOGUE. In this case disk I/O is handled using the IO instruction.

Syntax:

```
IO = N.LOG,Function [,Sector][:ERROR],Variable
```

Where:

N.LOG - Logical number from 1 to 15. (Must be one or two byte short numeric variable.)

Function - Must be a one or two byte short numeric variable specifying one of the functions provided by the PROLOGUE disk driver routine, as follows:

40H	-	Read
80H	-	Write
82H	-	Write and initialize (premark)

Sector - Starting sector address for the reading or writing of data. If omitted, default is sector Ø. This can be a long or short numeric variable.

:ERROR - Optional error branch parameter, consisting of two parts, :ADDR, E.
- ADDR is any program line number, which will be branched to if a non-zero status/-error code is returned.
- E is the variable in which the code will be returned.

Variable - Any variable containing the address of the data buffer where output data is to be found or input data is to be placed.

This instruction is thus used to write a block of data to a sector on the disk or to read a block of data into memory. Note that data is read or written on a sector-by-sector basis (256 bytes).

Example Program

This program recopies the contents of the diskette in floppy 0 onto the diskette in floppy 1.

```
    PROGRAM "COPY"
    DCL S%                           ;Sector number
    DCL E#                           ;Error code receptor
    DCL D$ = 256(16)                 ;Buffer (1 default granule)
    SEGMENT 0
    ASSIGN = 1, "FL0"                ;FL0 is logical #1, read only
    ASSIGN = 2, "FL1",WR             ;FL1 logical #2, read/write
 10 IO = 1,/40,S:90,E,D(1),4096        ;Read 1 granule from FL0
    IO = 2,/82,S:D(1),4096           ;Write data to FL1
    S = S + 16                       ;Next granule
    GOTO 10

 90 IF E = 4 GOTO 99                       ;Jump if end of disk
    PRINT=1:BELL,TABV(1),"ERROR ",E  ;Print unexpected error

 99 STOP
    ESEG 0
```

13.4 LOAD INSTRUCTION

This instruction allows the BAL program to load an object program (usually a subroutine) into memory.

Syntax:

```
        LOAD = N.LOG, Variable [:ERROR]
```

Where:

N.LOG	- Logical number from 1 to 15, as assigned to the object file by the ASSIGN instruction.
Variable	- Specifies the load address of the object program: a. If a short variable is supplied, it contains the loading address in memory for the object file. b. If a long variable or string variable is supplied, the address of this variable is the load address of the object program, i.e., the program is loaded into the specified variable and following variables up to the length of the program. In this case, it is the responsibility of the programmer to ensure that enough variable space is declared to accommodate the object program.
:ERROR	- Error branch parameter as described above, in the IO instruction.

Examples:

1.
```
ASSIGN = 1, "FLO.CVB"
LOAD = 1, /A000
```

Object program CVB (implicit type -0) is read from the disk and loaded into memory, starting at address /A000.

2.
```
DCL A%
SEGMENT 0
A = /A000                   ;load address
ASSIGN = 1, "FLO.CVB"
LOAD = 1,A
```

This program produces the identical result as example 1.

3.
```
DCL P$ = 256
SEGMENT 0
ASSIGN = 1, "LOAD"
LOAD = 1,P
```

The object file "LOAD", which is found on the user support device, is assigned logical number 1 and loaded into memory beginning at the address of variable P. If the object program is longer than 256 bytes, it will be located in the 256-byte area specified by P and those variables declared following P.

4.
```
DCL A%, P$ = 256
SEGMENT 0
ASSIGN = 1, "LOAD"
A = VPTR(P)
LOAD = 1,A
```

This produces the identical result as example 3.

13.5 CALL INSTRUCTION

This instruction is used to call a subroutine, specifying a parameter to be passed (which can be a pointer to a string of parameters), and either the starting address of the subroutine, or the name of the variable into which the subroutine was loaded by the LOAD instruction.

Syntax:

```
CALL Data,Address
```

Where:

Data	- A parameter to be passed to the subroutine.
Address	- Address of the subroutine, as: a. If the Address variable is a one or two byte short numeric, it contains the starting address of the assembly language program. b. If the Address variable is a long numeric or string variable, the address of the variable itself is the starting address of the assembly language program.

When the assembly language program is called:

Registers HL contain the starting address of the data.
Registers BC specify the length of the data.

Examples:

1.
```
DCL X
SEGMENT 0
CALL X, /A000                     ;Execute routine at /A000
```

2. In this example, we wish to pass several parameters (A,B,C,D,) to a subroutine. They are declared as a consecutive group, then the group is equivalences as parameter X, which is passed to the subroutine. The subroutine will of course be written to expect a group of parameters starting at the address of X.

```
DCL A$=4, B$=4, C$=6, D$=2    ; Declare A,B,C,D
FIELD =0,A                    ; parameter field starts
                              ; over at A
DCL X$=16                     ; Equivalence X to A,B,C,D
DCL P$=256                    ; Space for subroutine
SEGMENT 0
ASSIGN =1, "LOAD"             ; Assign log # to routine
LOAD=1,P                      ; Load subroutine
CALL X,P                      ; Call subroutine at P,
                              ; pass parameter X
```

3. In this example, we output the character contained in the variable I to the display.

```
DCL M%, I#, X$, J
SEGMENT 0
M = VPTR(X)                   ; Return address of X
FOR J = 1 TO 5
READ=0: I                     ; Read DATA into I
M = POKE (I)                  ; Put value if I into M
M = M=1
NEXT J
M = VPTR (X)
I = /41
CALL I,M                      ; Sent to CRT character /41,
                              ; 'A'
STOP                          ;
DATA                          ; Assl'y routine
REM  4E, /CD, /4C, /01, /C9
     LD C (HL); CALL 14C RET
ESEG
```

4. In this example, we input any character (including control code which normally would be filtered by ASK statement) from the keyboard to variable I

```
PROGRAM "INPUT"
DCL I#,M%, X$,J
SEGMENT 0
M=VPTR(X)                     ;M=address of var X
FOR J=1 TO 5
READ=0:I
M=POKE(I)
M=M+1
NEXT J
M=VPTR(X)
```

```
10     CALL I,M
       PRINTED :"VALUE OF INPUT CHARACTER" I,TABV(1)
       GOTO 10
       DATA  /CD,/49,/01,/77,/C9
       REM   CALL CI;LD(HL),A;RET
       ESEG  0
```

13.6 CHAIN INSTRUCTION

```
CHAIN =N;LOG [:ERROR]
```

This instruction can be used to chain type -T (BAL intermediate) files. One BAL program can execute an ASSIGN instruction to assign a logical number to another intermediate file, then execute a CHAIN instruction to load and execute that file. [:ERROR is the optional error branch parameter, as described in the 10 instruction.

Example:

```
PROGRAM "NAME"
DCL..........
.............
SEGMENT 0
.............
ASSIGN=2,"PGM2"
CHAIN 2
.............
STOP
ESEG 0
END
```

During normal execution of this programn the CHAIN 2 instruction is encountered (it can be anywhere in the program). When the STOP instruction is executed, porgram execution of the current program halts and Program PGM2 is automatically loaded from the disk and executed.

CHAPTER 14. INTRODUCTION TO FILE MANAGEMENT SYSTEM

14.1 GENERAL

The File Management System is an option of the PROLOGUE Operating System, which can provide you with a simple and convenient means of handling files of information. It includes three different means of handling data files, all of which handle the problems of mass storage "housekeeping" and allow your BAL program to manage data files by name on your PROLOGUE disk, using simple BAL language instructions. In your BAL program you need not concern yourself about about tracks, sectors, disk addressing, or calling file handling subroutines.

The three file management systems are:

1. Random Access (Relative Files) -- The files in this system are a sophisticated virtual memory.

2. Sequential File System -- Files are written to/read from memory and the disk in a sequential fashion, and appear one after the other, in the order they were originally written to the file.

3. Indexed Sequential Access Method (ISAM) -- This system organizes data files of any length by keyword, indexing them for very rapid random access.

These three systems are available as PROLOGUE options and operate within all the logical constraints of the PROLOGUE Operating System. You can obtain:

* Sequential and Random File Systems Only
* Indexed Sequential Access Method File System Only
* All three File Management Systems

Note: Random and Sequential are provided in a single software module, which is often referred to as the Sequential File System.

This chapter provides you with a general description of the File Management System and a glossary of the terms used in the Instruction descriptions. Chapter 15 describes the instruction syntax.

14.2 FILE SYSTEM GENERAL CHARACTERISTICS

14.2.1 General Types of Operation Performed

Each of the file systems operates with files of data, organized in different fashions. Certain types of operation must occur with each:

1. An ASSIGN statement must be executed. This refers to the file (or device) by name and assigns it certain characteristics. The functions performed by ASSIGN are:

   a. Specifies the file type, as random, sequential, indexed sequential.

   b. Assigns the file a logical number, which is then used by the various BAL instructions to refer to this file, rather than having to use the filename.

   c. Establishes whether writing is to be allowed on this file.

2. The file must be created, referring to the assigned logical number, and perhaps establishing additional characteristics.

3. The file must be OPENed to allow operations to occur. This positions an imaginary file pointer pointing to the correct data position in the file. (Many instructions, such as READ and WRITE, open the file automatically.)

4. Various read/write/positioning operations occur, depending on the file type.

5. The file is closed, and the logical number is released.

14.2.2 Status/Error Codes

Whenever a file system instruction is executed, the system returns a status/error code which indicates the result of the operation. This can be zero (Ø), which indicates that the command was executed as expected; can be a normal status indication, such as end-of-file, or can be an error indication. This code can then be analyzed in your program and appropriate action can be taken.

The parameter, ERROR, in which the code is returned is optional, but is dangerous to ignore, for the program is aborted if any non-zero status/error code is returned and no variable is available in which to place it.

14.3 DESCRIPTION OF RANDOM ACCESS FILE SYSTEM

As mentioned above, this system obeys all the constraints of the PROLOGUE Operating System regarding volume, filename, etc. It merely deals with data files, as opposed to source files or executable programs.

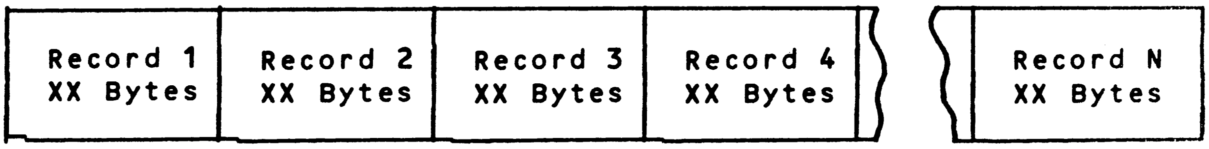
This file system operates as a sophisticated virtual memory. You begin by using a FIELD instruction in the declaration section of your program to declare a virtual memory file. This virtual memory file contains all variables subsequently declared by following DCL instructions until a new FIELD instruction is encountered.

Thus, once a random file is created, all of its elements are defined as variables. When the file is to be used, it is assigned a logical number, then the variables are used in your program just like any other variables. The differences are: 1) these variables are on the disk, not in memory, and 2) you can assign a file of random elements as either read/write or read-only.

14.4 DESCRIPTION OF SEQUENTIAL FILES

A sequential file is a group of contiguous records, such as a record of the day's transactions made by some type of data collection device in a retail store.

If you were to look at the sequential file on disk, it would appear thus:



You can perform the following operations with sequential files: create them, open them (required prior to read or write), write data into them (as a record, which is some group of associated bytes), read the data back, close the file, and delete the file.

Operations with the file always occur in regard to a file pointer. Items are always read with the pointer starting from the beginning of a file, and progressing down record-by-record to the item(s) required. Items are always added to the end of a sequential file, never in the middle.

Summary of important points regarding sequential files

1. A file must be assigned (by the file system ASSIGN instruc-

tion) before it can be used. This establishes it as a sequential file, assigns a logical number, and establishes whether or not writing is allowed. The logical number is then used for reference to the file by various BAL instructions.

2. Sequential files can be opened for reading or writing, but not both.

   a. If the file is opened for reading, the pointer is positioned before the first record in the file. A READ instruction reads that record and positions the pointer before the next record in the file.

   b. If the file is opened for writing, the pointer is positioned behind the last record in the file.

3. After a READ, the pointer points to the next sequential file (or to end-of-file).

4. A BACKSPACE instruction can be used to back the pointer to the preceeding file. No data is read by this instruction.

5. After a WRITE, the pointer is positioned behind the last record in the file.

6. It is not possible to modify records in a sequential file.

7. Status codes will be returned after each sequential file instruction is executed.

   a. Zero (Ø) indicates that the expected operation occurred with no errors.

   b. Any other code indicates either a status condition (such as end of file) or an error.

8. When using magnetic tape, you can write to or delete only the last file on the tape.

14.5 DESCRIPTION OF INDEXED SEQUENTIAL (ISAM) FILES

An indexed sequential file consists of any number of records of information (up to the storage capacity of the volume). Each record is composed of a unique record key and data associated with that key. As a simplified example, you can compare ISAM files to using your dictionary. If you wish to look up the meaning of the word FILES, you make one access to the section covering words beginning with F, another access to the page keyed as Fil or File, and a third access to the individual entry on that page which defines the meaning of the word Files. In this case, the word FILES is a unique key which points to associated useful data.

An ISAM file operates in a similar fashion. You specify a Record Key for each item stored by the system. This is a 2 to 20 byte unique identifier for every data item in the file. It could be item number, invoice number, part number, customer name, or some combination of data -- whatever you choose to uniquely identify your records.

These keys are organized and stored alphabetically by the File Management System in such a manner that whenever you wish to retrieve an item, you specify the key and the system looks it up and goes directly to the recorded data.

14.5.1 Characteristics Of The ISAM File

When an ISAM file is created, the following characteristics are defined:

1. File type -- file type is defined as:

   * Record Key Left Justified
   * Record Key Right Justified
   * Record Key Not Justified

   See 14.6 for more detail on justification of record keys.

2. Length of article -- Length of the data items to be stored.

3. Length of the Key -- All keys must be declared as the same length, 2 to 20 bytes.

14.5.2 Characteristics Of The ISAM Record

Once the ISAM file is created, you can insert, read, modify and delete individual records. The record has the following format:

Number of bytes	2	2 to 20	1	0 to 32,767
Type of Data	Optional Length Identifier	Record Key	Index	Data

Where:

Length Identifier - Optional, but if used returns the length of the data in an input record.

Record Key	- 2 to 20 bytes which uniquely identify this record.
Index	- One byte which allows the user to divide a file into 8 sub-levels. If not specified, assumed to be 1. See 14.6 for more detail on the Index.
Data	- This is the data item associated with the record key.

14.6 Definitions

This section includes detailed definitions of various terms used in the instruction description in Chapter 15.

1. Logical Number (N.LOG)

This is a number between 1 and 15, assigned to a file when it is ASSIGNed. It serves to uniquely identify this file and allows various BAL file system commands to refer to the file by number.

The file system permits you to open and manipulate several files at one time, a minimum of 4 (default value) and a maximum of 16. The maximum number of open files is established when your PROLOGUE system is configured. The Logical Number (N.LOG) can be a constant or short variable of value 1-15.

If you attempt to use an illegal number, or open a file when the maximum number is already open, an error will be returned.

2. Sector Address

In certain Random file system instructions, a sector address can be specified. This refers to a disk sector of 256 bytes, where the first sector of the first track is sector 0, and sectors are numbered consecutively through the maximum number available on the disk.

3. [:ERROR] & Status/Error Code

The optional parameter [:ERROR] contains two elements, ADDR, E. ADDR specifies the line number for a program branch which is taken any time the file system returns a status/error code unequal to zero.

The two byte integer variable E is used to receive the status/error code which is returned after every file system instruction is executed (or attempted). The status-/error code indicates normal instruction execution, a normal status condition, or an error condition. Appendix C contains a complete list of error codes which can be returned by the system.

Although this parameter is optional, it is usually important to analyze the codes that are returned and take appropriate action. If a non-zero code is returned by the file system when no ERROR branch parameter is available, the program displays the error code and debug address of the failing instruction, then aborts and returns to the PROLOGUE command level.

4. Input/Output Buffer

Sequential and Indexed Sequential instructions which read and/or write data require you to establish a buffer area in memory (via the DCL instruction), used for the input or output of data. In a BAL file instruction you can specify the name of your buffer variable and its length. This allows you to specify a length longer than the declared variable length and set up various overlays as your buffer. As a default case, the declared length of your buffer variable is assumed.

Both sequential and indexed sequential files require you to specify the data length, theoretically up to 32K bytes of data, but actually limited by your available memory space.

Not all instructions read/write the same size buffers, so specify carefully.

5. Record Key

The Record Key serves to uniquely designate a data item in an Indexed Sequential file. Each key is 2 to 20 bytes in length, all keys are a standard length for any one file, and no two keys may be the same.

The keys are stored in a separate file on the diskette (file type -I, generated by File Management System) and are alphabetized so that the lowest numbered key appears first. The keys each point to their associated data, serving as a fast lookup table for locating data items.

The key can be composed of any type of numeric or ASCII data.

6. File Type

When a file is created, type is defined as keys left justified (shifted), right justified, or non-justified. <u>The selection of correct justification of keys is important</u>.

A. <u>Left Justified Record Key</u> -- The key is assumed to be an ASCII character string and, if necessary, any input string is left shifted, eliminating blanks until the first character is non-blank. In most cases, this is no problem, because a character string normally appears left-justified in its field. This type of record key is usually specified where an alphanumeric ASCII string is to be used as the key.

Example: Assume that the following characters are entered into a 10 character string field:

```
ABCDEFGHIJ
AVDCFƀƀƀƀƀ       (ƀ indicates a blank)
!()9045ƀƀƀ
123ƀƀƀƀƀƀƀ
```

As discussed above, all keys are alphabetized and stored on the disk in a coherent manner. The keys are alphabetized according to their ASCII codes, starting from the <u>leftmost</u> character, then stored with the lowest numbers/alphabetics first. This evaluation is done left-to-right according to the ASCII priority, as illustrated in the table below. (See Appendix B for a complete ASCII code table.) Thus many, but not all, special characters evaluate as less than numbers, which evaluate as less than alphabetics (the /30 code of the zero is less than the /41 code of the A).

<u>Relative Value of ASCII Character Set</u>

```
ƀ ! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [ \ ] ^ <
```

B. <u>Right Justified Record Key</u> -- The key is assumed to be an ASCII character string. If necessary, any input string is shifted to the <u>right</u>, eliminating padded blanks until the right-most character is non-blank.

This type of key is often used where different length character strings of numbers are entered as keys. In comparing numbers, evaluation occurs from right to left, as opposed to ASCII string evaluation which occurs from left to right. Right shifting the keys forces an evaluation as a number, even though the characters are in ASCII.

For example, consider the ASCII strings of numbers:

```
     001
      02
     007
   00200
  000300
```

The above strings are shown as they would be stored when right shifted. If sequentially printed out, they would appear in the order shown.

If you were to use the same character strings as left-justified keys (entering strings with padded blanks), they would be left justified and evaluated from the left according to their ASCII codes, and appear as follows when alphabetized:

```
  000300
     001
   00200
     007
      02
```

Note that the order of the numbers has changed. If you wished to sequentially list these files, they would be printed in the order shown.

C. Non-justified Keys (Binary) -- In this file type, the keys are not shifted, but appear in the field and are evaluated exactly as the programmer carefully places them. This type of key is generally used when binary or hexadecimal data, or a combination of non-ASCII data, is to be used as a record key.

If a file type with a shifted key is used, all characters are assumed to be ASCII, which only uses the lower 7 bits of the byte. Thus, the 8th bit of any byte would be ignored, a definite problem if the data is not meant to be ASCII. For example, if the single byte numeric F1H (1111 0001) is inadvertantly handled as ASCII, it would become 71H (0111 0001).

Note that any unique data can be used in the record keys, but the construction of keys should be carefully considred if the sequential order of the files is important for printouts. The file system includes UP and DOWN instructions which can sequentially read the various items.

7. Index

A single byte is associated with each key in an ISAM file. This index allows you to divide the items in a file into 1 to 8 sub-groups, or sub-levels, with each of the eight bits defining a sublevel, per the following table.

Sub-level	Index Byte Value Decimal	Index Byte Value Hexadecimal
1	1	1
2	2	2
3	4	4
4	8	8
5	16	10
6	32	20
7	64	40
8	128	80

Every item in the ISAM file can be assigned to several index sub-levels, but must belong to at least one. If the index is not important in your program, just set it at 1 in all cases.

The index could be used to divide your file into several sub-files. When an item is searched for in the file, it is identified by key and index, where at least one sub-level of the stored index must match one of the specified search index sub-levels for a successful search. It is not required that all sub-levels match for a successful search.

Example:

Assume that you have an inventory file in which all items are keyed by part number. Using the index you can divide them into groups based on any criteria you wish, such as lead time, pricing level, stocking location, etc.

Lead Time Status	Sub-Level (bit)	Index Decimal Value
Available from stock	1	1
30 days ARO	2	2
60 days ARO	3	4
90 days ARO	4	8
Critical item	8	128

Thus, an index of 8 defines an item with a 90 day leadtime. An index of 136 defines an item with a 90 day leadtime which has also been assigned critical status.

CHAPTER 15. FILE SYSTEM COMMANDS

15.1 GENERAL

This chapter presents the BAL file system commands used with the BAL language. If you are not familiar with the File Management System, you may wish to refer to Chapter 14, which is an introduction to the system. Chapter 14 also contains detailed definitions of various parameters which will be used in the command descriptions in this Chapter.

The file management system is configured as a part of the PROLOGUE Operating System, operates only with sectored devices (disks) and magnetic tape, and obeys all the constrants of the PROLOGUE Operating System regarding volume, filename, etc.

If you attempt to use file system commands with a version of PROLOGUE which was not configured with the proper file system software option, the error message "Module Not Present In The System" will be output.

File System commands are organized in this chapter in the following order:

1. Common instructions -- Used in all file structures.

2. Random File instructions

3. Binary File instructions

4. Sequential File instructions

5. Indexed Sequential Access Method instructions

15.2 COMMON FILE SYSTEM INSTRUCTIONS

Instructions which are common to all file structures are:

ASSIGN	- Assigns a logical number to a file and establishes various file characteristics.
CFILE	- Create a file
OPEN	- Open a file for operations.
DFILE	- Delete (Destroy) a file.

RENAME	- Rename a file
EXTEND	- Extend the disk space assigned to a file.
CLOSE	- Close a file.

15.2.1 ASSIGN

This instruction must be executed in order to access any file on a support device.

Syntax:

```
ASSIGN = N.LOG, Name [,Options][:ERROR]
```

Where:

N.LOG - Logical number from 1 to 15.

Name - A string variable or literal string (enclosed in quotes) which can specify:
- any legal PROLOGUE filename of the form [Device.]Name[-Type][:Keys].
- the unique name of a support device, such as FLØ.

Options -

Code	Definition	Default
WR	Open for Writing	Read only
EX	Open as exclusive file, which cannot be shared by several processors	Sharable file
SQ	Sequential organization	If neither SQ nor SI is specified, the file can be accessed by blocks or sectors
SI	Indexed Sequential organization	

:ERROR - Optional error branch parameter, consisting of two parts, :ADDR, E.
- ADDR is any program line number, which will be branched to if a non-zero status/-error code is returned.
- E is the variable in which the code will be returned.

This instruction performs the following functions:

1. If this instruction references a logical number currently assigned to a file, that file is closed.

2. The syntax of the support name or filename is analyzed.

3. A descriptor is prepared for the file routines, containing the designation of the file and the options chosen.

Note that the Name can be specified as a support device, such as FL0, FL1, etc. In this case, the files <u>cannot</u> be managed by the File Management Sytem instructions and must be handled directly via the IO instruction, with the programmer keeping track of sector assignments, and other housekeeping details. See paragraph 13.3 for complete details on this instruction.

15.2.2 Create A File -- CFILE

<u>Syntax:</u>

```
        CFILE = N.LOG{[,D = record length]
                     { VD
```

<u>Where:</u>

N.LOG	- Logical number from 1 to 15, as assigned by ASSIGN instruction.
D=record length	- Record length in bytes for Sequential files only. Must be a constant or short numeric variable. Default is 256 (one sector).
VD	- Variable record length option for the file.

This instruction performs the following functions:

1. Closes the file if open.

2. Creates and opens the file according to the type declared by the corresponding ASSIGN instruction.

15.2.3 Open A File -- OPEN

<u>Syntax:</u>

```
        OPEN = N.LOG [:ERROR]
```

<u>Where:</u>

N.LOG	- Logical number from 1 to 15.

:ERROR - Optional error branch parameter, consisting of two parts, :ADDR, E.

- ADDR is any program line number, which will be branched to if a non-zero status/-error code is returned.
- E is the variable in which the code will be returned.

This instruction:

1. Closes the file if open.
2. Opens the file according to the mode indicated in the corresponding ASSIGN instruction, i.e., open for read/write, open for read only.

15.2.4 Delete A File -- DFILE

Syntax:

```
DFILE = N.LOG [:ERROR]
```

Where:

N.LOG	- Logical number from 1 to 15.
:ERROR	- Optional error branch parameter, as described above for the OPEN instruction.

This instruction:

1. Closes the file if open.
2. Deletes the file, releasing the disk space.

15.2.5 Rename A File -- RENAME

Syntax:

```
RENAME = N.LOG,NewName [:ERROR]
```

Where:

N.LOG	- Logical number from 1 to 15.
NewName	- New filename, in the standard PROLOGUE format of [Device.]Name[-Type][:Keys]

:ERROR - Optional error branch parameter, as described above for the OPEN instruction.

This instruction:

1. Opens the specified file if closed.

2. Assigns it the specified NewName, deleting the old name from the directory.

NOTE: After this operation, the specifed file remains Open.

15.2.6 Extend A File -- EXTEND

Syntax:

```
EXTEND = N.LOG [,no of sectors][:ERROR]
```

Where:

N.LOG	- Logical number from 1 to 15.
no of sectors	- The number of sectors by which the assigned file space is to be extended.
:ERROR	- Optional error branch parameter, as described above in the OPEN instruction.

This instruction:

1. Opens the file if closed.

2. Assigns the number of extension sectors specified. If the optional number of sectors is omitted, the file is extended one granule by default.

The EXTEND instruction has two basic uses:

1. In Random files it is used to add space to a specific file to allow for the declaration of additional variables.

2. In Sequential and Indexed Sequential, it is used to ensure that sufficient room is available on the volume for the size of file desired. For example, if you wish to create a sequential file containing 300 256-byte records, you might execute an EXTEND instruction to make sure 300 sectors are available. It is better to have the EXTEND fail than to receive an End-of-volume error message somewhere in the middle of writing your file.

When a file is extended, the file system attempts to allocate as much contiguous space as possible. If the volume is not very full, the system first attempts to allocate space at granule 8, if space is not available, then at granule 16 and so forth, jumping in blocks of 8 granules. If the space is available, it will be allocated, even if non-contiguous. However, the file cannot be extended to more than 18 non-contiguous blocks.

Note: The EXTEND instruction is not used often for ISAM files. However, if you use it, plan to allocate 30% more space than your actual data. This is required for keys, indexes and housekeeping.

15.2.7 Close A File -- CLOSE

Syntax:

```
CLOSE = N.LOG [:ERROR]
```

This instruction closes the specified file if open.

15.3 RANDOM FILE INSTRUCTIONS

Random files are a form of sophisticated virtual memory. The user declares his random file on the disk, thereafter refers to the variables in the file as though they were in memory.

Two file instructions are used in Random files: first, FIELD to declare the magnetic peripheral as the location for following variables being declared (via DCL instructions); next, the ASSIGN instruction to assign a logical number and various characteristics to the file before it is first used.

15.3.1 FIELD Instruction

Syntax 1:

```
FIELD = N.LOG [,beginning sector]
```

Where:

N.LOG	- Logical number from 1 to 15. (Logical number 0 is the memory.)
beginning sector	- The starting sector of the random file, where the default is sector 0. Note that this sector is specified relative to the beginning of the random file, allowing equivalencing of variables.

All variables declared following this instruction (until a new FIELD = is encountered) will be a part of the specified random file.

Syntax 2:

```
          FIELD = N.LOG,X
```

Where:

X - Any previously declared variable.

This instruction has the effect of equivalencing variables in the random file. All variables declared following this instruction, will be located beginning at the address of variable X.

15.3.2 ASSIGN - Random File System

Syntax:

```
          ASSIGN = N.LOG, "FileName" [,Options][:ERROR]
```

Where:

N.LOG - Logical number from 1 to 15.

FileName - Any legal PROLOGUE filename of the form [Device.]Name[-Type][:Keys].

Options -

Code	Definition	Default
WR	Open for Writing	Read only
EX	Open as exclusive file, which cannot be shared by several processors	Sharable file

:ERROR - Optional error branch parameter, consisting of two parts, :ADDR, E.
- ADDR is any program line number, which will be branched to if a non-zero status/-error code is returned.
- E is the variable in which the code will be returned.

This instruction must be executed prior to the execution of any instruction which uses one of the variables contained on this random file.

A random file is automatically opened upon the execution of the first instruction utilizing one of the variables in that file. If the file does not yet exist, it is created.

15.3.3 Example of Random File Instructions

The example program writes 256 bytes in the file.

```
     PROGRAM "EXAMP"
     DCL I%
     FIELD = 1                 ;Begin at sector 0
     DCL T#(256)
     SEGMENT 0
     ASSIGN = 1, "TABLE",WR    ;File name TABLE, write OK
     FOR I = 1 TO 256          ;Create and open file here
     T(I) = I
     NEXT I
     ESEG 0
```

By replacing the ASSIGN instruction in the example as follows:

```
     ASSIGN = 1, "FL1",WR
```

the variables will be located on floppy 1, and not subject to file management. They must be controlled through the IO instruction, described in Chapter 13.

15.4 SEQUENTIAL FILE SYSTEM INSTRUCTIONS

The length of records in a sequential file is fixed. Thus, the space required for a file of N records of length L is exactly N*L bytes.

Note that the Sequential file is type SQ, and the type must be specified by an ASSIGN instruction upon each utilization (opening) of a sequential file.

A sequential file can be opened for reading or writing, but not both. If you attempt to write on a file that has been opened for read, an error will be returned (and vice versa).

When a sequential file is opened for Read, the pointer points to the beginning of the file. When it is opened for Write, the pointer points behind the last record in the file.

15.4.1 READ A Sequential Record

Syntax:

```
          READ = N.LOG : [,ERROR,] Input Buffer
```

Where:

N.LOG	- Logical number from 1 to 15, as assigned by ASSIGN instruction.

:ERROR - Optional error branch parameter, consisting of two parts, :ADDR, E.
- ADDR is any program line number, which will be branched to if a non-zero status/-error code is returned.
- E is the variable in which the code will be returned.

Input Buffer - Name of the buffer for input of the record.

This instruction reads the next sequential record following the pointer into the Input Buffer.

The following errors can occur:

1. End of file mark encountered.

2. Length of input buffer is not exactly the length of the data record recorded in the file. The appropriate error code will be returned and no data is read.

15.4.2 BACKSPACE In Sequential File

Syntax:

```
BACKSPACE = N.LOG [:ERROR]
```

Where:

N.LOG	- Logical number from 1 to 15.
:ERROR	- Error branch parameter as described above.

This instruction opens the specified file if closed, then positions the pointer ahead of the preceeding record. The instruction re-positions the pointer only, no data is read.

It is possible to encounter the beginning of file. If so, the appropriate error number is returned.

15.4.3 WRITE A Sequential Record

Syntax:

```
WRITE = N.LOG : [ERROR,] Output Buffer, Length
```

Where:

N.LOG - Logical number from 1 to 15.

:ERROR	- Optional error branch parameter, as described above.
Output Buffer	- Variable containing the record to be output to the sequential file.
Length	- Specifies the length of the output buffer.

This instruction opens the specified file if closed, then writes the record from the output buffer, adding it as the last record in the file.

If the length of the output buffer is not exactly the same length as the standard length of the records in the file, the operation is aborted, no data is written, and the appropriate error number is returned in E.

15.4.4 Example of Sequential File Instructions

This example writes 100 records to the file, then re-reads them.

```
        DCL I%
        SEGMENT 0
        ASSIGN = 1, "FILE", WR,SQ
        CFILE = 1,D=2                ;record length=2 bytes
        FOR I = 1 TO 100
        WRITE = 1 : I
        NEXT I
        ASSIGN = 1, "FILE", SQ       ;close and re-open for read
   10   READ = 1 : I
        PRINT=1: (N32), I
        GOTO 10
        ESEG 0
```

15.4.5 Remarks

1. When a STOP or ESEG 0 instruction is executed, all files presently open are automatically closed by the system.

2. The end-of-file pointer is written when the file is closed.

3. You must close the files before removing the disk containing the file from its support device.

4. The same file can be used simultaneously for read and write.

15.4.6 Sequential Files For Magnetic Tape

1. The labels of these files are the ECMA-13 standard organization.

2. The label HDR2 is processed.

3. The tape blocks can be of variable length.

4. All programs using sequential files on sectored support devices can use the same files on magnetic tape, with these exceptions:

   a. Two sequential files on tape cannot be opened simultaneously.

   b. Only the last file on a magnetic tape can be deleted.

5. The standard mnemonic for magnetic tape is BM.

15.5 INDEXED SEQUENTIAL FILE SYSTEM INSTRUCTIONS

A Sequential file is type SI, and the type must be specified by an ASSIGN instruction upon each utilization (opening) of an indexed sequential file.

Note the following characteristics of an indexed sequential file.

1. The length of the record keys is constant in the same file. This length is defined at the time the file is created, and is 2 to 20 bytes, inclusive.

2. The length of the data associated with each record key is fixed for the same file. This length is also defined when the file is created.

3. When reading data, an input buffer must be specified. The length of this buffer can be specified, and can be longer or shorter than the declared length of the buffer variable. Thus data can be read in starting with a specified variable and overlayed into several variables. For example, A,B,C and D can be declared, in order, as 12 byte strings. A 48 byte indexed sequential data item can be read into buffer A, specified as 48 bytes. Data will thus be available in A, B, C, and D.

   If no buffer length is specified, the declared length of the buffer variable is assumed, by default.

When reading data, if the specified input buffer is too large, the unused portion is filled with ASCII blanks, /20. If the input buffer is too small, it will contain only the beginning portion of the data and an error code will be returned to signal loss of data.

15.5.1 ASSIGN Instruction

This instruction must be executed in order to access an indexed sequential file.

15.5.1.1 Normal Mode

Syntax:

```
        ASSIGN = N.LOG, Name [,Options][:ERROR]
```

Where:

N.LOG - Logical number from 1 to 15.

Name - Any legal PROLOGUE filename of the form [Device.]Name[-Type][:Keys].

Options -

Code	Definition	Default
WR	Open for Writing	Read only
EX	Open as exclusive file, which cannot be shared by several processors	Sharable file
SI	Indexed Sequential organization	

:ERROR - Optional error branch parameter, consisting of two parts, :ADDR, E.
- ADDR is any program line number, which will be branched to if a non-zero status/error code is returned.
- E is the variable in which the code will be returned.

This instruction assigns a logical number to the indexed sequential file and assigns it as a normal mode file. Any time an item is inserted, the table of record keys must be modified to maintain coherence of organization. In normal mode, any sectors that must be modified are saved before any modification is done. (In a very large file, a maximum of 5 or 6 sectors may be modified.) Thus, if a power failure should occur during modification, the old record key organization is preserved and coherence of the file can be restored. This is important, because an incoherant file is unuseable under the file system.

15.5.1.2 Copy Mode

Another mode of indexed sequential operations is available, copy mode. In this mode, the previous status is not saved prior to an insert function, thus operations are speeded up considerably. This mode is generally used only when copying a file, where recovery can be accomplished if a problem occurs.

This mode is specified by an ASSIGN instruction, of syntax:

```
ASSIGN = N.LOG, SI, Options, C
```

The parameters of this instruction are the same as described above. C specifies Copy mode.

15.5.2 Create An Indexed Sequential File -- CFILE

Syntax:

```
CFILE = N.LOG, [Options]
```

Where:

N.LOG - Logical number from 1 to 15.

Options -

Option	Description	Default
D= Data length	Specify length for this file	Length = 0
K= Key length	Length of record key, for binary type file (keys not shifted)	
LK= Key length	Length of record key, for file with keys left shifted	
RK= Key length	Length of record key, for file with keys right shifted	

Notes:

1. If option D is present, it must appear in the command line prior to any other option.

2. One of the key length options <u>must</u> be present, specifying a record key length between 2 and 20, inclusive.

3. The Data elements in any file are of fixed length, as specified in this instruction.

4. This instruction creates a file of record keys with the filename specified in the ASSIGN, plus the suffix I, and another file of data elements with the specified filename and suffix D.

15.5.3 Inserting A New Item -- INSERT

Syntax:

```
INSERT= N.LOG, Key [,Index]:[ERROR,]V[,L]
```

Where:

N.LOG	- Logical number, from 1 to 15.
Key	- Variable containing the record key for the current record. This key must be unique and cannot already exist on the file.
Index	- Optional one byte index. Default = 1.
:ERROR	- Optional error branch parameter, consisting of two parts, :ADDR, E. - ADDR is any program line number, which will be branched to if a non-zero status/-error code is returned. - E is the variable in which the code will be returned.
V	- Name of the variable which begins the output buffer which contains the item to be associated with the record key.
L	- Optional length of the output buffer. If this parameter is omitted, the output buffer length is the declared length of the buffer variable, V.

This instruction adds the key to the Key file, automatically classified in the correct alphanumerical order, and the specified index is attributed to it (or the default index of 1). (See paragraph 13.6 for more details on the Index.) After the insert, the file pointer points to the inserted key. The data is written into the data file.

Note that the Insert instruction cannot be used to modify data, because the specified key must not previously exist in the file (an error is returned if the key already exists.) The MODIF instruction is used to change an existing item.

15.5.4 Read An Indexed Sequential Item -- SEARCH

Syntax:

```
          SEARCH = N.LOG, Key [,Index]:[ERROR,]V[,L]
```

Where:

N.LOG	- Logical number, from 1 to 15.
Key	- Variable containing the record key for the item which is to be read.
Index	- Optional one byte index associated with the specified key. Default = 1.
:ERROR	- Optional error branch parameter, as described above in the INSERT instruction.
V	- Name of the variable which begins the input buffer which is to receive the data item associated with the record key.
L	- Optional length of the input buffer. If this parameter is omitted, the input buffer length is the declared length of the buffer variable, V.

This instruction causes a search for the specified key. If it is found, the associated data item is read into the input buffer. This instruction returns only the data.

If the input buffer is larger than the data item, unused bytes are padded with ASCII blanks. If the buffer is smaller than the data item, the first part of the record is read in and a status/error code is returned indicating loss of data.

If the specified key cannot be found, the appropriate status/error code is returned.

If you are using several Index sub-levels, the specified key may be found, but not with the specified Index. In this case, the appropriate status/error code, but no data, is returned.

After the Search, the file pointer points to the specified key.

15.5.5 Delete An Item -- DELETE

Syntax:

```
          DELETE = N.LOG, Key [,Index][:ERROR]
```

Where:

N.LOG	- Logical number, from 1 to 15.
Key	- Variable containing the record key for the record to be deleted.
Index	- Optional one byte index associated with the specified key. Default = 1.
:ERROR	- Optional error branch parameter, as described above in the INSERT instruction.

The file system locates the specified item and deletes it from the file, if all index sub-levels have been specified. If you have assigned several index sub-levels to the item, but do not specify them all in the DELETE, only the specified Index sub-levels will be deleted. The record key, item and the remaining index bits will remain on the file.

15.5.6 Sequential Read In ISAM File -- UP & DOWN Instr.

After an item is read, the file pointer still points at that item. The UP and DOWN instructions allow you to read the preceding or following item in the file. This allows you to sequentially search through a file, or to sequentially read the items for printing.

Two types of UP/DOWN instructions are available. They are:

Syntax 1:

```
          UP = N.LOG[,Index]:[ERROR,]V[,L]

        DOWN = N.LOG[,Index]:[ERROR,]V[,L]
```

UP points to and reads the key, index, and data of the item preceding the original location of the pointer.

DOWN points to and reads the key, index and data of the item following the original location of the pointer.

Where:

N.LOG	- Logical number, from 1 to 15.

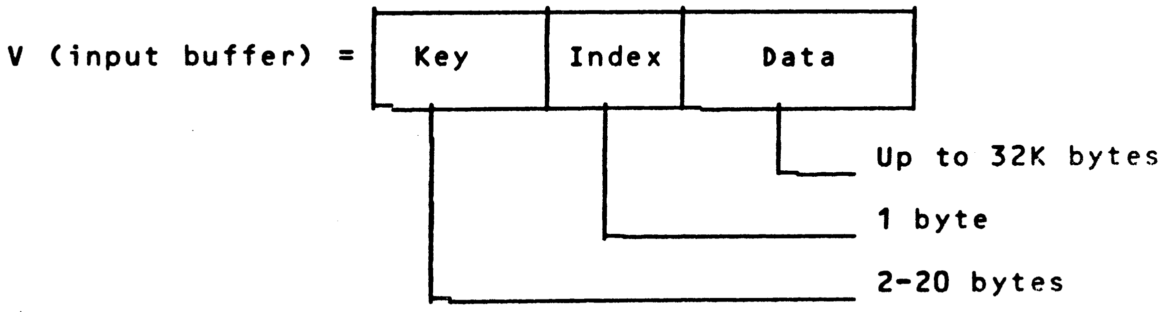
Index - Optional one byte index. This would only be specified when interested in items in a particular index sub-level.

:ERROR - Optional error branch parameter, as described above in the INSERT instruction.

V - Name of the variable which begins the input buffer. Note that this input buffer must be of correct length to receive the key, index, and data of the item to be returned.

L - Optional length of the input buffer. If this parameter is omitted, the input buffer length is the declared length of the buffer variable, V.

These instructions return the values of key, index and data, as illustrated below.



Note that the input buffer length for UP and DOWN is longer than that for SEARCH, where the key and index are known and only the data bytes are returned.

As in the other file instructions, if the optional buffer length parameter is omitted, the length is assumed as the declared length of the buffer variable, V.

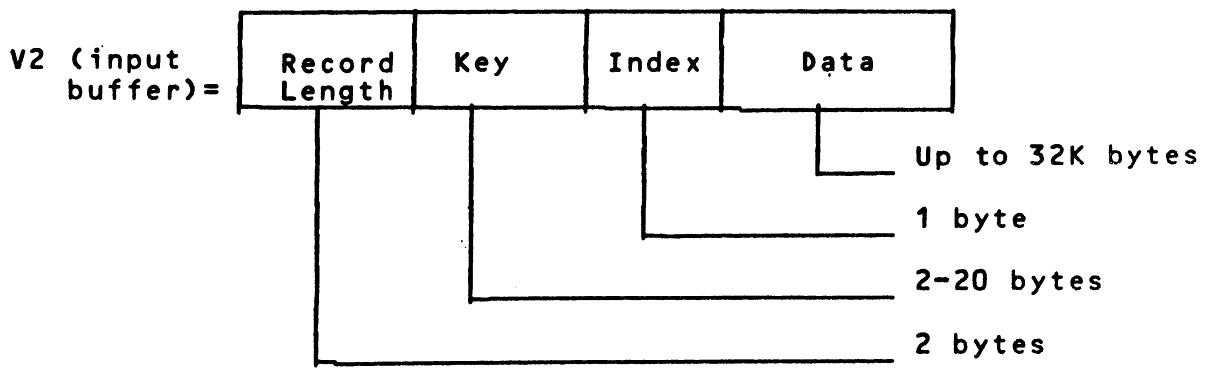
If the input record is smaller than the buffer length, unused bytes are padded with ASCII blanks (/20). If the input record is larger than the specified buffer, the first part of the record is read in, and a status/error message is returned indicating loss of data.

<u>Syntax 2: UP.L & DOWN.L</u>

```
  UP.L = N.LOG[,Index]:[ERROR,]V2[,L]

DOWN.L = N.LOG[,Index]:[ERROR,]V2[,L]
```

Every ISAM record includes two length bytes which record the length of that record. These versions of UP and DOWN return the length bytes, in addition to the key, index and data, as illustrated below. The only difference in the instruction syntax is the name of the instructions and the input buffer, (V2) which must be an addtional 2 bytes in length.



The buffer length considerations discussed for UP and DOWN apply for UP.L and DOWN.L as well.

15.5.7 Modify an Item -- MODIF

Syntax:

```
MODIF = N.LOG, Key [,Index]:[ERROR,]V[,L]
```

Where:

N.LOG	- Logical number, from 1 to 15.
Key	- Variable containing the record key for the record to be modified.
Index	- Optional one byte index, which must match at least one of the index sub-levels of the item. Default = 1.
:ERROR	- Optional error branch parameter, as described above in the INSERT instruction.
V	- Name of the variable which begins the output buffer which contains the modified data and/or index to be associated with the record key.
L	- Optional length of the output buffer. If this parameter is omitted, the output buffer length is the declared length of the buffer variable, V.

This instruction locates the specified record key, then modifies the associated index and/or data. You supply the new index in the optional Index parameter, the new data in the output buffer. Note that the length of the new data item must be the same as the item being replaced. If not, the appropriate status/error code is returned and no change is made to the file.

An existing indexed sequential item can only be changed by the MODIF instruction. You can modify the data and/or the index, but never the record key. To change a record key associated with an existing data record, you must first read the item into memory, then delete it from the file, then insert using the new record key.

After this instruction is executed, the file pointer points to the item which was modified.

15.5.9 Example of indexed file instructions.

This program creates a new indexed sequential file of 25 records and lets the user retrieve them randomly.

```
          PROGRAM "ISAM"
          DCL R%, E#
          DCL L%, K$=20,I#,D$=254
          SEGMENT 0
          ASSIGN=1, "ISAM", WR,SI
          DFILE=1:100,E           ;delete if existing
 100      CFILE=1,D=254,RK=20
          FOR R=1 TO 25           ; write 25 records
          K=CONV(R)               ; key
          D=CONV(R)               ; data
          INSERT=1,K:D
          PRINT=1:R
          NEXT R
 200      ASK=1,I=900:TABV(1),"ENTER KEY"=R;enterESC to exit,
          K=CONV(R)
          K=SHR(K)                ; justified right
          SEARCH=1,K:210,E,D      ; read record
          PRINT=1:TABV(1),"KEY:",K,"INDEX:",1,"DATA:",D
          GOTO 200
 210      IF E<> 78 GOTO 1000
          PRINT =1:TABV(1),"RECORD NOT FOUND"
          GOTO 200
 900      STOP
1000      PRINT=1:"FILE MANAGEMENT ERROR: ", E
          ESEG 0
          END
```

15.5.9 Indexed file instruction (continued)

Example 2

This program shows an example of uses of instructions MODIF, DELETE, VP, DOWN.

```
    PROGRAM "ISAM/2"
    DCL R%,E#
    DCL K#=20,I#,D#=254
    SEGMENT 0
    ASSIGN=2, "ISAM",WR,S:        ; same file previously
                                    created by program "ISAM"
 10 ASK=1,I=900,U=100,D=200 : "ENTER KEY FOR SEARCH OR UP
    ARROW FOR PREVIOUS RECORD OR DOWN ARROW FOR FOLLOWING
    RECORD"
    =R
    K=CONV(R)
    K=SHR(K)
    SEARCH =2,K:90,E,D
    ASK=1,I=900,U=10,01=50 :"K:",K,"CURRENT DATA:",D, "ENTER
    NEW DATA OR CTRL-A TO DELETE RECORD OR UP ARROW TO QUIT
    AND SEARCH FOR ANOTHER KEY OR ESC TO END PROGRAM"=D
    MODIF=2, K,:D
    PRINT =1 : "MODIFICATION DONE-", BELL,TABV(1)
    GOTO 10
 50 DELETE =2,K:D
    PRINT=1:"RECORD", K, "DELETED",BELL,TABV(1)
    GOTO 10
 90 IF E<>78 GOTO 1000
    PRINT=1:"RECORD NOT FOUND"
    GOTO 10
100 UP=2:110,E,K,275  ; input buffer includes key, index, data
    PRINT =1:"KEY:",K, "DATA:, D, TABV(1)
    GOTO 10
110 IF E <>64 GOTO 1000 ; IF NOT BEGINNING OF FILE: ABNORMAL
    PRINT=1:"BEGINNING OF FILE", BELL
    GOTO 10
200 DOWN=2:210,E,K,275
    PRINT=1:"KEY:",K,"DATA:,D,TABV(1)
    GOTO 10
210 IF E<>48 GOTO 1000 ; if not end of file : abnormal
    PRINT=1 "END OF FILE", BELL,BELL
    GOTO 10
900 PRINT =1/"NORMAL END"
    CLOSE=2
    STOP
1000 PRINT =1: "FATAL FMS ERROR :",E
    STOP
    ESEG
    END
```

CHAPTER 16. BAL TRANSLATOR & EXECUTOR

16.1 GENERAL

BAL is a compiler language which requires a BAL source file as input to a translator module. The translator (named TR) translates this source file and produces an intermediate file. This file can only be loaded and executed under control of the BAL Executor module (named EX).

This source file can have any valid filename and is written in the BAL format as described in this manual, using the PRO-LOGUE Editor (ED), described in detail in the BOSS PROLOGUE User's Reference Manual, publication number, B-1003. The source file is assigned implicit type -S if the type is not specified.

16.2 BAL TRANSLATOR - TR

When a BAL program has been prepared in the proper format using the PROLOGUE editor, it must be translated by the BAL Translator Routine, which is cataloged on a BAL disk as TR-0. Use the Translate command as follows (example given using our sample program, BALDEMO above):

Command Syntax:

```
->[Device.]TR,Filename[,Options]
```

Where:

[Device.]TR	- Specifies the Translator program.
Filename	- Specifies a BAL source file. The filename must be in the correct PROLOGUE format as: [Device.]Name [-Type][:Keys]. If Type is omitted, type -S is assumed.
Options	- Several compilation options can be specified, in any order, separated by commas. They are:

Option	Definition	Default Case
NL	No Listing	Listing on CRT Display if no list option specified.
LIS=LO	Listing on line printer	

ND	No debug -- Debug addresses are not output on the program listing	Debug addresses supplied
TP	Partial translation	Complete translation
DEST=Filename	Specifies name of resulting intermediate file. Type -T is the default case.	Intermediate file is assigned the same name as the source file with file type -T

This command results in the translation of the specified source file with the indicated options. Note the following:

1. If ND is omitted, the program listing is produced with debug addresses calculated and listed to the left of each program line. These are the actual memory addresses of the code for that instruction. These addresses are necessary when using the Debug option in correcting the program. See paragraph 16.4 for complete details on DEBUG.

2. If TP, partial translation, is specified you can select certain segments of your program to be translated. The system prints: ....SEGMENT NUMBER:. You must enter a segment number followed by a carriage return. The next segment number is then requested. When you respond with a carriage return instead of a number, the selected segments are translated.

   This option is useful if several segments of your program are working correctly and you wish only to translate one or more segments in which errors have been corrected. Note that if segment 0 changes, the entire program must be translated.

3. When DEST=Filename is specified, you can assign any valid filename to your intermediate BAL file.

When program errors are detected during translation, they are handled as follows:

1. If the listing is on the CRT, an error message is displayed and output halts, giving you time to note the error. The offending character is enclosed in parentheses and an error message is listed. See Appendix C. for complete list of error messages and descriptions. Pressing ESC continues the translations.

2. If the listing is on the printer, an error message is printed below the instruction in error and output continues.

```
Example:  DCL A,B,CD
          ***(D) DECLARATION ERROR  DBUG ADDRESS 0007
```

When a translation time error is fatal you will note that a program length of zero is listed at the end of the translation for the segment in which that error occured.

Examples:

1. TR,FL1.BALDEMO	Translator program loaded from System support device (FL0 in this case), BAL source file BALDEMO-S (S assumed as default case) loaded from floppy 1 and translated; intermediate file generated on User support device (FL1 in our example) and named BALDEMO-T by default; listing output on CRT display.

Note: To halt the translation, press the ESC (Escape) key. To return to PROLOGUE after ESC, press the R key.

16.3 BAL EXECUTOR

The BAL Executor (cataloged on the diskette as EX-0), is the run-time package used to execute translated BAL programs. Use the Execute command as follows.

Command Syntax:

```
->[Device.]EX,IntFilename[,DB]
```

Where:

[Device.]EX	- Specifies BAL Executor package.
IntFilename	- Specifies intermediate BAL file, produced by TR routine. Can be any valid filename. Type -T is assumed as the default case.
[DB]	- Specifies execution of the program under DEBUG control. See the BAL Reference Manual for a complete description of use of the DEBUG package.

When this command is executed, the BAL Executor is loaded into memory, then the specified BAL program is loaded into memory and executed. If any run-time errors occur, an error message is displayed (see the BOSS BAL Reference Manual for list of error messages), the BAL program is aborted and the PROLOGUE prompt is returned.

An error message will be in the format:

```
ERROR N IN SEGMENT X AT ADDRESS YY
```

Where :   N is the error number.
           X is the program segment.
           YY is the Debug address within the segment.

APPENDIX A. BIBLIOGRAPHY

Refer to the following documents for additional detail on topics discussed in this manual.

Title	Publication No.
BOSS System Operator's Manual	B-1001
BOSS PROLOGUE User's Reference Manual	B-1003
BOSS PROLOGUE System Programmer's Guide	B-1004
BOSS BASIC Reference Manual	B-1005

APPENDIX B. ASCII CODES

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
000	00	NUL	043	2B	+	086	56	V
001	01	SOH	044	2C	,	087	57	W
002	02	STX	045	2D	-	088	58	X
003	03	ETX	046	2E	.	089	59	Y
004	04	EOT	047	2F	/	090	5A	Z
005	05	ENQ	048	30	0	091	5B	[
006	06	ACK	049	31	1	092	5C	\
007	07	BEL	050	32	2	093	5D	]
008	08	BS	051	33	3	094	5E	^
009	09	HT	052	34	4	095	5F	<
010	0A	LF	053	35	5	096	60	'
011	0B	VT	054	36	6	097	61	a
012	0C	FF	055	37	7	098	62	b
013	0D	CR	056	38	8	099	63	c
014	0E	SO	057	39	9	100	64	d
015	0F	SI	058	3A	:	101	65	e
016	10	DLE	059	3B	;	102	66	f
017	11	DC1	060	3C	<	103	67	g
018	12	DC2	061	3D	=	104	68	h
019	13	DC3	062	3E	>	105	69	i
020	14	DC4	063	3F	?	106	6A	j
021	15	NAK	064	40	@	107	6B	k
022	16	SYN	065	41	A	108	6C	l
023	17	ETB	066	42	B	109	6D	m
024	18	CAN	067	43	C	110	6E	n
025	19	EM	068	44	D	111	6F	o
026	1A	SUB	069	45	E	112	70	p
027	1B	ESCAPE	070	46	F	113	71	q
028	1C	FS	071	47	G	114	72	r
029	1D	GS	072	48	H	115	73	s
030	1E	RS	073	49	I	116	74	t
031	1F	US	074	4A	J	117	75	u
032	20	SPACE	075	4B	K	118	76	v
033	21	!	076	4C	L	119	77	w
034	22	"	077	4D	M	120	78	x
035	23	#	078	4E	N	121	79	y
036	24	$	079	4F	O	122	7A	z
037	25	%	080	50	P	123	7B	{
038	26	&	081	51	Q	124	7C	¦
039	27	'	082	52	R	125	7D	}
040	28	(	083	53	S	126	7E	~
041	29	)	084	54	T	127	7F	DEL
042	2A	*	085	55	U			

ASCII is an acronym for American Standard Code for Information Exchange.

Standard Abbreviations for ASCII Characters 0 through 31
(00 through 1F Hex)

ACK	Acknowledge
BELL	Bell
BS	Backspace
CAN	Cancel
CR	Carriage Return
DC1	Direct Control 1
DC2	Direct Control 2
DC3	Direct Control 3
DC4	Direct Control 4
DLE	Data Link Escape
EM	End of Medium
ENQ	Enquiry
EOT	End of Transmission
ESC	Escape
ETB	End Transmission Block
ETX	End Text
FF	Form Feed
FS	Form Separator
GS	Group Separator
HT	Horizontal Tab
LF	Line Feed
NAK	Negative Acknowledge
NUL	Null
RS	Record Separator
SI	Shift In
SO	Shift Out
SOH	Start of Heading
STX	Start Text
SUB	Substitute
SYN	Synchronous Idle
US	Unit Separator
VT	Vertical Tab

APPENDIX C. BAL AND FILE MANAGEMENT SYSTEM ERROR CODES

C.1 GENERAL

When you are using the BAL system, errors may occur during translation of your program or execution of the program. This appendix contains a complete list of the various errors which may occur when using the BAL System and the optional File Management System. They are grouped according to Translation-Time errors, Execution Time errors, and File Management System status/errors.

Note that errors may occur when using various PROLOGUE Operating system routines. This includes errors which may occur using the peripherals. Refer to the PROLOGUE User's Manual for a complete list of these error codes, and suggested recovery procedures.

C.2 TRANSLATION-TIME ERRORS

When program errors are detected during translation, they are handled as follows:

1. If the listing is on the CRT, an error message is displayed and output halts, giving you time to note the error. The offending character is enclosed in parentheses and one of the error messages listed below is displayed.

   Pressing ESC continues the translation.

2. If the listing is on the printer, an error message is printed below the instruction in error and output continues.

3. Example:
```
DCL A,B,CD
***(D) DECLARATION ERROR  DBUG ADDRESS 0007
```

4. When a translation time error is fatal, you will note that a program length of zero is listed at the end of the translation for the segment in which that error occured.

Message Displayed	Remarks
Keyword Incorrect	1. Attempt to begin a program with a statement other than PROGRAM. 2. Spelling error. PRINT spelled PRIMT; GOTO written GO TO or with zeros, not letter O; keyword typed in lower case (must be all caps), etc.

	3. The equal sign that is part of several keywords (ASK=, PRINT=, READ=) may be misused, i.e., ASK = (no space allowed).
Ettiquete Error	Error in use of instruction: GOTO READ, IF A=B GOTO C (cannot GOTO A variable).
Value Not Binary	Attempt to use alphanumeric when binary required, e.g. SEGMENT = typed instead of SEGMENT 0.
Segment Number Incorrect	Attempt to use segment number larger than 15, use of incorrect segment number with ESEG.
End of Instruction Incorrect	Last portion of instruction can't be decoded. Often occurs when the ; is used to add comments to a line and is misspelled or edited out. Also can be a typing error (such as for input variable in ASK instruction).
FOR-NEXT Incorrect	Often an attempt to use FOR without a corresponding NEXT.
Syntax Error	Instruction constructed incorrectly. Incorrect operand, spelling error, error in required punctuation, e.g., must be ASK=1: not ASK=1;.
Incorrect Operator	One of operators (+,-,*,/) omitted or used incorrectly.
Variable Type Incorrect	Incorrect variable type used. Ex: numeric variable used in instruction to manipulate a string; string used as index in FOR-NEXT loop.
Format Error	Incorrect ASK or PRINT format.
Support Variable Incorrect	Variable used incorrectly in FIELD statement or in equivalencing variables.
Declaration Error	Error in DCL instruction. Often a typing error, such as DCL A,B,CD. Could be an attempt to declare two variables with the same name.
Binary Code Too Large	Object code generated for this instruction (usually a complicated ASK or PRINT) is too large. Find a way to shorten the instruction.

String Incorrect	Error in string construction. May have forgotten to indicate string by quote marks, as "STRING"; or tried to exceed the specified string length.
BCD Incorrect	Can occur if you use construction A=B**C (legal in some systems).
Stack Overflow Nested FOR Loops	Must reduce number of nested FOR-NEXT loops to the mximum of 16.

C.3 EXECUTION TIME ERRORS

The following errors can occur when you execute your BAL program. When one of these errors occurs, an error message is printed or displayed in the format below, the program is aborted, and you return to the PROLOGUE command level.

```
ERROR NN IN SEGMENT XX AT DEBUG ADDRESS YYYY
```

Where: NN is the error code as listed below.
XX is the program segment being executed.
YYYY is the debug address of the instruction which was in error.

Note: Error codes are in decimal.

Code	Description	Remarks
100	Segment non-existent	Attempt to reference a non-existent segment.
101	End of DATA for READ instruction	Need to add DATA elements or use RESTORE instruction.
102	Zero index in a table	Zero index, e.g., B(0) illegal.
103	Index too large	Attempt to use an index value larger than the maximum declared value.
104	String used as index	Illegal
105	Incorrect return in a GOSUB	
106	Overflow of long numeric variable	Overflow of capacity of long variable.
107	Too many levels of GOSUB or LDGO-SEG	Sixteen levels maximum.

Code	Description	Remarks
108	Arithmetic overflow	
109	Intermediate file incorrect	Error in structure of BAL intermediate file. Re-translate.
110	Long numeric variable incorrect	
111	Wrong peripheral no in ASK or PRINT	ASK=1 is only legal number. PRINT=1 or PRINT=2 OK, all others incorrect. Check your variable PRINT=A:..; for correct value.
112	Format error	
113	Variable too large	Variable too large to be contained in specified memory.
114	Memory overflow	Revise your program into several smaller segments which can be called into memory as needed.
115	Instruction unknown	Could have clobbered part of memory.
116	Logic No. not declared by ASSIGN	Attempt to use an instruction referencing a logical number before an ASSIGN has been executed, assigning a number to the specified file.
117	Logical number referring to file, not a support device.	In IO instruction, attempt to refer to the logical number of a file. Can only refer to a peripheral device.
118	External variable forbidden at this place in this instruction	
119	Logical number referring to support device, not file	In instruction which uses a file, have referred to the logical number assigned to a peripheral device. This is used only by IO instruction.
120	File or support device write protected.	For a file, you must supply the correct password for write, for device, remove write protection tab.

Code	Description	Remarks
121	Name furnished is support device name not file name.	
122	Input Buffer too small	In file management, the specified input buffer is too small to contain the the length or index of the instruction Searched for.

C.4 FILE MANAGEMENT SYSTEM STATUS/ERROR CODES

Each time you use one of the File Management System instructions, a status/error code will be returned (in parameter E, if specified). This code either indicates a normal status condition of the file, or indicates that an error occurred. The codes and descriptions are listed below.

Code (decimal)	Status/Error Condition
00	Normal execution of command
40	File unknown
41	File already exists
42	File closed
43	File already open
44	File cannot be shared
45	Attempt to extend file beyond 18 extension blocks
46	Volume overflow
47	Incorrect password (part of filename)
48	End of file
49	Directory full
50	Logical number too large (15 maximum)
51	Logic number table full
52	File not opened by you
53	Sector address unknown
54	No directory (volume not created by file system)
55	Requested file function unknown
56	Function does not exist in system (may not be configured into this version of PROLOGUE)
57	Support non-shareable
58	Volume still contains files
59	Incoherence in the structure of the volume
60	File type incorrect (Example, attempt to INSERT in a sequential file.)
61	Record length incorrect
62	Loss of information on write
63	Loss of information on read

Code (decimal)	Status/Error Condition
64	Beginning of file
70	Key length too small
71	Key length too large
72	Too many files open
73	Index value of zero
74	Key already blocked
75	Overflow of file of keys
76	Incoherence in the file of keys
77	Overflow of data file
78	Key does not exist
79	Key exists but not with specified index
80	Overflow of the structure of keys. No more room for keys.
81	Key already exists
82	Input buffer smaller than the key