The BASIC Answer

LOGICAL SYSTEMS INC.

11520 N. Port Washington Rd., Mequon, WI 53092
(414) 241-3066

The BASIC Answer

Table of Contents - - User Guide Section

Copyright (C) 1982 by Logical Systems, Incorporated
11520 N. Port Washington Rd. Mequon, WI 53092
(414) 241-3066

T h e B A S I C A n s w e r

Table of Contents - - Tutorial Section

The BASIC Answer - Introduction

The BASIC Answer is a processing utility designed to allow the creation of meaningful and structured BASIC programs. The BASIC Answer (from now on referred to as TBA) is only for use with the LDOS operating system created by Logical Systems, Inc. Programming with TBA will be similar to programming in a Microsoft-compatible interpretive BASIC. All of the commands that are available in interpretive BASIC will be usable with TBA. However, TBA incorporates many concepts not found in interpretive BASIC which will make program code more descriptive and structured. This becomes especially useful when debugging or modifying programs that have not been examined for long periods of time.

This user guide will contain a brief explanation of syntax and definitions of statements needed to utilize the TBA processor. All detailed explanations are contained in the accompanying tutorial manual. The tutorial manual also contains examples of processing and exercises to aid the beginner, an explanation of what the processor does, concepts of program structure, rationale for syntax and ideas for applications of some processor concepts.

Source Code Creation

Source Code can be created either through use of text processors such as LSCRIPT, SCRIPSIT, SUPERSCRIPSIT, or LED, or through the BASIC interpreter. Source lines have a maximum allowable length of 240 characters. If a text processor is utilized, the output file must be saved in a pure ASCII format. If BASIC is utilized, SAVE the source file by using the ASCII parameter ",A" after the filespec.

Upper and Lower Case Usage in Source Code

TBA utilizes certain technical phrases in lieu of the ordinary BASIC equivalents. Among these are LABELS, VARIABLES, and EXPRESSIONS all of which are explained in separate sections. Each of these have some things in common, one of which, is the use of upper and lower case.

Lower to upper case conversion will normally take place during processing exactly as it is done in a BASIC interpreter. Any lower case characters outside of quotes or REMark statements will normally be replaced with upper case equivalents. The TBA processor does provide a means of overriding this procedure. In that case, the technical phrases described will be unique if they are comprised of the same letters but with unmatched cases. Normally, the word LOOP, Loop, loop and LOop would be processed as the same word. If the Differentiate Case switch has been specified, those same words would be four unique labels, variables, or expressions.

Labels - How to use Labels

Labels are used to reference locations in source code instead of absolute line numbers. Source code need not be written with line numbers in front of statements unless it is actually created in BASIC. To branch in source code use a label.

Labels appear in two ways. First, as the procedure identifier so that execution will begin at the label. The label in this case must be the first phrase on the line. It could then be followed by a colon with BASIC statements contained on the same line,

```
@DELAY.LOOP : FOR LOOP% = 1 TO 2000: NEXT
          RETURN
```

or it could stand alone.

```
@DELAY.LOOP
          FOR LOOP% = 1 TO 2000: NEXT
          RETURN
```

Second, labels are used to indicate a branch to a procedure. In this case the label appears after a BASIC branch statement.

```
PRINT" Your chocolate pudding is infected": GOSUB  @DELAY.LOOP: CLS

IF INKEY$ = CHR$(13) THEN GOTO  @PROCESS.INPUT ELSE GOSUB @GET.KEYBOARD
```

Labels - Syntax and Allowable Characters

A label must have an "at" symbol <@> as its first character.

The second character must be ALPHABETIC in either upper or lower case.

There can follow up to thirteen more ALPHANUMERIC upper/lower case characters for a total label length of 15 characters.

Two SPECIAL characters can be used in lieu of one or more of the last thirteen characters. These are the period <.> and underline <_> characters.

NO OTHER special characters or spaces are allowed.

Examples of LEGAL labels :

```
@Input              @ Process.input      @RETRY_INPUT
@ get.a.record       @ playitagainSAM     @ get_the_money
@ Etc.etc.etc...     @A...number1        @ Print_period.
```

ILLEGAL labels include :

```
@1.for.the.money    : leading number; too long
@GET NAME           : space in label
@... delay          : leading special character
@LINEINPUT#1        : unauthorized character
```

Labels must be followed by a space, a colon, or a carriage return. Labels used to identify the entry point to a procedure must be the first element of the line. Ordinary BASIC statements can follow a label if separated by a colon. If a label defines the entry point of a procedure, then LOCAL variable definitions may follow the label in lieu of BASIC statements.

Variables - Syntax and Allowable Characters

TBA allows two variable implementations, Global and (pseudo) Local. The unique aspects of these will be delineated in separate sections. The similarities are discussed below.

TBA variables must be at least three total characters in length (including tag).

The first character of a variable must be ALPHABETIC in either upper or lower case.

At least one, but up to thirteen characters, following may be upper/lower case ALPHANUMERIC.

Two SPECIAL characters are allowed. The period <.> and/or underline <_> characters may be used in lieu of any of the second through fourteenth characters.

The last character MUST be one of the four BASIC variable declaration tags. These are <%> for an integer, <!> for a single precision, <#> for a double precision, or <$> for a string variable.

NO OTHER special characters or spaces may be used.

Certain RESERVED WORD conflicts MAY NOT be used. These are:

Two character BASIC keywords (FN, ON, OR, AS, IF, TO)
BASIC keywords ending with a declaration tag (TIME$, WD$, PRINT# etc.)
Any word in which the first three letters are : <u>REM</u>

Reserved words may be <u>embedded</u> in a variable.

Examples of <u>LEGAL</u> Variables :

Record.Number%	Total.due#	LAST_NAME$
LOOP%	LOOP1%	LOOP_2%
Money.owed!	TAX#	Start.time$
IF.done%	order#	X.Y_FUNCTION

<u>ILLEGAL</u> variables include :

TO%	: reserved word violation
FLAG END%	: no spaces allowed
end.of.sequence%	: too long
1st.record%	: incorrect lead character
input#	: reserved word violation

Variables are used in the same manner as they are in ordinary BASIC.

Variables Must be surrounded by spaces or non-alphanumeric characters such as parentheses, commas, math and relational operators to prevent misinterpretation.

Single character variables are allowed but are not processed or cross referenced by TBA. They are totally the responsibility of the operator.

<u>Variables - Global Variable Definitions and Implementation</u>

Global variables are those variables which will be utilized throughout the BASIC program. They are the most commonly used and are the only type available in interpretive BASIC. In TBA they must be DEFINED in source code.

To define Global variables, type an equal sign <=> as the first character of a line followed by the variable name. End the definition line with a carriage return. To define more than one Global variable, put them on the same line but separated by commas. Global definition lines may appear anywhere in the Source Code. NOTHING else may be on a Global definition line. An example of a proper line is

```
=LOOP%,LOOPI%,TOTAL#,START.TIME$
```

<u>Variables - Local Variable Definition and Implementation</u>

A Local variable is one that occurs only in a subroutine (procedure). Its value or purpose is only relative to that procedure and upon leaving the procedure the value or purpose is no longer of use. Therefore, Local variables cannot be used to pass information to and from the main body of a program.

A Local variable can ONLY be defined after a procedure label. After typing the label, type an equal sign <=> followed by the variable name. End the line with a carriage return. To define more than one Local variable, separate the variable names with a comma. Only ONE LINE of Local variable definitions is allowed per procedure. The following line is an example of the proper syntax :

```
@INKEY.INPUT=LOOP%,INK$,AT%,FIELD.LEN%
```

Variables - Array Variables

Array variables are to be defined as either Global or Local by utilizing the array name in a definition line. After defining, simply use DIM to declare the array size. The array name must conform to all variable name requirements.

It is not recommended that array dimensions larger than 10 be defined in a Local procedure. This means that no DIM statement is necessary and a Redimensioned Array error can be avoided.

It is possible to have an array name be a duplicate of a simple variable name. In most BASICS, the two variables LP% and LP%(xx) are two different variables.

Special Use of the BASIC REM statement

TBA processes the BASIC REM statement and its abbreviation <'> in two different ways. If the keyword REM appears as the first element of a line, the processor will pass the line unmodified to the object code. If REM appears later in a line it will NOT be recognized properly and may generate an error.

The abbreviation <'> will cause the processor to DELETE the line or remainder of a line from the object code. Unlike, REM, the <'> may appear anywhere in a line.

To pass a remark statement to BASIC use REM at the front of a line. To utilize the remark only in Source, use <'>.

Special Use of the BASIC Statement - RETURN

RETURN is used by the processor to signify the end of a subroutine. The RETURN statement can be used ONLY ONCE per subroutine. Furthermore, it must occur as the last physical statement in subroutine.

The RETURN must be the only BASIC statement in the line. However, a label may precede the RETURN statement, as in : **@EXIT.INPUT : RETURN**

Directives - Processor Output Modification and Identification

There are four Directives which can be embedded in Source to either modify the output of the processor or to identify it.

All Directives and their parameters (if any) must be the only elements of a Source line.

***PRLINES** is used to specify how many lines per page are to be printed. An X'0C' will be sent to the output device after the number of lines specified have been printed.

The syntax is ***PRLINES=N** where n is a number from 20 through 254.

The default value for *PRLINES is 56.

Proper pagation will occur automatically on 132 column printers. To achieve proper results on narrower printers, utilize the printer filter (PR/FLT) available on standard LDOS. Set the CHARS parameter to the proper width.

***LIST ON/OFF** is used to toggle the process listing on and off. The proper syntax is either ***LIST ON** or ***LIST OFF**. The default condition is on.

***PAGE** is used to send a top of form character (X'0C') to the lineprinter whenever *PAGE is encountered on the Source.

***TITLE** is used to print a header at the top of a page whenever a top of form has been issued. The proper syntax is ***TITLE "string"** where string is a group of up to fourteen characters. The header will be of the form :

BASIC Answer "string" September 21, 1982 12:03 A.M. Page 1

*TITLE is used ONCE anywhere in Source to toggle on the titles. The default is OFF for this Directive.

Directives - Conditional Processing of the Source Code

There are three Directives used to process Source on a conditional basis. All are interrelated in use. *IF alerts the processor that a CONDITION must be true or the subsequent code is to be ignored.

The *IF is followed on the same line, but separated by a space, from an EXPRESSION. The EXPRESSION is a phrase which must conform to the same rules as applied to variables except that there is NO declaration tag. The proper syntax is :

***IF expression.**

*END is used separately on a subsequent line to signify the end of the CONDITIONAL block opened by a previous *IF expression statement.

The *IF expression ... *END sequence cannot be embedded in another CONDITIONAL sequence.

In order for the CONDITIONAL sequence to be processed two different methods may be employed.

The first method is to place the EXPRESSION in the Source PRIOR to the CONDITIONAL block. To do this, type the SAME expression which occurs after the *IF expression statement, but precede it with an asterisk <*>. The proper syntax is : *EXPRESSION where EXPRESSION is a phrase syntactically identical to a variable without a declaration tag. The CONDITIONAL block *IF expression code, code ... *END containing the same phrase as *EXPRESSION will then be processed.

The second method is to enter the expression phrase during processing in answer to the "Directives" prompt.

How to Operate the Processor

To enter TBA type "TBA" at the LDOS Ready prompt.

Pressing the <BREAK> key any time during processing or in response to any input prompt will cause the processor to abort and return to LDOS Ready.

Answer the prompt "Source File" with the name of the ASCII file to be processed. TBA assumes a default extension of "/TBA".

Answer the prompt "Object File" with any legal filename or filespec. Pressing <ENTER> will default to the **"source filename/BAS"** .

Processing Parameters

Answer the "Processing Parms" prompt with <ENTER> or one or a list of parameters. Separate the list with commas.

The DEFAULT when pressing <ENTER> is screen processing.

- LP - - processing will be sent to the line printer instead of the screen.
- TO - - only the object code and NO Source will be displayed.
- NL - - NO processed code will be displayed.
- NX - - NO cross reference will be displayed.
- FC - - all extraneous spaces will be removed from the object code.
- DC - - NO conversion from lower to upper case in variables, labels, or expressions will occur.

Directive Prompt

Answer this prompt by pressing <ENTER> to bypass.

OR

Enter one or more EXPRESSIONS matching the expressions contained in *IF expression statements. Separate multiple entries with commas. The prompt will repeat. If more than one line of EXPRESSIONS is necessary, enter them on subsequent repetition of the prompt.

The Directives prompt will continue to repeat until only the <ENTER> key is entered as the first character of the input line.

TBA will now read and process the source file. Current pass information will be displayed on video. After pass five the normal output will be a line of Source followed by "====>>" and the object code. After this occurs to the end of the Source, a Cross Reference Table is generated.

If some error is encountered during processing the following events will occur:

Disk or hardware errors are handled just as they are by LDOS.

Erroneous response to an input prompt will cause the prompt to repeat.

Errors in Source will cause the processor to display the error, the code at which the error occurred, and then PAUSE until any key is depressed.

For a detailed list and explanation of the Error messages, refer to the tutorial manual starting at page 41.

The BASIC Answer - Introduction

The BASIC Answer is a processing utility designed to allow the creation of meaningful and structured BASIC programs. The BASIC Answer is only for use with the LDOS operating system created by Logical Systems, Inc. Programming with The BASIC Answer (from now on called TBA) will be similar to programming in a Microsoft-compatible interpretive BASIC. All of the commands that are available in interpretive BASIC will be usable with TBA. However, TBA incorporates many concepts not found in interpretive BASIC which will make program code more descriptive and structured. This becomes especially useful when debugging or modifying programs that have not been examined for long periods of time.

The user's manual will contain different samples of written BASIC code capable of being processed by TBA. It will also show the program code generated.

TBA - What it is and what it does

TBA is a processing utility, rather than a BASIC compiler. The difference between a compiler and TBA is that a compiler will generate a "CMD" (command) type file. TBA will generate an interpretive program similar to programs that would normally be created in BASIC. It is a compiler-type utility because it does process non-executable program text into an executable BASIC program.

TBA functions in the following manner:

1.) Program text is developed and saved to a file in ASCII format. This non-executable program text will be referred to as SOURCE CODE (or just SOURCE). Source code may be written using a word processor, text editor or written directly in BASIC. Using a word processor or text editor is the best way to create source code since editing can be done very easily.

2.) TBA is used to process the source code. The result of this processing is an ASCII file which is a BASIC program. This processed executable program will be referred to as OBJECT CODE (or just OBJECT).

3.) After the processing has been successfully completed, the resulting object file can be run as a normal BASIC program.

This explanation has given a brief idea of how TBA is used and the logical flow of operations needed to create programs. The remainder of the documentation will be divided into 5 sections. The material covered in these sections will be grouped as follows:

Section I	- Differences between writing normal BASIC programs and writing source code.
Section II	- Writing Source Code.
Section III	- Using TBA to Process files.
Section IV	- Technical Information - How the Processor Operates
Section V	- General Operational Guidelines & Program Maintenance

Section I - Differences between writing BASIC Programs and writing source code

Writing source code to be processed by TBA will be somewhat like writing normal BASIC programs. All of the BASIC program commands which are available in normal BASIC programming will be allowed when writing source code. However, since the source code has to be processed by TBA, there are some guidelines that need to be followed when writing it. This section of the manual will detail the differences between writing normal BASIC programs and writing source code.

LABELS, NOT LINE NUMBERS

The first aspect of writing source code is that LABELS are used in lieu of line numbers. A label can be thought of as a phrase which denotes a point of transfer for program execution or as a reference point in a program. This is similar to a line number in an ordinary BASIC program. When writing source code, line numbers MUST NEVER be used within the program code. BASIC statements (such as GOTO 100, GOSUB 6000, RESUME 820, etc.) which require a line number following the command, should be written in the source code with a label following the statement. In turn, the program line to be referenced will be identified by a label.

A label consists of the <@> character, followed by a label name. The label name can be from two to fourteen alphanumeric characters in length and the first character in a label name must be alphabetic. A period or underline character <.> or <_> may also be used in a label name.

The following are examples of valid labels:

```
     @field.buffer1              @ strobe.kbd            @ Process_data
     @check.file                 @CHECK.FILE            @  back_to_LDOS
     @Get.A.Record               @ROUTE.TAKEN001        @  find_PRIME
```

The following are examples of invalid labels. The part of the label which makes it invalid will be indicated.

@1.character	: leading character not alpha
#field.buffer	: first character not @
@check/file	: incorrect special character
@input char	: no spaces allowed
@field@buff	: @ sign within label
@point-taken	: incorrect special character

Note from the examples of valid labels that the first character of a label name must be alphabetic. To make labels more readable, a period and/or underline character <.> or <_> may be used to break up words within the labels. It must be emphasized that spaces and other special symbols are not allowed in a label name.

The processing function will normally convert lower case characters to upper case characters except in REM statements or within quotes. However, the processor provides a switch to prevent this from happening. That is, a label specified as upper case could represent a different label than the same label name entered in lower case. If differentiate case is on, then the labels @CHECK.FILE and @ check.file would be two distinct labels and would be referenced as such. Normally, however, @CHECK.FILE and @check.file would represent the SAME label.

The decision to incorporate cases as different, should be made prior to writing Source code. Proper care should be exercised in either case because when a label is referenced and no conversion is done, the label must be exactly as it was defined (in essence, a character for character match). On the other hand, if conversion will take place, the variable names must incorporate different arrangements of characters in order to be unique.

Like line numbers, a label may only be defined once within the source code. Keep in mind that labels are analogous to line numbers. It is not allowed to have a label defined more than once in the source (just as there cannot be two line number 100's in an ordinary BASIC program). Although a label cannot be defined more than once, it can be referenced as often as necessary (e.g. in an ordinary BASIC program, there may be only one line numbered 100, however, as many GOTO 100 statements as required).

Defining and Using Labels

In order to define a label, type it at the beginning of a line (exactly as a line number in ordinary BASIC). When defining a label, it must appear first in the text of the line. That is, a label definition cannot be embedded within a line.

Labels not used as markers but as branch references will appear later in the line after the appropriate BASIC keyword. (e.g. GOTO @PLAN.3, GOSUB @INPUT.PROCESS).

The appropriate term for a block of code referenced by a label in most high level languages is a "procedure". In BASIC, these procedures are called subroutines. However, source code in TBA is not BASIC code. Therefore, throughout this documentation, all subroutines will be called PROCEDURES.

The following example shows how to define a label. The example is a procedure (referenced by the GOSUB command) which will flash a message on the video display, and will continue flashing the message until the <ENTER> key is pressed. The example represents actual source code which could be processed by TBA.

```
@flash.message
     cls
     forl=1to20
          i$=inkey$
          if i$=chr$(13) then goto @end.flash.mssg
     nextl
     print@512,"Flashing Message - Press <ENTER> to continue"
     forl=1to50
          i$=inkey$
          if i$=chr$(13) then goto @end.flash.mssg
     next l
     goto @flash.message
@end.flash.mssg
     return
```

The code could be comprised of either upper or lower case characters. The point is that labels are merely line numbers expressed in a different form. Note that the first line in this procedure defines the label. The body of the procedure is written just as ordinary BASIC program code would be. The line containing the statement " goto @flash.message" shows how to reference a label as a point of transfer in a source program.

If any BASIC statements are to follow a label reference, they MUST be separated from the label by a non-alphanumeric character (usually a space or a colon). This is to distinguish the label name from the ensuing statement. As an example, suppose it is desirable to reference this procedure from within a program depending on whether or not a specific condition was met. The following line will represent such a statement.

```
if a=10 then gosub @flash.message else a=a+1
```

If the space between the label "@ flash.message" and the keyword "else" was omitted, the keyword "else" would be taken to be a part of the label name. This would produce an error when the program is processed.

Since this particular routine is written to be referenced by a GOSUB command, note the use of the RETURN command. The RETURN command will terminate the execution of this procedure and return control to the program statement immediately following the call to the procedure.

N 0 T E

Special attention should be paid to the method by which execution of a procedure is terminated. In both of the lines which perform the comparison to see whether the <ENTER> key has been pressed (if i$=chr$(13) ...), if the comparison is true (i.e. the <ENTER> key was pressed) a branch is performed to the @ end.flash.mssg label in order to execute the RETURN command. In ordinary BASIC programming, it would be permissible to have the RETURN statement following the "then" in an "if" conditional, rather than branching to a RETURN statement. However, when writing source code, the RETURN statement is handled in a special manner. **Only one RETURN statement can be used in a procedure and it must always be the last BASIC statement in a procedure.**   The reason for these restrictions of the use of the RETURN command will be explained more thoroughly in the sections on Variable Usage and Processing Source Code.

Why Use Labels?

Labels are merely used to denote a point of transfer in a program and are analogous to BASIC line numbers. What might not be obvious is the advantage gained by using labels as opposed to using line numbers.

One advantage is program readability. When reviewing source code, it is much more descriptive to see a statement such as **GOSUB @flash.message** (rather than GOSUB 1000) in order to know what sequence of events is supposed to follow. The key to making the most efficient use of label names is to choose a name that will be descriptive regarding the function of the routine in question. (It would defeat the idea behind using labels to name the routines @routine1, @routine2, @procedure1, etc.)

Another advantage gained by using labels is the ability to readily remember procedure-names while writing code. When writing BASIC programs, it is much easier to remember a descriptive name pertaining to a routine than it is to remember the line number associated with a routine. Again, carefully chosen routine names will increase program code clarity.

The last reason for using labels is that absolute line referencing may be eliminated. For example, suppose there is a program that contains many different references to the same procedure. Upon completion of the program, it is realized that additional statements need to be added at the beginning of the routine. If the addition of these statements meant that lines would need to be inserted in front of it, then it is obvious that all references to this routine within the program would have to be changed (e.g. if line 1000 was the initial entry point to the routine, and line 995 had to be added and was to become the new entry point, all GOSUB 1000 statements in the program would need to be changed to GOSUB 995). By using labels no absolute line numbers are ever used, and making such a change when writing source code would be accomplished simply by inserting the necessary code. No changes to referencing routines will ever need to be done when using TBA, since labels (rather than absolute line numbers) are used to denote routines.

The idea of having no absolute line referencing can be carried one step further. Normal BASIC programming does allow one to "Merge" programs and routines together. However, absolute line references still must be considered. When writing source code, procedures that will be used in different programs can be merged in with the main text of the program in a very convenient manner without accounting for line number conflicts or remembering line number references. With TBA it is possible to create a library of procedures that can be used over and over again with many different programs.

In summary, labels provide a means to make program source code more readable and understandable. They allow for ease of referencing routines. In addition, since absolute line number references no longer exist, procedures that will be common to various programs can be merged into the main program without the need to worry about line number conflicts, or the line number needed to reference the routine. Because there are no absolute line references, the capability now exists to write BASIC procedures that are relocatable within a program. Any procedure can be placed anywhere within a BASIC program without having to worry about referencing it.

How the TBA Processor Deals with Labels

It has been noted numerous times throughout this section that Labels take the place of line numbers in source code and that line numbers MUST NEVER be used when writing source. The reason for this is that the processing operation (the changing of source code to object code) will insert line numbers into the code. The processing operation will create a program which will run under interpretive BASIC. After processing has taken place, all lines in the source code will have a line number assigned to them. In addition, any lines in the source code which referenced a label (e.g. GOSUB @flash.message) will have a line number substituted for the label.

Using the @flash.message routine again as an example, assume that the first CLS line in the procedure (the line immediately following the @ flash.message label definition) had line number 500 assigned to it after processing. Any reference to this label in the source code will appear in the object code as GOSUB 500 or GOTO 500. Although no line numbers are used when writing source code, the object code will be constructed to include line numbers, just as any ordinary BASIC program would. If absolute line numbers do appear within the source code, they will be left unchanged by the processing routine, and will more than likely produce some type of BASIC error (undefined line) when the object code is executed.

Applying the Theory of Labels to Practical Programming

Knowing the proper syntax for defining and referencing labels is necessary for using them. Without realizing when and why labels are used, programming with TBA would be no different than programming in ordinary BASIC. With the proper use of labels within the source code, the user will have the necessary tools for building and maintaining structured, easy to follow, meaningful code.

A structured BASIC program can be broken up into two distinct elements. The first element would be the main body of the program. The main body of the program should be written in a manner which incorporates a logical flow. Most of the time, the main body of the program is comprised of controlling loops and routines that perform specific operations on the data used by the program. For instance, it will handle the printing of specific prompting messages, or it will perform calculations based on the data used by the program (perhaps in a loop), etc. Branching (using GOTO) in most cases will be limited to either a loop operation, or going to a specific function of the program from a main menu (and returning to the main menu after the particular function is complete).

The second element in a structured program is the PROCEDURE. Procedures can be used for functions that will be done at different logical points throughout the program. An example of a procedure would be a block of code which controls the input of all data to a program. Whatever the purpose of a procedure, it will be accessed by the main body of the program. Procedures should be written in a manner that will be usable by various functions in the main body of the program. A procedure should never be involved in a decision making process with respect to the main body of the program. The function of a procedure should be limited to accepting/retrieving data for the main body of the program to manipulate or make decisions upon. It will also be quite common for a procedure to set conditions for the main body of the program to evaluate. Realize that the procedure should only set the conditions, and should let the main body of the program make the decisions based on the conditions set.

Branching statements in a procedure should be limited to a branch internal to the procedure, and should never branch to a point in the main body of the program. This implies that the proper way to exit a procedure is with the RETURN command.

For the most part, labels will be used to identify procedures that the main body of the program will be referencing. When using labels in this manner, they will represent procedure names accessed with the GOSUB command.

Labels may be used within the main body of the program to represent logical branching points. Labels, when used in this manner, will be referenced by the GOTO command.

It will take some forethought on the user's part to create procedures which will be accessed by the main body of the program. The function of a procedure should be well defined and should not deal with specifics. The more generic a procedure is, the more applicable it will be, not only in the specific program, but also in future programs that will be written.

INTRODUCTION TO VARIABLE USAGE

The use of variables with TBA does differ from ordinary BASIC programming. This section will highlight the differences in variable usage.

Unlike interpretive BASIC, when writing source code, there is not a two character limitation to represent a variable name. Variables may be up to fourteen characters in length. Having longer variable names will make program text more readable and meaningful because descriptive names may be used.

Two different types of variables can be utilized when writing source code. They are Global variables and Local variables. Global variables are variables that will be maintained throughout the program. Local variables are variables that will be maintained only through the duration of a procedure.

Having two different types of variables (Global vs. Local) will make variable usage much more manageable. The programmer will be at liberty to choose temporary variable names which may be used at other points in program code. In this respect the need to worry about destroying information stored in a variable by using it in a different part of the program will be diminished.

Since TBA introduces this concept of Local and Global variables, certain guidelines will need to be followed concerning their usage.

Valid Variable Names

Variable names may be up to fourteen characters in length. The first character must be alphabetic. After the first character in a variable name, the remaining thirteen characters may be alphanumeric. The period and/or underline character <.> or <_ > may also be used in a variable name to provide readability. The variable name MUST be followed by one of the four variable type declaration tags (one of <%>, <!>, <#> or <$>). The declaration tags will represent the following types of variables:

%	Integer
!	Single Precision
#	Double Precision
$	String

Variable names should be at least two characters long (not including declaration tags). The use of single character variables is permitted when writing source code, but single character variables are handled in a special manner. More details on the use of single character variable names will be given later in this section.

The use of upper and lower case for variable names is also important. As was the case with Labels, consistency in using upper and lower case must be maintained when defining and referencing variable names if no conversion will be done during processing. For example, the variable name LOOP1% could represent a different variable than the name loop1%. Normally, however, the two names will be identical. The following will illustrate valid variable names.

```
     Loop1%                 first.name$
     loop1%                 last_name$
     loop1$                 total.dollars#
     Account_total#         spvariable!
```

Notice that using a different declaration tag with the same variable name will create a unique and distinct variable. Note also that all of the variables listed above do contain a declaration tag. If the declaration tag was left off when defining a variable, a "Global Variable Definition Format" error would occur. Although it may seem to be a bit of a bother specifying declaration tags, in the long run it will make source code more readable and meaningful. The specific type of variable being used will always be carried along in the variable name.

Invalid Variable Names

The variable names allowed when writing source code will enable the capability of choosing more descriptive and meaningful variable names. However, there remain some variable names that MUST NEVER be used. The following is a list of such variable names.

Two character BASIC keywords (such as IF, OR, ON, FN, AS)
BASIC keywords that end with a declaration tag (such as TIME$, MKDS, INPUT# etc)
A variable whose first three characters begin with the letters REM

It is allowed to have these BASIC keywords embedded within a variable name (such as real**time$**, sp**rem**ember%, **if**set#, etc.). Any BASIC keyword that is longer than 2 characters (and does not end with a declaration tag) may be used as a variable name (such as goto%, lset!, etc.). If any invalid variable name is used, an "Illegal Variable" error will occur.

Before any variables may be used, they must be defined in some type of definition statement. The definition of variables will differ, depending on whether Global or Local variables are being defined.

Defining Global Variables

To define Global variables, begin a line of source code with an "equal sign" <=> followed by the variable or list of variables to define. If more than one variable is to be defined, the variables must be separated by commas. A Global variable definition statement must be restricted to just the definition of variables and no other statements may appear on a variable definition line. The following example will illustrate how Global variables are to be defined.

```
   =loop1%,loop2%,totaldollars#,firstname$,lastname$,spvariable!
```

In the above example, six variables are defined as Global variables. The variables **loop1%** and **loop2%** will be integer, **totaldollars#** will be double precision, **firstname$** and **lastname$** will be string, and **spvariable!** will be single precision.

Global variables may be defined anywhere within the source code. It is highly recommended that Global variables be defined at the very beginning of the source code. Defining variables at the beginning of the program will cause it to have more structure. It will no longer be necessary to "hunt" through the program code to determine if a variable has been defined. Defining variables at the beginning of the code will also serve as a quick reference list of all Global variables currently in use.

Defining Local Variables

Local variables will be defined in conjunction with the entry point to a given procedure. To define a variable as Local to a given procedure, the variable (or list of variables) will follow the label which defines the entry point into the procedure. An "equal sign" <=> must separate the end of the label name from the first character of the first local variable defined. If more than one variable is to be defined as local to a given procedure, these variables must be separated by commas.

As the name implies, Local variables are used explicitly in a given routine. The value assigned to a Local variable is only important during the duration of the routine. After the routine has been completed, the value of the variable will have no effect on the remainder of the program.

As an example, consider the following rewrite of the example which incorporates the use of local variables. The following will represent the actual source code that could be written to include local variables in this routine.

```
@flash.message= delayloop1%,delayloop2%,kbdscan$
cls
for delayloop1% = 1 to 20
    kbdscan$=inkey$
    if kbdscan$=chr$(13) then goto @end.flash.mssg
next delayloop1%
print@512,"Flashing Message - Press <ENTER> to continue"
for delayloop2% = 1 to 50
    kbdscan$=inkey$
    if kbdscan$=chr$(13) then goto @end.flash.mssg
next delayloop2%
goto @flash.message
@end.flash.mssg
return
```

In this example, there are three local variables which are defined (delayloop1%, delayloop2%, and kbdscan$). Notice that defining local variables is much the same as defining global variables. The exception is that the variable definition appears after a label.

Global vs. Local - An Overall Perspective

Global variables are transformed throughout the program, while local variables are transformed only during the procedure for which they were defined. This is a result of the processing which is performed when changing source to object code. The processing operation will create a program which will be run under the normal interpretive BASIC environment. The operation performed by the processor when dealing with variables is to change each variable encountered to a two character variable. The first phase of the processing will replace all local variables with a distinct two character variable name. After the local variables have been processed, all global variables will have the same replacement process performed on them.

The result is that a local variable will be transformed to a unique two letter variable for the lines actually contained in a given procedure. At no other place in the code will this particular combination of letters be used again. This makes the variable "local" in essence. However, the object code will contain a valid variable name that makes it no different from any other variable. The real difference between Global and Local variables is merely how TBA processes them and does not actually have anything to do with altering the BASIC interpreter.

The following example is an extension of the @ flash.message routine. It is a program consisting of two procedures, which are referenced by the main body of the program. The program does contain several lines which will serve to end the operation of the program. The variable **testvar%** is defined as a global variable. Note that the each of the two procedures contain two local variables.

```
     ' ***
     'Main body of program
     '***
     clear 1000
     =testvar%
@beginning
     testvar%=0
     gosub @flash.message1
     if testvar%=1 then  goto @ending.mssg
     gosub @flash.message2
     if testvar%=1 then  goto @ending.mssg else  goto @beginning
     '***
     '
     'Procedure #1
     '
     '***
@flash.message1=kbdscan$,loop%
     kbdscan$=""
     cls:for loop%=1to20
          kbdscan$=inkey$
            if  kbdscan$=chr$(13) then  goto @end.flash1
            if  kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash1
     next  loop%
     print@512,"flashing  mssg I - enter for 2, x to abort"
     for loop%=1to50
          kbdscan$=inkey$
          if  kbdscan$=chr$(13) then  goto @end.flash1
          if  kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash1
     next  loop%
     goto @flash.message1
@end.flash1
     return
     '***
     '
     'Procedure #2
     '
     '***
@flash.message2=kbdscan$,loop%
     kbdscan$=""
     cls:for loop%=1to20
          kbdscan$=inkey$
            if  kbdscan$=chr$(13) then  goto @end.flash2
            if  kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash2
     next  loop%
     print@512,"flashing  mssg 2 - enter for 1, x to abort"
     for loop%=1to50
          kbdscan$=inkey$
            if  kbdscan$=chr$(13) then  goto @end.flash2
            if  kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash2
     next  loop%
     goto @flash.message2
@end.flash2:return
     'Note : The preceding line DOES meet restrictions on the use of "RETURN"
     '***
     '
     'End of program
     '
     '***
 @ending.mssg
     cls:print@512,"this program has been run in its entirety."
     end
```

The main body of the program will initialize the variable **testvar%** as a global variable after which it will set this variable equal to zero. From this point a looping situation will occur. The main body of the program will call the first procedure which will cause message #1 to flash on the screen. Message #1 will continue to flash until one of two keys is pressed. If the <ENTER> key is pressed a return from the procedure to the main body of the program will be performed and the global variable **testvar%** will be unchanged. If the <X> key (either upper or lower case) is pressed, the global variable will be set equal to one, and a return will be made to the main body of the program.

Once control of the program is returned to the main body from the first procedure, the global variable will be tested. If it is set equal to one (the <X> key was pressed), the program will branch to the ending message routine at the end of the program, and program execution will stop. If the global variable is equal to zero (the <ENTER> key was pressed), normal program execution will continue, and the second procedure will be called.

The second procedure will execute in a similar fashion to the first (with the exception that flashing message will be different). Upon returning to the main body of the program, either a branch will be performed to the beginning of the main body of the program (<ENTER> was pressed to exit the second procedure), or a branch will be performed to the ending message part of the program (<X> was pressed to exit the second procedure).

Although somewhat simplistic, the example will illustrate the differences between global and local variables. These differences can be noted by observing the object code which will be produced when the source code is processed (Note that the remarks listed in the following object code would not normally be produced by the processing operation, but have been inserted to make the object code more readable and understandable).

```
1 '***
2 'Main body of program
3 '***
4 CLEAR 1000
7 TE%=0
8 GOSUB 18
9 IF TE%=1 THEN GOTO 60
10 GOSUB 39
11 IF TE%=1 THEN GOTO 60 ELSE GOTO 7
12 '***
13 '
14 'Procedure #1
15 '
16 '***
18 KB$=""
19 CLS:FOR LO%=1TO20
20 KB$=INKEY$
21 IF KB$=CHR$(13) THEN GOTO 32
22 IF KB$="X" OR  KB$="x" THEN TE%=1:GOTO 32
23 NEXT LO%
24 PRINT@512,"flashing  mssg 1 - enter for 2, x to abort"
25 FOR LO%=1TO50
26 KB$=INKEY$
27 IF KB$=CHR$(13) THEN GOTO 32
28 IF KB$="X" OR  KB$="x" THEN TE%=1:GOTO 32
29 NEXT LO%
30 GOTO 18
32 RETURN
33 '***
34 '
35 'Procedure #2
36 '
37 '***
39 KC$=""
40 CLS:FOR LP%=1TO20
41 KC$=INKEY$
42 IF KC$=CHR$(13) THEN GOTO 53
43 IF KC$="X" OR  KC$="x" THEN TE%=1:GOTO 53
44 NEXT LP%
45 PRINT@512,'-flashing  mssg 2 - enter for 1, x to abort"
46 FOR LP%=1TO50
47 KC$=INKEY$
48 IF KC$=CHR$(13) THEN GOTO 53
49 IF KC$="X" OR  KC$="x" THEN TE%=1:GOTO 53
50 NEXT LP%
51 GOTO 39
53 RETURN
54 '***
55 '
56 'End of program
57 '
58 '***
60 CLS:PRINT@512,"this program has been run in its entirety."
61 END
```

From comparing the original source code to the processed object code note that the labels and variables contained in the source code were transformed into line numbers and two character variables in the object code. The chart listed below will give the transformations performed on the labels.

Labels in Source	Line Numbers in Object
@beginning	7
@flash.message1	18
@end.flash1	32
@flash.message2	39
@end.flash2	53
@ending.mssg	60

The processing performed on the label names should be quite obvious. The file was assigned line numbers based upon the character X'0D'. Line 1 was comprised of all characters from the beginning of the file until the first "<ENTER>" key character. Line 2 was comprised of all characters from that point until the next carriage return (<ENTER>) was encountered, etc. The processing performed on the variable names may not be quite as obvious. Below is an explanation of the results of the processing performed on both the global and local variables.

When variables are processed, the local variables are done first. Consider the flashing program listed. The first variables processed were the local variables (kbdscan$ and loop%) defined in the @flash.message1 routine. Compare the source to the object code and it will be detected that the variable kbdscan$ in the source code is represented as KB$ in the object. Similarly, loop% was transformed to LO%.

Notice that in the source of the @flash.message2 procedure, the same two local variables defined in the first procedure were also defined here. The point to notice is the transformation of these two variables. In the second flashing routine, the variable kbdscan$ was translated into KC$ (as opposed to KB$ in the first routine), while loop% was changed to LP% (as opposed to LO%). This translation of the same local variable name (used in two **different** procedures) into two different variable names is the basis for having local variables.

For the duration of any procedure, all occurrences of a defined local variable will be translated into a unique variable name. A procedure is defined as being the self-contained code between the label defining the entry point into the procedure and the first occurrence of a RETURN statement. All variables that are within this self-contained procedure and are defined as being local to the procedure will be translated into unique variable names. If the same local variable is defined in more than one procedure, it will get translated into a **different** variable name in each subsequent procedure. What this means is that if a variable is defined to be local to a procedure, there is no chance of destroying a value which might be assigned to a variable that has the same name. This concept of local variables will eliminate the need to "chase down" variables when a new variable is introduced. There is no longer a need to worry about destroying a value already assigned to a variable.

Using local variables will also aid in the debugging of programs. When entering a procedure, the value assigned to a local variable will be solely determined by the procedure. This will prevent the occurrence of having a variable assigned to some obscure number as the procedure is entered. When re-entering a procedure, it will be up to the programmer to initialize the local variables in the procedure (if required). If a procedure is re-entered, the values assigned to the local variables will be carried over from the last time the procedure was executed. Values carried to and from procedures must be equated to Global variables.

When the processing of local variables in a given procedure is taking place, the transformation of variables will take place from the beginning of the procedure up to the RETURN statement. For this reason, there may only be one RETURN statement associated with each procedure definition and the RETURN statement **MUST** appear as the last line of the procedure, and **MUST** be the only BASIC statement on that line.

If it is necessary to perform a return from somewhere within a procedure, the return must be done as illustrated in the @ flash.message routines. That is perform a GOTO, and branch to a label at the end of the procedure. Following this label will be the RETURN statement. (It is allowed to have the label and the RETURN statement on the same line. E.G. "@end.flash1:RETURN") As is illustrated in the @ flash.message examples, the label name "@ end.procedure" could be chosen to represent the branch to the RETURN statement (e.g. @end.flash1 was used as the label name to represent the branch to the RETURN statement).

After all local variables have been processed, the global variables are transformed into two character variable names. In the above example, the only global variable defined was the variable testvar%. This variable was GLOBALLY replaced throughout the entire program text by the variable TE%. Note that regardless of where testvar% appeared in the source code (i.e. in the main body of the program as well as within the procedures), it was transformed into the variable TE%.

If it is desirable to have a variable which is maintained throughout the entire body of the main program, define it to be a global variable. If the only concern is with the value of a variable during the duration of a procedure, defined it as a local variable. To pass variables between the procedure and the main body of the program, the variables common to both should be defined as global (in the flashing message program above, testvar% is a variable that is passed between the main program and the procedures and hence was defined as global).

When defining variables (either local or global), at least two characters (not including the declaration tag) must be used to represent the variable name. This is because no processing will be performed on single character variables. No transformations will occur to single character variables. If it is necessary to have variables within the program that will not get processed and will be maintained solely by the user, then use single character variable names. To use single character variables, incorporate the variable within the code WITHOUT defining it. Defining a single character variable as local or global will generate an error during processing.

The same variable name may be defined as both global and local. The variable name will represent a different variable in the local routine than it represents on a global basis. Consider the following source code which uses the variable **test%** as both a local and global variable.

```
   =test%
        clear 1000
        test%=0
        gosub @sub1
        print"this is the current value of  test% -->";test%
        end
   @sub1=test%
        for test%=1 to 10
             print  test%
        next  test%
        return
```

If this source code were to be processed it would produce the following object code.

```
   2 CLEAR 1000
   3 TF%=0
   4 GOSUB 8
   5 PRINT"this is the current value of  test% -->";TF%
   6 END
   8 FOR TE%=1 TO 10
   9 PRINT TE%
   10 NEXT TE%
   11 RETURN
```

Note that in the procedure @sub1, the local variable   test% was transformed into the variable TE%, while the global variable test% was transformed into TF%. This means that to pass a global variable to a procedure, define the variable as being global **ONLY**, and do not define it as a local variable in the procedure. Doing the latter will cause the variable not to be passed. Different variable names will represent the variable.

If a procedure performs a call to a second procedure and it is desired to have variables common to both procedures then these common variables MUST be defined as global. Consider the following example which illustrates one procedure (@sub1) that references another procedure (@sub2).

```
   @sub1=var1%,loop%
        for loop%=1 to 100
             var1%=loop%
   '
   'call of @sub2 within @sub1
   '
             gosub @sub2
             print var1%
             return
   '
   'entry to @sub2
   '
   @sub2=var1%
        var1%=var1%+(5*100)
        return
```

Notice that @sub1 defines the variable var1% to be local, as does @sub2. If the above source code was processed, the following object code would be generated. (But without the REM statements.)

```
   2 FOR LO%=1 TO 100
   3 VA%=LO%
   4 REM
   5 REM call of @sub2 within @sub1
   6 REM
   7 GOSUB 14
   8 PRINT VA%
   9 RETURN
   10 REM
   11 REM entry to @sub2
   12 REM
   14 VB%=VB%+(5*100)
   15 RETURN
```

Comparing the source code to the object code, observe that the variable var1% in the @sub1 routine was translated into VA%, while var1% in the @sub2 routine was translated into VB%. This would mean that the variable would not be passed between the two procedures. To remedy this situation, it is necessary to define var1% as a global variable and NOT define it as a local variable in either procedure.

Up until this point, the use of simple variables has been discussed and not the use of arrays. The procedure used to define, dimension and utilize arrays is very similar to that of simple variables. To define an array, place the array **NAME** in a variable definition statement. The array name MUST be followed by a declaration tag just as though it was a simple variable being defined. Only the array name should be in the definition statement (the use of parentheses and/or a subscript value in a variable definition statement will cause an error during processing). The following example of source code will show the proper method to define, dimension and utilize an array (the array that will be utilized is called **TOTALS#** - all other variables in the definition statement represent simple global variables).

```
=totals#,loop1%,loop2%,dimrow%,dimcolumn%
     clear 1000
     dimrow%=50
     dimcolumn%=20
     dim totals#(dimrow%,dimcolumn%)
     for loop1%= 1 to  dimrow%
          for loop2%= 1 to  dimcolumn%
               totals#(loop1%,loop2%)=0
          next loop2%
     next loop1%
```

If the source code were to be processed, the following object code would be generated.

```
2 CLEAR 1000
3 DI%=50
4 DJ%=20
5 DIM TP#(DI%,DJ%)
6 FOR LO%= I TO DI%
7 FOR LP%= 1 TO DJ%
8 TP#(LO%,LP%)=0
9 NEXT LP%
10 NEXT LO%
```

By comparing the source code to the object code, observe that any references to the array totals# in the source code were translated to TP# in the object code. Using defined variables as subscripts (in the above example, the variables dimrow% and dimcolumn% were used as subscripts) is allowed. Defined variables may be used in any facet normally used in a BASIC program (i.e. variables may be used along with any BASIC function).

The final point concerning the use of variables is that if they have a declaration tag, they must ALWAYS be defined (with the exception of single character variables). If a variable is referenced but not defined, no processing will be performed on the variable and an "Undefined Variable" ERROR will occur in processing. Variable names without tags will not be processed and might cause a syntax error.

By far the most common errors to occur when learning these procedures are the omission of a declaration character to variables and the inadvertent use of an undefined variable. The latter may often be due to misspelled variable names.

Miscellaneous Differences and Information

The final part of Section I will deal with minor differences between programming in BASIC and writing source code. An additional concept concerning the use of Labels will be explained.

The REM statement can be used within the source code in the same manner that it is used when writing normal interpretive BASIC programs. When the REM statement (not the <'> abbreviation) is used, no processing will be performed on it, and it will remain in the object code as it appeared in the source code. When the REM abbreviation <'> is used in the source code it will be removed by the processing routine and will NOT appear in the object code. If it is necessary to have REM statements appear in the object code use the REM statement. The REM keyword must be first on a line and therefore, the REM line must be a separate line. If it is desired to have remark statements in only the source code then use the REM abbreviation

Spaces may be used throughout the source code text to improve readability. When the processing of source code is performed, all spaces that are not contained in literal strings can be removed and the object code will have no extraneous spaces. This will allow the incorporation of spaces within the source code without the worry of using up extra memory in the processed object code. The only exception to this is the use of any variable beginning with the letter "C" when used after the keyword "AS" in a field statement. Deleting this space would form the reserved word "ASC" and result in a syntax error. Therefore, the space is not deleted.

When defining a label, additional BASIC statements may appear on the same line as the label definition, provided that no local variables need to be defined along with the label. Note that a label definition (where it is the start of a branch) may NOT be embedded within a line and must appear as the first statement on the line. To incorporate BASIC statements on a labeled line, the label must be followed by a colon <:>. The following example line will show how to write a source line which contains BASIC statements which follow a label.

```
   @delay.loop:for l= 1 to 2000:next l
```

This concludes Section I of this manual. The user is encouraged to re-read this section if all of the concepts are not totally understood.

Section II - Writing Source Code

This section of the manual will explain how to write source code. Source code may be written in one of two ways - either by using a word processor/text editor (such as SCRIPSIT or LED) that has the capability of saving a file in ASCII, or in the ordinary interpretive BASIC environment. Regardless of how source files are created, it is highly recommended that they be assigned a common file extension when saved to disk. Since both source and object files will be utilized, using a common extension (such as /TBA for The BASIC Answer) will help differentiate between the two. Only the object file will be an executable program.

Using a Word Processor/Text Editor to Write Source Code

Writing source code is very easy when using a word processor or a text editor. There are several things that are required to ensure that the source code will be processed without errors.

First of all, the source text to be processed MUST contain only source text. Characters or lines that serve a specific function to the word processor or text editor may not be contained in the source code when it is processed. Page formatting lines, block markers or any other control characters cannot appear in the source code to be processed. However, they may be used when writing source code (to move blocks of code around or to make source listings more readable). If used, these special characters must be deleted from the source code before it is processed.

The second point is that all text MUST be saved as an ASCII file. Certain word processors will save some characters in the text as non-ASCII characters unless told to save the file in ASCII. (i.e characters outside of the range X'00' to X'7F'). For example, SCRIPSIT will save a carriage return (X'0D') as a non-ASCII character (X'8D'). No characters may appear in the text that have a value less that X'20' (with the exception of a carriage return or linefeed character). For more information on saving a file in ASCII, consult the documentation of the word processor or text editor that will be used.

The last character in the file MUST be a carriage return. All lines in the program text must end with a carriage return character and the last character in the text must be a carriage return. Some word processors may have "extraneous" spaces at the end of the last carriage return. It is advised that prior to saving a program text file a "delete to the end of text" is performed after the last carriage return in the program.

N 0 T E for S C R I P S I T users

The upward arrow on the keyboard has a conflicting use in BASIC as opposed to the word processor. In BASIC, the symbol generated is used to designate exponentiation. "3[2" means three squared. The key in Scripsit will generate a reverse line feed. Under LDOS version 5.1.1 or later, if the KI driver is active (under LSCRIPT), the proper character will be entered by using <CLR><,> (clear key and comma key).

Using the BASIC Interpreter to Write Source Code

The easiest and most efficient way to write source code is to use a word processor or a text editor. However, source code may be written in the normal BASIC environment as well. If using interpretive BASIC to write source code, there are several guidelines that must be adhered to.

As was noted in Section I, the use of internal line numbers in the source code should not be done. This is true regardless of how source code is created. However, when writing source in interpretive BASIC, line numbers MUST be used as a means to enter the source code. This is due to the fact that BASIC will interpret any keyboard entries issued from the READY prompt, and act on them immediately unless preceded by a line number (in which case the line will be stored in memory as program text). The idea behind writing source code in BASIC is to use line numbers solely for the purpose of storing the program text in the memory reserved by BASIC for text. No line number references should appear in the source code.

The following example will show how source code should be written from within BASIC.

```
110 @FLASH.MESSAGE
120 CLS
130 FORL=1TO20
140    I$=INKEY$
150    IF I$=CHR$(13) THEN GOTO @END.FLASH.MSSG
160 NEXTL
170 PRINT@512,"Flashing Message - Press <ENTER> to continue"
180 FORL=1TO20
190    I$=INKEY$
200    IF I$=CHR$(13) THEN GOTO @END.FLASH.MSSG
210 NEXTL
220 GOTO @FLASH.MESSAGE
230 @END.FLASH.MSSG
240 RETURN
```

Note from the above example that no references are made to the line numbers used. The line numbers are used so that source code may be stored in memory without being acted upon immediately by the BASIC interpreter.

When source code is processed, any line numbers that are found at the beginning of a line will be stripped off. The result will be just the source code (without line numbers).

After the source code has been written, it can be saved just as a normal BASIC program would be. However, when saving source code, the "A" parameter MUST be specified in the SAVE command. The processor can only deal with ASCII files. If the "A" parameter is not specified, all BASIC keywords will be saved in their "compressed" form which will cause non-ASCII characters to be saved to the file. This will cause totally unpredictable results when the file is processed. To save a file in ASCII, use the following syntax.

```
SAVE"filename",A
```

Two more IMPORTANT points need to be made concerning the writing of source code in the BASIC environment. First, no program line can exceed 240 characters in length (including the line number). This is because BASIC cannot load a program which has a line longer than 240 characters. If an attempt is made to load a program line longer than 240 characters, the load will be aborted and a "Direct Statement in File" error will result. The rest of the program will be inaccessible. It will be necessary to load programs in BASIC so that the source code may be edited. Having source lines longer than 240 characters will lead to disastrous results.

The second point is that although line numbers are not referenced in the source code, the sequence of line numbers used is important. As program lines are written, the BASIC interpreter will insert them into the program text in sequence by line number. For this reason, the source statements which are written must have line numbers assigned to them that will cause the proper sequencing of the lines. When using line numbers to write source code, it is a MUST to choose line numbers that will result in the proper sequencing of lines, just as it is done when writing normal BASIC programs. This does not allow the transportability of source code because it will no longer be relocatable within the program. For this reason, it is highly recommended that source code be written with a word processor or text editor, rather than from the BASIC environment. Another disadvantage of using BASIC to create source code is that lower case is automatically converted to upper case outside of quotes or remarks. Variables, labels, and directives can not be generated in lower case.

Using DIRECTIVES when Processing Source Code

The use of directives will enable the advantage of several advanced features incorporated in TBA. This section will describe how Directives can be used within the source code, as well as specifying Directives during the processing phase.

A Directive serves as a means to alter the output associated with a processed file. In addition, by proper use of Directives, conditional processing of source files may be achieved. Here is a list of the allowable Directives.

*PRLINES	(May be used in Source code only)
*LIST ON/OFF	(May be used in Source code only)
*PAGE	(May be used in Source code only)
*TITLE	(May be used in Source code only)
*IF expression	(May be used in Source code only)
*END	(May be used in Source code only)
*expression	(May be used in Source code or at Directive prompt)

Any Directive may be incorporated within the source code. Only the Directives that deal with "expressions" may be used at the "Directive" prompt in the processing utility. Source code Directives are identified as an asterisk (*), followed by the desired Directive. If used in the source code, a Directive must be the only information contained on a line. Directives cannot be embedded within a line and no other statements (either BASIC or definition) may follow a Directive on the Directive line.

The first four Directives discussed (*PRLINES, *LIST, *PAGE and *TITLE) will control the output generated by the processing utility, and may only be used within source code.

*PRLINES

The *PRLINES directive can be used to set the number of lines to be printed on a page. It must be placed on its own line in the source code and be followed by a parameter which is a number between 20 and 254
The proper syntax is :

```
     *PRLINES=N
```

where n is a number from 20 through 254. If the directive is not specified the default would be 56 printed lines per page. While printing, when the specified number of lines has been printed, an X'0C' character will be issued to the printer, resulting in a top of form position.

Initial top of form alignment should be done prior to processing if the line printer is to be used. In order to do this, be sure the printer is "on line" and issue a top of form command. (E.G. From LBASIC the command "LPRINT CHR$(12)" would do or if MINIDOS filter is active a <CLR><SHF><T> can be used.) After the paper has moved, position so that the initial print will occur where desired. From this point on, a top of form command will position the paper at the same relative line on subsequent pages.

If the printer is capable of printing only 80 characters per line, the LDOS printer filter (PR/FLT) should be used with the CHARS parameter set at 80 or proper pagination will not occur. In order to do this, at LDOS Ready type FILTER *PR   PR (C=80). Other print filter parameters may be set, but note that any INDENT will wreak havoc with the formatted output.

LIST ON/OFF

The *LIST Directive can be used to turn on/off the listing of the processed code which is generated by the processing utility. It will work primarily as a switch so that the listing of certain parts of the code may be disabled. The best way to explain its use and function is through an example.

Assume that it was desired to process source code and only the main body of the program and the ending message were to be displayed in the listing. The following *LIST Directives could be added to the source code to produce the desired results. In the source code listing that follows the addition of the *LIST Directives will be denoted by being underlined and bold faced.

```
          clear 1000
          =testvar%
     @beginning
          testvar%=0
          gosub @flash.message1
            if testvar%=1 then  goto @ending.mssg
          gosub @flash.message2
            if testvar%=1 then  goto @ending.mssg else  goto @beginning
     *LIST OFF
     @flash.message1=kbdscan$,loop%
          kbdscan$=""
          cls:for loop%=1to20
                 kbdscan$=inkey$
                 if  kbdscan$=chr$(13) then  goto @end.flash1
                 if  kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash1
          next  loop%
          print@512,"flashing  mssg 1 - enter for 2, x to abort"
          for loop%=1to50
                 kbdscan$=inkey$
                 if  kbdscan$=chr$(13) then  goto @end.flash1
                 if  kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash1
          next  loop%
          goto @flash.message1
     @end.flash1
          return
     @flash.message2=kbdscan$,loop%
          kbdscan$=""
          cls:for loop%=1to20
                 kbdscan$=inkey$
                 if  kbdscan$=chr$(13) then  goto @end.flash2
                 if  kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash2
                next  loop%
          print@512,"flashing  mssg 2 - enter for 1, x to abort"
          for loop%=1to50
               kbdscan$=inkey$
               if  kbdscan$=chr$(13) then  goto @end.flash2
               if  kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash2
          next  loop%
          goto @flash.message2
     @end.flash2
          return
```
***LIST ON**

```
     @ending.mssg
          cls:print@512,"this program has been run in its   entirety.":end
```

If, during processing, this code is listed to the printer or screen, the following will represent the listed output produced.

```
       clear 1000
     ====>> 1clear10000

       =testvar%
     ====>>

       @beginning
     ====>>

       testvar%=0
     ====>> 2TE%=0

       gosub @flash.message1
     ====>> 3gosub7

       if testvar%=1 then  goto @ending.mssg
     ====>> 4ifTE%=1thengoto35

       gosub @flash.message2
     ====>> 5gosub21

       if testvar%=1 then  goto @ending.mssg else  goto @beginning
     ====>> 6ifTE%=1thengoto35elsegoto2

       *LIST OFF
     ====>>

       @ending.mssg
     ====>>

       cls:print@512,"this program has been run in its   entirety.":end
     ====>> 35cls:print@512,"this program has been run in its   entirety.":end
```

It can be seen from the above example that the *LIST Directive will allow control of the exact parts of the program that will be listed during processing. *LIST will have no effect on the output of the processed file. The entire program will be written to disk. Listings to either the printer or the screen will be affected by *LIST.

Notice that the *LIST ON need not be specified at the beginning of the source file. A listing will always be generated until the first *LIST OFF command is encountered Also the *LIST OFF Directive will appear in the listing, and the *LIST ON will not. This is normal and will have no effect on the object file.

*PAGE

The *PAGE Directive will allow the sending of a top of form character (X'0C') to the printer during the listing of a processed file. This will be useful to break up the printed listing produced. The *PAGE Directive will only affect a listing to the printer and will have no affect on listings sent to the screen. The following example will illustrate the function performed by the *PAGE Directive.

Using the flashing message program once more as the source code, here is how to generate a listing to the printer of the processed file. By using the *PAGE Directive the printer listing can be sectioned, so that the main body of the program listing will appear on a separate page, as well as each individual procedure.

```
          clear 1000
          =testvar%
     @beginning
          testvar%=0
          gosub @flash.message1
            if testvar%=1 then  goto @ending.mssg
          gosub @flash.message2
            if testvar%=1 then  goto @ending.mssg else  goto @beginning
     *PAGE
     @flash.message1=kbdscan$,loop%
          kbdscan$=""
          cls:for loop%=1to20
                kbdscan$=inkey$
                  if  kbdscan$=chr$(13) then  goto @end.flash1
                  if  kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash1
          next  loop%
          print@512,"flashing  mssg 1 - enter for 2, x to abort"
          for loop%=1to50
                kbdscan$=inkey$
                if  kbdscan$=chr$(13) then  goto @end.flash1
                if  kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash1
          next  loop%
          goto @flash.message1
     @end.flash1
          return
     *PAGE
     @flash.message2=kbdscan$,loop%
          kbdscan$=""
          cls:for loop%=1to20
                kbdscan$=inkey$
                  if  kbdscan$=chr$(13) then  goto @end.flash2
                  if  kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash2
          next  loop%
          print@512,"flashing  mssg 2 - enter for 1, x to abort"
          for loop%=1to50
                kbdscan$=inkey$
                  if  kbdscan$=chr$(13) then  goto @end.flash2
                  if  kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash2
          next  loop%
          goto @flash.message2
     @end.flash2
          return
     *PAGE
     @ending.mssg
          cls:print@512,"this program has been run in its   entirety.":end
```

The printed listing of this file would be contained on four physical pages. The first page would contain the main body of the program. The second page would contain the first procedure (@flash.message1). The third page would contain the second procedure (@flash.message2). The final page would contain the ending message routine. The normal printed output would be one continuous block of printed text.

*TITLE

The *TITLE Directive will allow processed listings to contain titles at the top of each page. Its primary use will be with printed listings, but it can also be used with listings which are displayed on the video screen. The *TITLE Directive may appear anywhere in the source code, and will cause ALL pages of the listing to be titled. Unlike the *LIST Directive, this one may not be turned off.

The syntax for the *TITLE Directive is as follows.

***TITLE "string"**

Where "string" is a parameter that MUST be specified with the *TITLE Directive. It will represent the title string that will be printed. The title string can be up to 14 characters in length and must be enclosed within quotes (the ending quote is optional). If the title string is not specified, an error will be generated during processing, and the appropriate error message will be displayed. If the title string specified is longer than 14 characters, all characters after the 14th will be ignored and will not be printed.

In addition to the title string being printed, the following information will also appear on the title line.

Page Number of listing (Numbered consecutively from one).
Current Date
Current Time (retrieved from the time clock)

The following represents a sample of the title line that will be generated when using the *TITLE Directive. The title string that will be used in this example is "Flash Message".

```
         BASIC Answer  Flash Message  June 24, 1982  12:03 A.M.     Page 1
```

The last piece of information printed on the title line <1> will represent the page number of the listing. If a title is used, a double space will separate the title from the first line listed on the page.

The next three directives (*IF expression, *END and *expression) will allow conditional processing of source code.

*IF expression / *END

The *IF expression (which will be referred to as just *IF) and *END Directives will allow the establishment of conditional processing of the source code. This will be a useful feature if there is a need to produce multiple object files that will utilize the same code, with the exception of a few differences.

To illustrate how the *IF / *END Directives operate, consider the following BASIC line using the conditional IF/THEN statement.

```
   IF A=10 THEN  PRINT"Condition is  true":PRINT"A is equal to ten"
```

This very simple example can be used to explain the concept behind *IF / *END Directives. In the above line, a condition is tested (is the variable A equal to 10?). If the condition is true, the two PRINT statements following the THEN will be executed. If the condition is false, the two PRINT statements following the THEN will be ignored.

With the *IF / *END Directives, the same type of conditional testing may be applied. It will be applied to the processing of source code, as opposed to being incorporated in the BASIC program. Conditional blocks are allowed within the source code. These blocks will begin with the *IF Directive and end with the *END Directive. All code that exists between the *IF and *END Directives will belong to that conditional block. If an *IF Directive is encountered during processing, a conditional test will occur. If the condition turns out to be true, all code within that conditional block will be processed. If the condition test is false, all code within that conditional block will be ignored by the processor.

The means by which conditions are tested are by an "expression" following the *IF. An expression is represented by alphanumeric characters. It can be up to fourteen characters in length and the first character must be alphabetic. Period and underline characters (<.> <_>) may also be used in expressions to improve readability.

Before continuing the explanation of *IF / *END Directives, consider the following example of source code which incorporates their use.

```
   =total.items%,index.array%
   '
   *IF hard.drive
   total.items%=4000
   *END
   '
   '
   *IF floppy.drive
   total.items%=500
   *END
   '
   '
   dim index.array%( total.items%)
```

In the above example, the variable total.items% and the array index.array% are defined to be global. Two conditional blocks follow the definition statement. The first conditional block will test to see if the condition " hard.drive" is true. If it is, all code between the *IF and the corresponding *END will be processed and will be contained in the object file after processing is completed. If the condition proves to be false, all code between the *IF and the *END will be ignored during processing and will not appear in the object file.

The second conditional block will test to see if the condition " floppy.drive" is true. If the condition proves to be true, all code between the *IF and *END will be included in the processing and written out to the object file. If it proves to be false, all code between the *IF and *END will be ignored during processing and will not be written out to the object file.

As the names of the conditional blocks might imply, this particular source code may be a part of a larger program which will manage data stored on a drive. If the drive in question is a hard drive (and the proper condition is set), the object code that will be generated will allow the variable total.item% to be initialized to 4000 and the array index.array% will be dimensioned accordingly. If the drive in question is a floppy, the object code that will be generated will allow the variable to be initialized to 500 and the array will be dimensioned accordingly. The difference in the initialization of the variable might indicate that 4000 data items will be allowed by the program if running on a hard drive and only 500 if running on a floppy.

Up to this point, a foundation has been laid for establishing conditional blocks and explaining to an extent how they function. There are additional concepts to be used with conditional blocks. The following is the proper method of establishing conditionals (i.e. setting a condition true or false). Conditional values are established using the *expression Directive.

<u>*expression</u>

The *expression Directive is used to establish a condition of being true or false. It can be incorporated within the source code, or can be issued at the "Directives :" prompt in the processing utility. This directive will be used in conjunction with the *IF / *END directives and will dictate the outcome of an *IF conditional block.

The "expression" used will be the same as the expression specified in the *IF Directive. If an *expression is specified, the corresponding *IF conditional be true and all code within the conditional block will be processed. By NOT specifying an *expression, the corresponding *IF conditional be false and the code within the conditional block will NOT be processed.

Consider the following example, which incorporates the *expression Directive in the hard/floppy drive program from the last section.

```
*floppy.drive
'
=total.items%,index.array%
'
*IF hard.drive
total.items%=5000
*END
'
'
*IF floppy.drive
total.items%=500
*END
'
'
dim index.array%( total.items%)
```

Since the expression "* floppy.drive" was specified, any occurrence of the conditional "*IF floppy.drive" would be true and the code within the conditional block would be processed. Since the expression "* hard.drive" was NOT specified, any occurrence of the conditional "*IF hard.drive" would be false and none of the code within the conditional block would be processed. If the above source code were processed, the following object code would be produced.

```
1 TP% = 500
2 dim IN%(TP%)
```

The variable total.items% was translated into TP%, while the array (index.array%) was translated into IN%. Notice that since the expression * floppy.drive was specified (the first source code line), the *IF floppy.drive conditional proved to be true and the variable TP% (total.items%) was initialized to equal 500. Since the expression *hard.drive was not specified, the *IF hard.drive conditional proved to be false and the code within that conditional block was not written out to the object file.

In general, to use Directives to perform conditional processing of the source code, follow these steps.

1.) Establish the conditional block within the program by using the *IF and *END Directives.

2.) To process the code within the conditional block, specify the "*expression" within the source code, where the expression will be the same as the one used in the *IF block to be processed.

3.) In order NOT to process a conditional block, do NOT specify the "*expression" which was used in the corresponding *IF Directive.

In simple terms, conditional blocks may appear anywhere within the source code. If an "*expression" is found which matches the expression in the *IF directive, the *IF conditional will evaluate to be true and the code within the block will be processed. If no "*expression" is found to match the expression in the *IF conditional, it will evaluate to be false and the code within the block will not be processed.

Besides using "*expression" in the source code. The "*expression" Directive may be issued during processing. This is done by entering the "*expression" to be set true as a response to the "Directives :" prompt which is displayed by TBA. This gives the capability of defining the conditional processing required at processing time, rather than having the conditions embedded within the source code (this was done in the previous example).

To set conditions at the time of processing, simply type in the "*expression" to set to true in answer to the "Directives :" prompt. For example, to set the " hard.drive" conditional to true in the previous example, answer the "Directives :" prompt by typing in the following.

```
   Directives : hard.drive <ENTER>
```

To input more expressions simply separate *expressions with a comma. Any *IF conditional found in the source code that contains the expression " hard.drive" will be evaluated to be true. After entering the "*expression" at the Directive prompt, the prompt will reappear, at which time press <ENTER> or add more "*expression" directives. It is important to note that in order to end the "Directives :" prompt that the <ENTER> key must be pressed as the first character of the line. Otherwise, the prompt will reappear because it is assumed that more *expressions are to be entered.

A few additional points need to be made concerning the entering of "*expressions" to set conditionals true. These will apply to having "*expressions" in the source code, as well as entering them in response to the "Directives :" prompt.

As with variables and labels, the processor will normally ignore upper and lower case to differentiate "*expressions". An expression "*HARD.DRIVE" would be exactly matched by "*hard.drive" because the only difference is the case of the letters. However, TBA can be switched to differentiate cases. In that case, it is most important that the "*expression" entered MUST be a character for character match of the corresponding expression in the *IF conditional. Lower case letters will produce a different expression than the same expression name in UPPER case (e.g. * hard.drive will represent a different expression than * hard.Drive).

More than one conditional may be listed on an "*expression" line both during processing and within the source code. Separate the conditionals that are to be set to true by commas.

The following examples will show how more than one conditional may be set in a single line (the spaces following the commas and preceding the next expression name are not required).

```
During Processing:
Directives : single.den, floppy.drive, forty.track, model1

Within the Source:
*single.den, floppy.drive, forty.track, model1
```

Either of the above lines will cause the four conditionals listed to be set to true.

When specifying *IF , *END and *expression Directives within the source code, they MUST be self contained on separate lines. They cannot be embedded within a line and no statements (either BASIC or definition) can follow them.

The *expression Directive must appear in the source code BEFORE the corresponding *IF conditional. If the *IF conditional physically precedes the *expression, it will evaluate to be false. As a general rule, all *expressions used within the source code should appear somewhere in the beginning of the source text. Using this as a guideline will not only ensure that all conditionals get set, but it will also group the setting of conditionals together at the beginning of the text. It will not be necessary to "chase" down any unwanted conditionals that were set from previous processings of the file if this method is used.

If an *END Directive is encountered without a corresponding *IF, it will be ignored. The results of the processing will be the same as if there were no conditional.

If an *IF is specified without a corresponding *END, all code from the *IF statement to the end of the source file will be taken to be a part of the conditional and will be processed accordingly.

It is permissible to have as many *IF / *END conditional blocks within the source code as is desired. However, these conditional blocks must be self contained. No nesting may be done with conditional blocks. The following arrangement of conditional blocks is NOT allowable.

```
*IF expression

*IF expression

*END

*END
```

If two *IF Directives are encountered without an *END separating them, the following error message will be displayed.

Nested IF Encountered

The processing of this file will be terminated and returned to the "LDOS Ready :" prompt.

Section III - Using TBA to Process files

After source code has been written it must be processed into object code so that it may be run as a normal BASIC program. This section of the manual will explain how to use TBA to process source code into an executable program, as well as additional advanced features that can be used when processing.

No matter how the Source Code was generated, a numbered source listing should be obtained prior to processing. At the "LDOS Ready" prompt enter the following :

```
     LIST filespec/ext:d (N,P)
```

This will send a numbered listing to the printer of the source file. The numbers on the left of the printout will be referenced by the processor both as line numbers and as the reference point for processing errors.

If printer output is intended, top-of-form, should be established on the printer before preceding.

Processing Source Code

To use TBA enter the following from the LDOS Ready prompt.

TBA <ENTER>

After typing the above command, the following screen display will appear.

```
=====================================================
| LDOS - BASIC Processor                            |
| Copyright (C) 1981 by Logical Systems Inc.        |
| Version 1.x                                       |
|                                                   |
|                                                   |
| Source File :                                     |
=====================================================
```

At this point it is possible to enter a filespec to tell TBA what file is to be processed.

Pressing the <BREAK> key at any prompt will abort the processing program, and return to the LDOS Ready prompt. Answering the "Source File" prompt with a   filespec will allow the processing of a source file into an executable object file. The default extension on the filespec will be "/TBA". If the source filespec has no extension, type the filename followed by a "/".

The next prompt will be

Object File :

which can be answered in several ways

```
<ENTER> - - - - will default the filename to be the same as the source
                 filespec but with the extension "/BAS" to the first
                available drive.
</ ext>  - - - - will default the file as above but append the
                specified extension (/  ext).
<:d>  - - - - - will default the filename as above and append a /  BAS:d
                extension.
< filespec>  - - will save the object code under the specified
                 filespec.  If no extension is utilized, the default
                will be "/BAS".
< filespec/> - - will save the object code under the   filespec and no
                extension.
```

WARNING While processing the source file, the processing <u>WILL</u> begin to write processed code to the diskette under the filespec specified for object. Therefore, be aware that the object file write may occur regardless of user selection. Choose an object filespec so that nothing you wish preserved will get overwritten.

The next prompt will be

Processing Parms :

At this time specify the type of output to be generated pertaining to the processed file.

In order to continue, there are several different ways that this prompt could be answered depending upon the type of output that would be required. Pressing <ENTER> in response to this prompt will utilize the default processing options listed as number one below. The choices for output are :

1.) A listing of the processed code and cross reference will be displayed on the screen.

OR

2.) A listing of the processed code and cross reference will be sent to the line printer.

3.) Only the object lines (NO source) of the processed code and cross reference will be done in option 1 or 2.

OR

4.) NO processed code will be displayed but only the cross reference will be done in option 1 or 2.

OR

5.) No cross reference will be done in option I or 2.

6.) Full compression of the object code will occur.

7.) Variables labels, and directives WILL NOT be converted to upper case.

In the above list note that item 1 is the default option performed unless another option is switched in. Option 1 or 2 may be selected (either one or the other) and one of those is further modified by option 3, 4 or 5. Option 6 and 7 may be used with any combination. The following list will show the switches that are available to alter the processing options respective to the above list.

1.) <ENTER> - (for default) all processed code and cross reference will be produced and sent to the video display.

2.) LP - - - (for Line Print) all processed code and cross reference will be sent to the printer.

3.) TO - - - (for Text Only) only text lines and cross reference will be done in option 1 or 2.

4.) NL - - - (for No Listing) only a cross reference table will be generated in option 1 or 2.

5.) NX - - - (for No Xreference) no cross reference table will be generated in option I or 2.

6.) FC - - - (for Full Compression) to eliminates as many spaces as possible from the object file.

7.) DC - - - (for Differentiate Cases) will cause all variables, labels and directives to appear in the object code <u>exactly</u> as it was in the source with no conversion from lower case to upper case.

NOTE: Only one output device may be used to display the listing of the processed code. If the LP switch is set, then no listing will appear on the video display. If NL and TO are used together, the NL switch will take precedence, resulting in no source listing but a cross reference. If NL and NX are used together no video or printed output will occur.

Any allowable combination of the switches may be specified. By not specifying a switch the default for that particular output option will be used. If multiple switches are specified, they must be separated by commas. The following example should clarify how to properly answer the Processing Parameters prompt.

To have the output of the processed file sent to the printer and create a disk file which contains the processed program and in addition generate a cross reference table of variables and labels, answer the Processing Parameters prompt in the following manner.

```
LP <ENTER>
```

Specifying LP as an option will cause a listing to be sent to the printer (NOT to the video), and a cross reference table will be generated.

```
NL,NX <ENTER>
```

This combination will cause no video or printer output but will process the code and write the object to disk.

```
LP,NL,NX <ENTER>
```

This combination is identical to the preceding one.

<ENTER>
Pressing <ENTER> alone will cause the process code to appear on video followed by a cross reference on video.

The FC output option causes the processor to eliminate as many spaces as possible from the object code. If NOT specified all single spaces from the source file will remain intact, however, all groups of spaces will be reduced to a single space. For example, the following source line :

```
if  Credit.Limit# <  Current.Bal# then  gosub @Get.Hostages
```

With FC NOT specified the processor would render the line like this

```
IF CR# < CU# THEN GOSUB 128
```

Notice that the single spaces from source were left intact but that the spaces generated from reducing variables and labels were removed. With FC switched, the same source line becomes :

```
IFCR#<CU#THENGOSUB128
```

With FC on, the only spaces not removed will be in "REM" lines, between quote marks, and after the keyword "AS" if the following variable begins with "C".

The DC option will prevent the processor from converting lower case characters to upper case characters in variables,labels and directives. This means that a Variable called LOOP1% now would be unique from loop1% and from LOop1% etc. A label called @INPUT would be different from @input, @Input, @ InpuT etc.

The DC option should NOT be utilized unless the source code was written to accommodate it.

The last prompt will be

Directives :

This can be answered in one of two ways

<ENTER> - - - -	signifies no more Directives (or no Directives if this is the first time the prompt appears.
exp,exp,etc - - -	establishes conditional Directives which affect processing

To signify zero Directives merely press <ENTER>. Otherwise place proper Directives separated by commas. If directives exceed the input line width, enter as many as possible and press <ENTER>. The "Directives :" prompt will reappear. Input as many *expressions as desired. This prompt will be repeated until only the <ENTER> key is pressed as the first character of the line.

After all prompts have been answered, the disk drive will access and the message "Pass 1" will be displayed. It is at this time that the source file will be passed into memory and then written to the object file.

The source file is processed at this time (labels will be changed to line numbers, variable names will be compressed, etc.). For a more detailed description of the processing that occurs, refer to Section IV of the manual.

During the processing phase, the utility will perform a check to see that valid source code has been written. (Note: valid source code is not necessarily working BASIC code.) If no error is detected, the processing phase will continue and the object code created. As each pass begins a message "Pass X" will appear on the screen.

If an error is detected during processing, one of two things will happen. If it is a hardware error (e.g. Disk I/O Error, Parity Error, etc.), the utility will abort and will resume at the LDOS Ready prompt.

If the error exists within the source code (i.e. something in the source code cannot be processed), the appropriate error message will be displayed. The processing will be suspended at this time. Pressing <BREAK> will abort the processing, and return to the LDOS Ready prompt. Pressing any other key will continue the processing so that additional source code errors may be discovered. After the entire source code has been processed, control will return to the LDOS Ready prompt and no additional prompts will appear.

Processing Errors

The following is a list of processing errors that may be generated due to mistakes or errors within the source code. Whenever one of these errors is encountered, pressing any key will continue the processing and display additional errors (if any), while pressing <BREAK> will abort the processing routine and return to the LDOS Ready prompt.

Illegal Procedure Label

This error will be the result of specifying a label name that does not conform to valid label names.

Multiply Defined Label

This error will be generated when the same label is DEFINED more than once within the source code.

Illegal Variable

This error will indicate that a variable has been defined which does not conform to the variable name rules.

Variable Definition Format Error

This error will indicate that the proper syntax was not used in a line which defines variables.

Local Procedure Used without a RETURN

This error will indicate that a local procedure was defined, but had no RETURN statement ending it (remember that each procedure must have as its last statement RETURN, contained on a separate line).

Undefined Procedure Label

This error will indicate that a reference has been made to a label (either a GOTO, GOSUB or RESUME) and that label has not been defined.

Multiply Defined Global Variable

This error will indicate that a variable has been defined as global in more than one place.

Undefined Variable

This error indicates that a variable was encountered that had not been defined

Illegal Title Format

A Title Directive was encountered which did not follow the criteria for a correct title

Illegal Directive Format

A directive expression was used that did not follow the correct criteria for directives

The following errors will abort immediately if encountered.

Insufficient Memory to Load Text

Not enough RAM memory to process the source file.

Symbol Table Overflow

Either too many variables were used or there are too many references to those variables.

Source Line too Long

The length of a line exceeds the 240 character limit.

Variable usage Overflow

More than 930 variables of a certain declaration tag were used.

The following errors can only occur at the input prompts. If encountered, the prompt will be repeated.

Illegal Filespec

This error will occur if either the source or object filespec is not a proper file specification.

Bad *expression format

An illegal input format was used in response to the Directives prompt. Only the current input line will be cancelled.

Bad Parameter(s)

This error will result if the Processing Options prompt was not answered correctly.

   Identical Source and Object Filespecs

This is the result of naming the object file identically to the source file.

If appropriate, the above error messages will display the LINE NUMBER in which the error was encountered. For debugging purposes, it is recommended that numbered source code line list be obtained prior to processing. In order to do this at the LDOS READY prompt type:

```
     LIST filespec (N,P)
```

A printed output of the source file will be generated with an assigned "line number" to the left of each line. Refer to these numbers regarding error message line numbers.

Sample Screen and Video Output

The following represents a sample of the output that will be seen if a listing is to be displayed to the screen or sent to the printer. For the most part, this sample listing will appear the same, whether it is displayed on the screen or sent to the printer.

If this is the source file which is being processed:

```
           clear 1000
           = testvar%
     @beginning
           testvar%=0
           gosub @flash.message1
             if testvar%=1 then  goto @ending.mssg
           gosub @flash.message2
             if testvar%=1 then  goto @ending.mssg else  goto @beginning
     @flash.message1=kbdscan$,loop%
           kbdscan$=""
           cls:for loop%=1to20
                kbdscan$=inkey$
                  if  kbdscan$=chr$(13) then  goto @end.flash1
                  if  kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash1
           next  loop%
           print@512,"flashing  mssg 1 - enter for 2, x to abort"
           for loop%=1to50
                kbdscan$=inkey$
                  if  kbdscan$=chr$(13) then  goto @end.flash1
                  if  kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash1
           next  loop%
           goto @flash.message1
     @end.flash1
           return
     @flash.message2=kbdscan$,loop%
           kbdscan$=""
           cls:for loop%=1to20
                kbdscan$=inkey$
                  if  kbdscan$=chr$(13) then  goto @end.flash2
                  if  kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash2
           next  loop%
           print@512,"flashing  mssg 2 - enter for 1, x to abort"
           for loop%=1to50
                kbdscan$=inkey$
                  if  kbdscan$=chr$(13) then  goto @end.flash2
```

```
              if  kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end-flash2
         next  loop%
         goto @flash.message2
      @end.flash2
         return
      @ending.mssg
         cls:print@512,"this program has been run in its   entirety.":end
```

This is the output that will be generated to the screen/printer (with the FC parameter specified

```
      clear 1000
    ====>> 1clear1000

      =testvar%
    ====>>

      @beginning
    ====>>

      testvar%=0
    ====>> 4TE%=0

      gosub @flash.message1
    ====>> 5gosub10

      if testvar%=1 then  goto @ending.mssg
    ====>> 6ifTE%=1thengoto42

      gosub @flash.message2
    ====>> 7gosub26

      if testvar%=1 then  goto @ending.mssg else  goto @beginning
    ====>> 8ifTE%=1thengoto42elsegoto4

      @flash.message1=kbdscan$,loop%
    ====>>

      kbdscan$=""
    ====>> 10KB$=""

      cls:for loop%=1to20
    ====>> 11cls:forLO%=1to20

      kbdscan$=inkey$
    ====>> 12KB$=inkey$

      if kbdscan$=chr$(13) then  goto @end.flash1
    ====>> 13ifKB$=chr$(13)thengoto24

      if kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash1
    ====>> 14ifKB$="X"orKB$="x"thenTE%=1:goto24

      next  loop%
    ====>> 15nextLO%

      print@512,"flashing  mssg 1 - enter for 2, x to abort"
    ====>> 16print@512,"flashing  mssg 1 - enter for 2, x to abort"

      for loop%=1to50
    ====>> 17forLO%=1to50
```

```
   kbdscan$=inkey$
====>> 18KB$=inkey$

  if kbdscan$=chr$(13) then  goto @end.flash1
====>> 19ifKB$=chr$(13)thengoto24

  if kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash1
====>> 20ifKB$="X"orKB$="x"thenTE%=1:goto24

  next  loop%
====>> 21nextLO%

   goto @flash.message1
====>> 22goto10

  @end.flash1
====>>

  return
====>> 24return

  @flash.message2=kbdscan$,loop%
====>>

   kbdscan$=""
====>> 26KC$=""

   cls:for loop%=1to20
====>> 27cls:forLP%=1to20

   kbdscan$=inkey$
====>> 28KC$=inkey$

  if kbdscan$=chr$(13) then  goto @end.flash2
====>> 29ifKC$=chr$(13)thengoto40

  if kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash2
====>> 30ifKC$="X"orKC$="x"thenTE%=1:goto40

  next  loop%
====>> 31nextLP%

  print@512,"flashing  mssg 2 - enter for 1, x to abort"
====>> 32print@512,"flashing  mssg 2 - enter for 1, x to abort"

  for loop%=1to50
====>> 33forLP%=1to50

   kbdscan$=inkey$
====>> 34KC$=inkey$

  if kbdscan$=chr$(13) then  goto @end.flash2
====>> 35ifKC$=chr$(13)thengoto40

  if kbdscan$="X" or  kbdscan$="x" then testvar%=1:goto @end.flash2
====>> 36ifKC$="X"orKC$="x"thenTE%=1:goto40

  next  loop%
====>> 37nextLP%
```

```
   goto @flash.message2
 ====>> 38goto26

   @end.flash2
 ====>>

   return
 ====>> 40return

   @ending.mssg
 ====>>

   cls:print@512,"this program has been run in its   entirety.":end
 ====>> 42cls:print@512,"this program has been run in its   entirety.":end
```

```
Procedure Label      Defn #   Line #     Referenced at Line #'s
---------------      ------   ------     ----------------------
@beginning             3        4        8
@flash.message1        9        10       5,22
@end.flash1            23       24       13,14,19,20
@flash.message2        25       26       7,38
@end.flash2            39       40       29,30,35,36
@ending.mssg           41       42       6,8

Variable Label       Defn      XLATE     Referenced at Line #'s
--------------       -----     -----     ----------------------

kbdscan$ *             9        KB$      10,12,13,14,14,18,19,20,20
kbdscan$ *             25       KC$      26,28,29,30,30,34,35,36,36

testvar%               2        TE%      4,6,8,14,20,30,36
loop% *                9        LO%      11,15,17,21
loop% *                25       LP%      27,31,33,37
```

When a listing is being displayed on the screen it will scroll. To temporarily pause the listing use the <SHIFT><@> pause keys. If the listing is paused, pressing any key will resume the listing.

The information which is displayed in a listing should be quite easy to understand. Each line (sequentially) in the source code will be displayed. Following the source line will be the object line which it was translated into during the processing. The first line of source code (clear 1000) was translated into:

```
1clear1000
```

Object lines will be denoted by a "====>>" symbol preceding the line. Note that the second and third source lines (= testvar% and @beginning) were displayed in the listing, but did not get translated into anything during the processing. Label and variable definition statements will never be translated into executable object code. The line following these two definition statements (testvar%=0) is where the translation process continues. This line was translated into:

```
4TE%=0
```

Note that lines in the object code will be numbered consecutively starting from line number 1 and that lines which do not exist in the object code are also numbered, but that the number is missing after processing. Since there are 42 lines of source code that can be translated into executable object code, the object file consists of lines 1-42 with line numbering skipping all deleted lines.

Following the last line of the source/object code translation is a sample cross reference listing. The first part of the table gives the translation that was performed on labels. In the left most column of the table is each label name that was defined and referenced.

The next column will give the line number of the object code representing the translation performed on the label. For example, the @beginning label was translated into line 4 in the object program, which is represented by the first executable object statement following the label definition (in this case, line 4 is the first executable statement following the label definition).

The last column in the label cross reference part of the table lists all line numbers in the object code which reference the given label. Note from looking at the listing of the processed code that the @beginning label was referenced in the source line that corresponds to object line 8 (i.e. goto @beginning). Note also that object line 8 contains the statement GOT04, which is the line in the object code that represents the label @beginning.

The second part of the cross reference table shows the variable translations that took place. All variables that are defined in the source code (either globally or locally) will be listed in this table. The first column in the variable translation table will list all variables as defined and used in the source code. Any variable defined as local will have an asterisk <*> following the variable name. Note also that the variables kbdscan$ and loop% appear twice in the table. This is because they represent local variables used in two separate procedures, and have been translated into two distinct variables.

The second column of the variable cross reference table (XLATE) gives the variable name in the object code that the variable in the source code (listed in the first column) was translated into. From the table, it can be seen that the global variable testvar% was translated into TE%. Also, the variable kbdscan$ was translated into KB$ in the first procedure that it was used in, and KC$ in the second procedure.

The last column gives the line numbers in object code corresponding to where these variables are referenced. The variable TE% (testvar%) appears in lines 4,6,8,14,20,30 and 36 of the object code.

Although it cannot be seen in this cross reference table (due to the program being relatively short), variables in a cross reference table will be grouped in a special manner. String variables will be listed first, followed by integer, single precision and double precision. Within these subgroups of variables, the following order will be maintained. All global variables will appear first within a subgroup, and will be listed in alphabetical order (according to the variable name used in the source code). After the global variables will come the local variables, which will again be alphabetized. This type of grouping will make it easy to pick out specific variables within the table.

Section IV - How the Processor Operates

This section of the manual will detail the operations that are performed by the processor to change source code into object. All information necessary to write source code is detailed throughout this manual. This section of the manual is being provided solely to explain the more subtle aspects of TBA. It may not be necessary to read this section of the manual. However, in order to utilize the processing utility to its fullest extent, read through this section. It is only through a full understanding of the processing which takes place that the most efficient source code will be written.

The processing of source code into object code is performed in a series of steps. In each of these steps, a "pass" is performed on the source code. The term "pass" is used to mean the operation in question is being performed on current interim code. Each particular pass will alter the code changing it from the source code into executable object code and place the interim results in the object filespec. A pseudo object code is written to disk under the filespec specified as the object filespec and is subsequently during the succeeding passes.

There are six passes that are performed on the source code. Each pass will perform a specific processing function to change the source code into object code. During these passes, tables are created and maintained. These tables are used to store information about labels, directives and variables defined and to store the cross-reference information for labels and variables. The following will describe briefly the actions that occur during each pass.

Pass #1

Any pertinent text information from the source file is written to the disk using the object filespec. As it is being read in, several processing functions are performed. Any line numbers that exist at the beginning of the line (if the source was written in BASIC) are stripped off. Line numbers are assigned to all text lines, starting with line number one, being incremented by one. As labels are picked up, a check is done to see whether or not the label has already been defined. If a multiply defined label is detected, the proper error message will be displayed.

The line number that will be assigned a label definition statement will be the next text line encountered.

Pass #2

In pass #2, all local variables are evaluated and processed. This is done by going through the list of labels that signify procedures and changing the local variables to two character variables. Only variables that have been defined as local will be affected.

The processor maintains a table of two character variable names that have already been used. The first two characters in the local variable are examined and the processor performs a check to see whether or not that variable name has been used. For example, if the local variable test.variable% is encountered, the processor will attempt to change this variable to the two character variable TE%. If TE% already exists, TF% will be used (if it has not already been used).

Each procedure is processed individually, starting at the procedure definition line and ending with the associated RETURN. This is the reason that the same local variable may be used in two different procedures and will be translated into two different variables. For instance, if the local variable   test.variable% is used in two different procedures, it will be translated into TE% in the first procedure and TF% in the second.

Passes #3 & #4

In passes 3 and 4, all global variables are translated. Only variables that have been defined as global will be affected. The entire source code is examined and any match of a global variable that has been defined will be translated into a two character variable name. The process used is very similar to that of translating local variables. Since local variables are translated first, it is permissible to use the same variable name to represent a local and global variable. For instance, if the variable test.variable% is defined to be global and is also defined in a local procedure, it will be translated into TE% in the local procedure and TF% everywhere else in the program (since local variables are processed first.

To use a global variable in a procedure, do not define it as local, then it will not be translated during the processing of local variables (since it was not defined to be local). It will be translated when the global variables are processed and thus will be assigned the same variable name that is used to represent the global variable throughout the program.

Pass #5

In pass 5 all references to labels are changed to reference the line number associated with the label. This is done by going through the table containing the label definitions (and associated line number to be translated into) and translate all occurrences of the label reference in the source code to the corresponding line number. Any text lines will have the extraneous spaces removed.

Pass #6

In pass 6 (the final pass) the source code is partially or fully compressed and any output options (either a listing to the printer/video or the creation of an object file on disk) are performed. If FC has been specified all spaces (with the exception of those found within quotes, REM statements and some others) are compressed out and any characters that are found to have the high bit set will have this bit reset to produce pure ASCII text. The text which is a result of this processing is the object code.

Section V - General Operational Guidelines & Program Maintenance

This section of the manual will detail the all-around programming environment that should be used when maintaining programs written under TBA. In addition, several points that could not be conveniently brought up in other sections of the manual will be discussed here. This section will end with a listing and explanation of some sample files and exercises. These source and object files are contained on the Master diskette that accompanies this manual. They are provided for the user to inspect and study, so that user programs may be written with the structure and efficiency allowed by TBA.

Use of Error-Trapping Routines

Error trapping routines play a major role in any well written BASIC program. Because of the nature of the processing which takes place, a few special considerations are needed when constructing the error handling routines.

All ONERRORGOTO statements should be written to reference a label (e.g. ONERRORGOTO @error.routine). The entry point into the error trapping routine will have as its first line the label definition. To resume to a specific line from an error trapping routine, use the RESUME command, followed by the label which represents the point of return. The following example will illustrate how to properly establish error trapping routines.

```
   'branch to @error.detected if an error is encountered.
   '
   ONERRORGOTO @error.detected
   '
   'program code
   '
   '
   '
   'end of program code
   '
   '
   'definition of @error.detected routine
   '
   '
   @error.detected
   '
   'error trapping code
   '
   '
   'resume after error at line defined by the label @ error.done
   '
   RESUME @error.done
```

Please note that the use of a label to define an error trapping routine functions the same as referencing a line in the program with GOTO. One point that must be stressed is the fact that since error trapping routines are terminated using a RESUME (and not a RETURN), local variables should NOT be used in error trapping routines. Remember, the processing utility will only identify RETURN as a proper means to end a procedure (and the definition of local variables used with the procedure). Using local variables in an error trapping routine which is terminated by use of the RESUME statement will cause an error to occur during processing.

Throughout the manual it has been pointed out that absolute line numbers should never be used in the source code. There is one exception to this rule and that is the statement ONERRORGOTO 0. The ONERRORGOTO 0 statement may still be used within the source code, as its function is not to denote a point of transfer, but rather to "turn off" any active ONERRORGOTO statement.

Maintaining Programs

When maintaining programs created by TBA (either fixing bugs or adding new features), changes to the program should be made to the SOURCE code only and NOT the object code. The reasons for this should be quite evident. The source code (if written properly) should be much easier to follow because it is written in a descriptive dialogue. The object code is more difficult to follow, since labels have been changed to line numbers, variable names are only two characters and possibly no extraneous spaces exist. One of the main reasons for using TBA is ease of program readability. Performing edits to the object code instead of the source code will obviate the merit of writing source code.

More important, is the concept of having only one program which represents the current version. Suppose, for example, that there was a slight error in the program, which was corrected by modifying the object code. If it was decided to add an additional feature to the program (done by editing the source), it should be easy to see that the source code would not reflect the original change made to the object code. To ensure that the source file always contains the latest version of the program, perform ALL edits on the source code. After the appropriate edits have been made, the source file may be processed to create object code reflecting these changes.

Enhancing Program Operation and Speed

This section will list some suggestions that can be implemented in source code to improve program execution time.

When object files are created by the processing utility, they are written to disk in pure ASCII. No compression codes are used to represent BASIC keywords. To decrease the time required to load an object file, it is suggested that the program be LOADed and SAVED to disk. The program will then be saved using compression codes for all BASIC keywords and any subsequent load of the program will be performed much faster.

To write code that will execute faster, there are two suggestions. In order to use a variable (either locally or globally), it must be defined. However, defining a variable in the source code, whether local or global, will NOT initialize the variable in the object program. BASIC does allow initializing variables "on the fly" (as they are needed). Utilizing variables in this manner does tend to slow the operation of the program. The variable has to be established in BASIC's variable table prior to being used. If all GLOBAL variables are initialized prior to actually using them, the speed of the program will be increased, because these variables will already exist in the variable table prior to being needed. Once a variable has been initialized and placed in the variable table, the time required to access this variable will be diminished. Due to the nature of the processing utility, only global variables may be initialized.

The following example of source code will illustrate the proper method to initialize global variables. All variables in the example below will be initialized to zero. Variables can be initialized to any value, particularly to contain a specific value as the program is entered.

```
    'Source Code variable definition statement
    '
    =testvar%, delay.loop1%, delay.loop2%,  total.items%,  total.dollars#
    '
    'variable initialization statements
    '
    testvar%=0:delay.loop1%=0:delay.loop2%=0:total.items%=0:total.dollars#=0
```

In dealing with a small number of global variables, the increase in operating speed will be insignificant. However, when using many global variables, initializing them at the beginning of program execution will eliminate any unexpected delays when a variable is initially encountered in the program.

There is a trade-off to be considered when initializing variables. There will be a noticeable delay encountered when the variable initialization statements are executed (provided many variables are being defined). In the long run, it is better to encounter such a delay before the actual program execution begins rather than encounter the delay during execution.

Foresight should be used in setting up the physical variable initialization statements. Global variables that will be used frequently throughout the program should appear at the beginning of the variable initialization statement. This is because BASIC establishes a variable table when a variable is first encountered in a program. Every time a variable is accessed, a scan is done of the variable table. If a frequently used variable is the first variable to appear in the initialization statement, the time it takes BASIC to scan the variable table will be decreased because it will locate the variable at the beginning of the variable table.

BASIC can perform operations much faster on integer (%) variables as opposed to single or double precision. If possible, variables used in a loop (in conduction with FOR / NEXT) should be integer type. Since integer values are stored in two bytes (as oppposed to single precision which are stored in four bytes and double precision which are stored in eight), operations using them will be processed much quicker. If decimal numbers are required, they should be contained in single precision variables wherever possible. Double precision variables will be processed much slower than either integer or single precision.

In string variables, unfortunately, the "garbage collecting" routines used by BASIC will inevitably slow up processing. One suggestion to speed up program operation when dealing with strings is to initialize all string constants that will be assigned to variables at the beginning of the program in the same manner that numeric variables were initialized. Once initialized to represent a string constant, these variables should NOT be used to store any other string information (such as string information input from the keyboard or a disk file). The string which is represented by the variable will be referenced by its actual location in the program text in RAM. String variables used in such a manner will never be included in the garbage collecting routine.

Source code should be designed to consist of a program Main body which references procedures to perform various tasks. The tasks that these procedures will perform will be decided totally on programming requirements. However, the structure used to write the program may speed up its operation.

Consider how the BASIC interpreter functions to perform branch operations. Every time a backward branch is executed (either with a GOTO or a GOSUB), the program text is scanned, starting at the beginning line until the line to branch to is encountered. Since this is the case, it makes sense that frequently referenced routines (such as an input procedure) should appear at the beginning of the program text. To allow this, perform a branch to the Main body of the program, which will bypass the procedures that are located at the beginning of the program text. If there are many global variables to be initialized, there will still be substantial program code in front of these procedures. Since the initialization of global variables is a one time operation, it is better to perform the initialization of variables at the end of the program text. The following will represent such a program structure. In many cases the following program layout will increase the speed with which a program will execute.

1.) Global variable definition statements (will NOT appear in processed code)
2.) Branch statement to the global variable initialization (step #6)
3.) Procedures which are accessed frequently
4.) Main Body of the program
5.) Procedures (such as error trapping) which are accessed infrequently.
6.) Global variable initialization statements
7.) Branch statement to the Main body of the program (step #4)

Summary

The point stressed above, is not an imposition of a specific program structure, but rather the idea of consistency. In the final analysis, YOU will be the person writing the BASIC program. Whether to use (or even consider to use) the stated concepts in writing programs is a private decision. What is important is to develop consistent programming methods and techniques. It may take a great deal of forethought to determine the method to use when writing a program. However, once having established an acceptable method for writing programs, stick with it! It will be found that programming (and debugging) time required to produce a program will be diminished drastically.

Following is a listing of some BASIC programs (both source and object code). The actual text for the programs can be found on the master diskette. Experiment with these files prior to writing source code, so that a feeling for how the processor functions and the results that are obtained as a result of the processing operation will be fluently learned.

Below is a typical small uncomplicated BASIC source. The name of the source file on the master disk is FACTOR/TBA.

```
    *TITLE"EXERCISE 1"
     'global definition statements are below
    =FACTOR$,LOOP1!,LOOP2!,HALF!,HORIZ.LINE$,START!,END!,INPUT$
     =PRIME.FLAG%,PRIME.ARRAY!,PRIME.FACTOR$,COUNTER%,START2!
     =LOOP.COUNT%,PRIMES.FOUND%
     'dimension and clear statements; note that both array variables are defined above
    DIM PRIME.ARRAY!(10),PRIME.FACTOR$(10)
     CLEAR1000
     'program execution start in case subroutines are placed here at a later date
@START
     CLS
     INPUT "ORIGIN OF SCAN"; INPUT$
       'all IF statements are indented +2 from tab mark to set them apart
       IF VAL(INPUT$)<2 THEN @START
     START!=INT(VAL(INPUT$))
     INPUT "   END OF SCAN"; INPUT$
     END!=INT(VAL(INPUT$))
       IF END!<START! THEN T=START! : START!=END!: END!=T
     HORIZ.LINE$=STRING$(63,61): CLS
     FOR LOOP1! = START! TO END!
                'statements contained within a FOR/NEXT loop are tabbed over for clarity
         HALF!=LOOP1!
          FACTOR$=""
          ?@0, "factoring "USING"###,###";LOOP1!;
          ?@32,"primes found on this  scan"USING"##,###";PRIMES.FOUND%;
          ?@64,"prime factors : "CHR$(30)
          ?HORIZ.LINE$;: START2!=2
@RE.LOOP
     'loop within a loop is tabbed in further
           FOR LOOP2! = 2 TO HALF!
                  IF HALF!/LOOP2! = INT(HALF!/LOOP2!) THEN GOSUB @GOT.ONE: GOTO @RE.LOOP
            NEXT LOOP2! : IF VAL(FACTOR$) = LOOP1! THEN FACTOR$="* Prime Number *"
          :PRIMES.FOUND%= PRIMES.FOUND%+1 ELSE FACTOR$=LEFT$(FACTOR$,LEN(FACTOR$)-1)
          PRIME.ARRAY!(COUNTER%)=LOOP1!
          PRIME.FACTOR$(COUNTER%)=FACTOR$
          LOOP.COUNT%=COUNTER%
            FOR LOOP2!=0 TO 10
               ?@64*LOOP2!+192,PRIME.ARRAY!(LOOP.COUNT%),PRIME.FACTOR$(LOOP.COUNT%);
CHR$(30);
               LOOP.COUNT%=LOOP.COUNT%-1
                  IF LOOP.COUNT%=-1 THEN LOOP.COUNT%=10
               NEXT LOOP2!
          COUNTER%=COUNTER%+1
          IF COUNTER%=11 THEN COUNTER% =0
     NEXT LOOP1!
     END
'procedure at end of main body note that this is a subroutine

@GOT.ONE
     FACTOR$=FACTOR$+STR$(LOOP2!)+" x"
     ?@80,FACTOR$;
     HALF!=HALF!/LOOP2!
     RETURN
```

If the processed output is to be directed to a printer, the printer should be prepared for output. A top of form should be sent to the printer at this time to ensure proper pagination. To do so, enter the following command at the LDOS Ready prompt.

```
LBASIC LPRINT CHR$(12) :CMD"S"
```

This will cause the printer to advance the paper to its top of form position. If the paper is situated so that the first line of print will appear at the start line of the paper, then do nothing, otherwise, take the printer off line and manually advance to the desired position.

If you are using an 80 column printer, you may want to use PR/FLT to allow for proper paging. Normally, TBA will assume that 132 characters are to be printed per line. With an 80 column printer, this may cause wrap-around, and paging could be affected. To allow proper paging on an 80 column printer, enter the following command:

```
FILTER *PR  PR (CHARS=80)
```

NOTE: If you have a full LDOS system, the top of form may be accomplished by using the MINIDOS/FLT command -- <CLR><SHIFT><T>.

If the printer is ready, the processed code can be directed to it. If no printer is available, process to the video. See if you can easily predict the function of this program prior to running it. It should be a lot easier to determine the results when viewing this listing, as opposed to viewing a normal listing.

To process the above file into a working BASIC program, enter the following command.

```
TBA <ENTER>
```

This will cause the processing operation to begin. The first prompt that you will see will be for the Source file. Answer this prompt by entering the following.

```
FACTOR<ENTER>
```

The disk will access as TBA looks for a file called "FACTOR/TBA". Entering the entire filespec would also be allowed. The above command illustrates how the defaults can be used when specifying a filespec.

The next prompt to appear will be for the Object   filespec. You may answer this prompt by pressing <ENTER>. This will cause TBA to use "FACTOR/BAS" as a filename both for the processing operation and the finished file.

The next prompt to appear will requesting information dealing with processing   parms. If you want the processed stream to be sent to the video, answer this prompt with <ENTER>. Otherwise, if you want the listing to be sent to the Printer, answer the Processing parms prompt with the following.

```
LP <ENTER>
```

The next prompt to appear is requesting information on any directives to be acted upon. Since the source file does not contain any processing expressions, this prompt may be answered by pressing <ENTER>.

At this time, disk access will begin, and as each pass starts, a message will appear on the screen. Shortly after the Pass 5 message appears, the screen or printer processing will take place and the final object file will be written to the diskette.

You may now wish to view the resultant BASIC program. To do so, enter the following command LBASIC LOAD"FACTOR".

Since TBA creates an ASCII file, none of the compression codes used by BASIC are stored in the file. It is totally permissible to load in a BASIC program stored in ASCII (as opposed to being saved with compression codes). However, the time it takes to load a program which is stored in ASCII is greater than the time it takes to load a program which is stored in compressed form. For this reason, you may wish to "Re-save" the program, so that it will be stored on the diskette in compressed form. Doing so will cause subsequent loads of the program to be performed faster. To save the program in compressed form, enter the following command from the BASIC Ready prompt.

```
SAVE"FACTOR"
```

At this time you may wish to LIST the program and draw comparisons between the source code and the object code. Note that there is a variable defined in source code that is not used. By inspecting the cross reference table, you should be able to locate this variable.

If the processing operation was completed successfully, the resulting object code should appear as follows.

```
7 DIM PR!(10),PR$(10)
8 CLEAR1000
11 CLS
12 INPUT "ORIGIN OF SCAN"; IN$
14 IF VAL(IN$)<2 THEN 11
15 ST!=INT(VAL(IN$))
16 INPUT "   END OF SCAN"; IN$
17 EN!=INT(VAL(IN$))
18 IF EN!<ST! THEN T=ST! : ST!=EN!: EN!=T
19 HD$=STRING$(63,61): CLS
20 FOR LO! = ST! TO EN!
22 HALF=LO!
23 FA$=" "
24 PRINT@0, "factoring "USING"###,###";LO!;
25 PRINT@32,"primes found on this  scan"USING"##,###";PS%;
26 PRINT@64,"prime factors : "CHR$(30)
27 PRINTHD$;: SU!=2
30 FOR LP! = 2 TO HA!
31 IF HA!/LP! = INT(HA!/LP!) THEN GOSUB 48: GOTO 30
32 NEXT LP! : IF VAL(FA$) = LO! THEN FA$="* Prime Number *" :PS%= PS%+1 ELSE
FA$=LEFT$(FA$,LEN(FA$)-1)
33 PR!(CO%)=LO!
34 PR$(CO%)=FA$
35 LO%=CO%
36 FOR LP!=0 TO 10
37 PRINT@64*LP!+192,PR!(LO%),CHR$(30)PR$ (LO%)
38 LO%=LO%-1
39 IF LO%=-1 THEN LO%=10
40 NEXT LP!
41 CO%=CO%+1
42 IF CO%=11 THEN CO% =0
43 NEXT LO!
44 END
48 FA$=FA$+STR$(LP!)+" x"
49 PRINT@80,FA$;
50 HA!=HA!/LP!
51 RETURN
```

Running the program will demonstrate its **prime**-ary purpose depending on the **factors** involved. How successful was the prediction of output?

The next section will demonstrate how existing procedures might be combined into a main body program. Two procedures will be merged into source code. Since there are two methods of doing this, the first section will outline the step by step process under the LSCRIPT word processor and the second under LBASIC.

Merging Procedures with LSCRIPT

Below is a listing of a procedure which is contained on your TBA Master diskette. The name of the file is CENTER/SCR. It can be viewed by loading it into LSCRIPT.

```
     'procedure to center a string variable given the
     'column width and device
     '
     'merge into any TBA source file & define CENTER$ as a Global
     'define WIDTH% and DEVICE% as Global
     'TO USE: define desired string as CENTER$
     'define DEVICE% as 0 for video OR 1 for printer
     'set WIDTH% to total columns and GOSUB @CENTER.DISPLAY
     @CENTER.DISPLAY=FROM.LEFT%,WIDTH1%,LENGTH%
     @TOO.LONG:LENGTH%= LEN(CENTER$): WIDTH1%=INT(WIDTH%/2)
             IF LENGTH% > WIDTH% THEN CENTER$=MID$(CENTER$,(LENGTH%-WIDTH%)/2,WIDTH%)
          :GOTO @TOO.LONG
          FROM.LEFT%= WIDTH1%-INT(LENGTH%/2)
             IF DEVICE% < 0 OR DEVICE% > 1 THEN DEVICE%=0
          ON DEVICE% +1 GOTO @SCREEN,@PRINTER
     @SCREEN:?TAB(FROM.LEFT%)CENTER$: GOTO @EXIT.CENTER
     @PRINTER: LPRINTTAB(FROM.LEFT%)CENTER$
     @EXIT.CENTER:RETURN
```

The first eight lines serve as comments to document the procedure. They describe the variables that need to be defined as global, the function of the procedure, the parameters required and the calling method. Take note of the   globals to be defined and the procedure name for use when writing the main body.

```
PROCEDURE = @CENTER.DISPLAY
GLOBAL = WIDTH%,CENTER$,DEVICE%
METHOD = SET TARGET $ TO CENTER$: DETERMINE WIDTH% & OUTPUT DEVICE% : GOSUB
```

Also included on your Master TBA diskette is a file named INPUT/SCR. By viewing this file in LSCRIPT, the following notations can be made.

```
PROCEDURE = @INPUT
GLOBAL = INP$,AT%,FIELD%
METHOD = SET SCREEN POSITION TO AT%: MAX INPUT CHARS TO FIELD%: GOSUB : INPUT RETURNED
         IN INP$
```

In this exercise, the Source file DEMO/SCR will be created, and will utilize these pre-existing procedures. The results of this program will be to input information into a string array and print it to the center of the screen or printer. The @INPUT procedure acts as an input subroutine, used to enter information into the string array. The @CENTER.DISPLAY procedure will center the information when sending it to the desired output device.

The following listing is the DEMO/SCR file. You will need to use LSCRIPT to create this file. To facilitate your input, you may wish to set tabs at locations 5, 10 and 15.

```
'exercise 2 main body program : filename DEMO/SCR
=INP$,AT%,FIELD%,CENTER$,WIDTH%,DEVICE%,STRING.ARRAY$,LOOP%,LOOP1%
     CLEAR 2000
     DIM STRING.ARRAY$(50)
     GOTO @MAIN
' INPUT PROCEDURE WILL GO HERE
'CENTER PROCEDURE WILL GO  HERE
@MAIN
     CLS:?"Enter a sentence of less than 50 words"
     ?STRING$(63,61)
     ?@128,"Enter @ to stop"
     FOR LOOP% = 0 TO 50
            IF LOOP%>0 THEN ?@192,CHR$(30)"Last Entry "STRING.ARRAY$(LOOP%-1)
          ?@320,"Current Entry ="
          AT%=336:FIELD%=10:GOSUB @INPUT
            IF INP$ = "@" THEN @DISPLAY.VIDEO
          STRING.ARRAY$(LOOP%)=INP$
          NEXT LOOP%
@DISPLAY.VIDEO
     WIDTH%=64:DEVICE%=0
     CLS: FOR LOOP1%= 0 TO LOOP%-1
          CENTER$= STRING.ARRAY$(LOOP1%)
          GOSUB @CENTER.DISPLAY
             IF LOOP1%>0 AND LOOP1%/14 = INT(LOOP1%/14) THEN GOSUB @WAIT.FOR.ENTER
          NEXT LOOP1%
     END
@WAIT.FOR.ENTER
     ?@960,"Press <ENTER> to continue";
@AGAIN:AT%=985: FIELD%=1: GOSUB @INPUT
       IF INP$ <> "" THEN @AGAIN
     CLS
     RETURN
```

After you have entered this text, you will want to save it to disk. To do so, enter the following commands: (Note that the file will be saved as DEMO/SCR, because LSCRIPT employs a default extension of /SCR when performing loads and saves).

```
<SHIFT> <ENTER> -- This will display the Special Command prompt
<L> <,> <C> <Space> <D> <E> <M> <O> <ENTER>
```

Once the file has been created and saved, the merging of the procedures may be done. This is accomplished by performing a Load and Chain of each procedure. First, Load and Chain the @INPUT procedure by entering the following command at the Special Command prompt.

```
<L> <,> <C> <Space> <I> <N> <P> <U> <T> <ENTER>
```

The @INPUT procedure will now be added to the end of the text currently in memory (the DEMO/SCR text).

Similarly, you will need to Load and Chain the @CENTER.DISPLAY source code. To do so, enter the following at the Special Command prompt.

```
<L> <,> <C> <Space> <C> <E> <N> <T> <E> <R> <ENTER>
```

After both procedures have been merged in, the source file could be processed, and the resultant object file executed. Note that since line numbers are not used, procedures could virtually appear anywhere within the source file. However, when writing source code in a word processing atmosphere, blocks of code can easily be moved around. Note in the main body of the program (DEMO/SCR) the remark statements indicating the place where these subroutines are to be inserted.

The following will describe the procedure used to move these blocks of code (procedures) when operating under the LSCRIPT environment. The first thing you will need to do is position the cursor over the "@" symbol in the @INPUT label. To move this text into the main body of the program, you will need to define it as a block. To do so, enter the following commands.

```
<CLR> <5> - This will establish the Block Command Mode.
<A>       - This will be the name of the block.
```

A block mark followed by "A>" will be inserted BEFORE "@INPUT". You will now need to position the cursor to the end of text. This can be done by either depressing the <DOWN ARROW> key (to scroll through the text), or by depressing simultaneously the <SHIFT><DOWN ARROW><Z> keys which represent the END control function in LSCRIPT.

After the cursor has been positioned to the end of text, enter the following commands (to signify the end of Block A).

```
<CLR> <5> - Block Command Mode
<SHIFT> <DOWN ARROW> <Z> - Denote the END of the block.
```

The end block marker should appear. Once the block has been identified, it may be inserted anywhere within the text. To insert it in the proper place, position the cursor in the main body of the source code program directly on the "'" in the line "' INPUT PROCEDURE WILL GO HERE". Then key in the following sequence:

```
<CLR><1> <CLR><5> - This will allow the insertion of a block
```

The prompt "NAME OF BLOCK TO INSERT" will appear on the bottom of the screen. Answer this prompt by typing in the letter A (the name that was assigned to the block). If the procedure was handled correctly both the INPUT and CENTER procedures will be inserted into the text immediately following the cursor.

Note that when you insert blocks, after the insert process, the same code will appear twice within your text (once as the original block, and once as the block that was inserted). At this point, you will want to delete the original block. To do so, position the cursor over the block marker to the left of the "A" and enter the following commands.

```
<CLR><3>
```

The prompt "DELETE OR UNMARK BLOCK (D or U)" will appear at the bottom of the screen. Answer this prompt by typing D to delete the block.

At this point, you will want to save your source code to disk.

WARNING: After a chain load, the filename used by LSCRIPT will be the same as the last file chained. Thus, in the above example, the current filename as seen by LSCRIPT would be <u>CENTER/SCR, NOT DEMO/SCR</u>. When saving any file back to disk after performing a load and chain, a filespec should be included with the save command. Using a default filespec with the save command will overwrite the last file chained with the text currently in memory.

You may wish to process the file and note the results. Although the end program is somewhat useless, the procedure is clearly illustrated. Entire programs can be merged out of many procedures in much less time then recomposition in ordinary BASIC.

Merging Procedures with LBASIC

Below is a listing of a procedure which is contained on your TBA Master diskette. The name of the file is CENTER/TXT. It can be viewed by loading it in as a BASIC program.

```
10 'Procedure to center a string variable given the
20 'column width and device
30 '
40 'merge into any TBA source file & define CENTER$ as a Global
50 'define WIDTH% and DEVICE% as Global
60 'TO USE: define desired string as CENTER$
70 'define DEVICE% as 0 for video OR 1 for printer
80 'set WIDTH% to total columns and GOSUB @CENTER.DISPLAY
90 @CENTER.DISPLAY=FROM.LEFT%,WIDTH1%,LENGTH%
100 @TOO.LONG:LENGTH%= LEN(CENTER$): WIDTH1%=INT(WIDTH%/2)
110        IF LENGTH% > WIDTH% THEN
CENTER$=MID$(CENTER$,(LENGTH%-WIDTH%)/2,WIDTH%) :GOTO @TOO.LONG
120      FROM.LEFT%= WIDTH1%-INT(LENGTH%/2)
130         IF DEVICE% < 0 OR DEVICE% > 1 THEN DEVICE%=0
140      ON DEVICE% +1 GOTO @SCREEN,@PRINTER
150 @SCREEN:?TAB(FROM.LEFT%)CENTER$: GOTO @EXIT.CENTER
160 @PRINTER: LPRINTTAB(FROM.LEFT%)CENTER$
170 @EXIT.CENTER:RETURN
```

The first eight lines serve to document the procedure. They describe the variables that need to be defined as global, the function of the procedure, the parameters required and the calling method. Take note of the globals to be defined and the procedure name for use when writing the main body. Also note line numbers used in the procedure. Since line numbers are used in interpretive BASIC, the program plan for the main body should include an area reserved for procedures to be merged in. If you have a means to renumber BASIC programs, line numbers are not extremely important in the procedures themselves, as the lines in the procedure can be renumbered to "fit into" the main body of the program. (NOTE: The examples in this section show the renumbering procedure by use of the CMD"N" command. This feature is only available when using a standard LDOS operating system.)

```
PROCEDURE = @CENTER.DISPLAY
GLOBAL = WIDTH%,CENTER$,DEVICE%
METHOD = SET TARGET $ TO CENTER$: DETERMINE WIDTH% & OUTPUT DEVICE% GOSUB
LINE NUMBERS TO BE USED IN MAIN BODY = 200 - 299
```

Also included on your Master diskette is a file named INPUT/TXT. By viewing this file in BASIC, the following notations can be made.

```
PROCEDURE = @INPUT
GLOBAL = INP$,AT%,FIELD%
METHOD = SET SCREEN POSITION TO AT%: MAX INPUT CHARS TO FIELD%: GOSUB INPUT RETURNED
         IN INP$
LINE NUMBERS TO BE USED IN MAIN BODY = 100 - 199
```

In this exercise, the Source file DEMO/TBA will be created. The results of this program will be to input information into a string array and display it to the center of the screen or printer. The @INPUT procedure acts as an input subroutine used to enter information into the string array. The @CENTER.DISPLAY procedure will center the information when sending it to the desired output device. The following listing is the DEMO/TXT file. You will need to use BASIC to create this file.

```
10 'exercise 2 main body program : filename DEMO/TXT
20 =INP$,AT%,FIELD%,CENTER$,WIDTH%,DEVICE%,STRING.ARRAY$,LOOP%,LOOP1%
30      CLEAR 2000
40      DIM STRING.ARRAY$(50)
50      GOTO @MAIN
100  ' INPUT PROCEDURE WILL GO HERE
200  'CENTER PROCEDURE WILL GO HERE
300  @MAIN
310      CLS:?"Enter a sentence of less than 50 words"
320       ?STRING$(63,61)
330       ?@128,"Enter @ to stop"
340       FOR LOOP% = 0 TO 50
350              IF LOOP%>0 THEN ?@192,CHR$(30)"Last Entry : "STRING.ARRAY$(LOOP%-1)
360            ?@320,"Current Entry ="
370            AT%=336:FIELD%=10:GOSUB @INPUT
380              IF INP$ = "@" THEN @DISPLAY.VIDEO
390            STRING.ARRAY$(LOOP%)=INP$
400            NEXT LOOP%
410 @DISPLAY.VIDEO
420       WIDTH%=64:DEVICE%=0
430       CLS: FOR LOOP1%= 0 TO LOOP%-1
440            CENTER$= STRING.ARRAY$(LOOP1%)
450            GOSUB @CENTER.DISPLAY
460              IF LOOP1%>0 AND LOOP1%/14 = INT(LOOP1%/14) THEN GOSUB @WAIT.FOR.ENTER
470            NEXT LOOP1%
480       END
490 @WAIT.FOR.ENTER
500       ?@960,"Press <ENTER> to continue";
510 @AGAIN:AT%=985: FIELD%=1: GOSUB @INPUT
520         IF INP$ <> "" THEN @AGAIN
530       CLS
540       RETURN
```

After you have entered this program, you will want to save it to disk. To do so, enter the command:

```
SAVE"DEMO/TXT"
```

Notice that the line numbers in this program are set to accomplish an easy MERGE of the two procedures. First, however, the procedures must be renumbered into their new line number ranges.

To renumber the @INPUT procedure, load it into BASIC using the command -- <LOAD"INPUT/TXT">. After the file is in, renumber it to the chosen 100 to 199 range by using the LBASIC renumber utility CMD"N". The proper syntax is:

```
CMD"N 10,100,1"
```

This takes the old line number 10, changes it to 100 and increments each subsequent line by one. The procedure now should start at line 100 and end at line 143. You will need to Save the result using the ASCII option, since it will later be MERGED into the main body. To do so, enter the command:

```
<SAVE"INPUT/MRG",A>.
```

Next, load in the CENTER/TXT procedure and renumber it into the 200-299 range using the command:

```
CMD"N 10,200,1"
```

The @CENTER.DISPLAY procedure should now start at line 200 and end at line 216.

Save this file using the command <SAVE"CENTER/MRG",A>. All of the preparatory work is now complete. If there had been several procedures the idea is to continue in the same manner to load them in, renumber to a <u>predetermined</u> range, and save them as an ASCII file for latter merging.

To ready the file for final processing, execute the following four command sequences.

```
 LOAD"DEMO/TXT"
MERGE"INPUT/MRG"
MERGE"CENTER/MRG"
SAVE"DEMO/TBA",A
```

You may wish to process the file and observe the resultant code. Experiment by placing the procedures elsewhere in the source code. The general idea of source code creation is to absolutely ignore the line numbers for composition but use them only for final positioning of written modules. This is much easier than BASIC recomposition.

The Final Word

The last exercise is to process the file EXAMPLE/TBA completely on your own. The Source is provided as an LSCRIPT file. If you wish to alter the source to be readable by LBASIC, run the BASIC program EXAMCON/BAS. This will add line numbers in front of the LSCRIPT source lines so that EXAMPLE/TBA can be loaded and edited in LBASIC. When finished, the object code should match the listing on page 69.

The following is a list of the files necessary to run all of the examples. They are grouped into an LSCRIPT group and an LBASIC group depending on the method you plan on using when writing Source. You may delete files that are not relevant to you, preferably, on a backup of the master diskette.

Files Needed by LSCRIPT

TBA/CMD
INPUT/SCR
CENTER/SCR
FACTOR/TBA
EXAMPLE/TBA
EXAMPLE/BAS

Files Needed by LBASIC

TBA/CMD
INPUT/TXT
CENTER/TXT
FACTOR/TBA
EXAMPLE/TBA
EXAMCON/BAS
EXAMPLE/BAS

Note to standard LDOS users

The BASIC Answer is highly conducive to JCL usage, especially for debugging. Since LSCRIPT default extensions are SCR or TXT, a JCL file to copy one of these to a /TBA extension is quite helpful. RENAMEing the file is not as advantageous because when re-entering LSCRIPT the default extensions would no longer apply. In considering a JCL to perform this copying function, all files can be constructed to have the same file names, and different file extensions (such as EXAMPLE/SCR for use with LSCRIPT, EXAMPLE TBA for use with TBA, and EXAMPLE/BAS to represent the final object code). All you need to remember is the filename itself.

TBA inputs can be sent via a JCL file as well, and of course, LBASIC can be entered and the object code run. In debugging your object code, you may find it to be extremely useful to exit LSCRIPT, type DO filename and have the JCL copy the program to the different extension, enter TBA with all inputs pre-answered, and then load and run the object. This process will alleviate performing the repetitive steps required to incorporate minor changes or solve picayune bugs in the source code.

```
      clear 2000
=ARRAY.SIZE%,LABEL.ARRAY$,FIELD%,AT%,INP$,EDIT.FLAG%,GLOBAL.LOOP%,TEMPORARY%
=TEMPORARY$,LABEL.LENGTH%
=NAME.OF.FILE$,CLEAR.LINE$,CLEAR.SCREEN$,NO.OF.LABELS%,INK$,GLOBAL.LOOP1%
=AT.ENTER%
     GOTO @START
'
'INPUT SUBROUTINE
'
@INPUT=INK$,LOOP1%,LOOP2%,FLASH.LOC%,FIELD.LEN%,AT.LEN%
     IF EDIT.FLAG%=0 OR INP$="" THEN @BEGIN.INPUT
     FOR LOOP1%=LEN(INP$)TO 1 STEP -1
          IF MID$(INP$,LOOP1%,1)=" " THEN NEXT LOOP1%:GOTO @BEGIN.INPUT
     IF LOOP1%=FIELD% THEN LOOP1%=LOOP1%-1
     FIELD.LEN%=FIELD%-LOOP1%
     INP$=LEFT$(INP$,LOOP1%)
     AT.LEN%=AT%+LEN(INP$)
     GOTO @DISP.FLD
@BEGIN.INPUT
     FIELD.LEN%=FIELD%:AT.LEN%=AT%
     INK$="":INP$=""
     PRINT@AT%,STRING$(FIELD%,138);
@RE.INPUT
     FOR LOOP1%=1TO10
          INK$=INKEY$
          IF INK$<>""THEN @PROC.INPUT
     NEXT LOOP1%
     FLASH.LOC%=LEN(INK$)
     LOOP1%=0
@DELAY
     IF LOOP1%=0 THEN POKE15360+AT.LEN%+FLASH.LOC%,32:LOOP1%=1:GOTO @DELAY2
     IF LOOP1%=1 THEN POKE15360+AT.LEN%+FLASH.LOC%,138:LOOP1%=0
@DELAY2
     FOR LOOP2%= 1 TO 10
          INK$=INKEY$
          IF INK$<>""THEN @PROC.INPUT
     NEXT LOOP2%
     GOTO @DELAY
@PROC.INPUT
     IF INK$=CHR$(13)THEN @END.INPUT
     IF INK$=CHR$(8) THEN AT.LEN%=AT.LEN%-1: FIELD.LEN%=FIELD.LEN%+1
     IF FIELD.LEN%>FIELD% THEN @BEGIN.INPUT
     IF INK$=CHR$(8) THEN INP$=LEFT$(INP$,LEN(INP$)-1):
     IF ASC(INK$)<32 THEN INK$="": GOTO @DISP.FLD
     INP$=INP$+INK$
     FIELD.LEN%=FIELD.LEN%-1
     AT.LEN%=AT.LEN%+1
     IF FIELD.LEN%=0 THEN PRINT@AT%,INP$;:GOTO @END.INPUT
@DISP.FLD
     PRINT@ AT%, INP$+STRING$(FIELD.LEN%,138);
     GOTO @RE.INPUT
@END.INPUT
     EDIT.FLAG%=0
     IF FIELD.LEN%<>0 THEN PRINT@ AT%,INP$;STRING$(FIELD.LEN%,32);
     RETURN
'
'END  OF INPUT SUBROUTINE
'

'===
'PRESS ENTER TO CONTINUE SUBROUTINE
'===

@PRESS.ENTER
     PRINT@ AT.ENTER%,"PRESS <ENTER> TO CONTINUE"
@INPUT.ENTER
     AT%= AT.ENTER%, FIELD%=1:GOSUB @INPUT
     IF INP$<>"" THEN @INPUT.ENTER
     RETURN

'END  OF PRESS ENTER SUBROUTINE


'===
'DISPLAY LABEL SUBROUTINE
'===


@DISPL.LABEL

     PRINT@320,CLEAR.SCREEN$
     FOR GLOBAL.LOOP%=1 TO ARRAY.SIZE%
          PRINT@320+(64* GLOBAL.LOOP%),"LINE #";GLOBAL.LOOP%;" "; LABEL.ARRAY$
          (GLOBAL.LOOP%)
     NEXT GLOBAL.LOOP%
     RETURN


'===
'ABORT PRINTING ROUTINE
'===


@ABORT.PRINTING


     PRINT@960,CLEAR.SCREEN$;"DO YOU WISH TO ABORT PRINTING (Y/N)";
@INPUT.ABORT
     AT%=998:FIELD%=2:GOSUB @INPUT
     IF INP$="Y" OR INP$="N" THEN PRINT@960,CLEAR.SCREEN$; ELSE @INPUT.ABORT
     RETURN


@START
'===
'IF THE DIRECTIVE "EIGHT" WAS PASSED, SET ARRAY.SIZE% TO 8
'===

*IF EIGHT
    ARRAY.SIZE%=8
    GOTO @DIMENSION
*END
'===
'
'IF THE DIRECTIVE "EIGHT" WAS NOT PASSED, SET ARRAY.SIZE% TO 6
     ARRAY.SIZE%=6

@DIMENSION
     dim LABEL.ARRAY$(ARRAY.SIZE%)
     CLEAR.LINE$=CHR$(30):CLEAR.SCREEN$=CHR$(31)

```

```
00121. '
00122. '===
00123. 'IF THE DIRECTIVE "CHARS" WAS PASSED, TAKE THE INPUT FOR THE
00124. 'NUMBER OF CHARACTERS PER LINE
00125. '===
00126. '
OR127. *IF CHARS
00128. @CHARS.INPUT
00129.      CLS:PRINT@524,"ENTER NUMBER OF CHARACTERS PER LABEL"
00130.      AT%=561:FIELD%=2:GOSUB @INPUT
00131.       IF INP$="@" THEN GOTO @END.PROGRAM
00132.       IF INP$="" THEN @CHARS.INPUT
00133.      FOR GLOBAL.LOOP%=1TOLEN(INP$)
00134.           TEMPORARY$=MID$(INP$,GLOBAL.LOOP%,1)
00135.            IF TEMPORARY$<"0" OR TEMPORARY$>"9" THEN @CHARS.INPUT
00136.       NEXT GLOBAL.LOOP%
00137.       LABEL.LENGTH%=VAL(INP$)
00138.       GOTO @INPUT.FILE
00139. *END
00140. '
00141. '
00142.      LABEL.LENGTH%=35
00143. '
00144. '
00145. @INPUT.FILE
00146.
00147.      NAME.OF.FILE$=""
00148.      FOR GLOBAL.LOOP%=1 TO ARRAY.SIZE%
00149.           LABEL.ARRAY$(GLOBAL.LOOP%)=" "
00150.      NEXT GLOBAL.LOOP%
00151.      CLS:PRINTTAB(16)"LABEL PRINTING PROGRAM"
00152.      PRINT@330,"DO YOU WISH TO USE AN EXISTING FILE (Y,N,@)"
00153.      AT%=376:FIELD%=2:GOSUB @INPUT
OV154.       IF INP$="@" THEN @END.PROGRAM
00155.       IF INP$="N" THEN @BUILD.LABEL ELSE IF INP$<>"Y" THEN @INPUT.FILE
00156.
00157. @ENTER.FILE
00158.      PRINT@330,CLEAR.LINE$;"ENTER THE NAME OF THE FILE"
00159.      AT%=358:FIELD%=15:GOSUB @INPUT
00160.       IF INP$="@" THEN @INPUT.FILE
00161.      ONERRORGOTO @NO.SUCH.FILE
00162.      NAME.OF.FILE$=INP$
00163.      OPEN"I",1,NAME.OF.FILE$
00164.      ONERRORGOTO0
00165.      TEMPORARY%=0
00166.
00167. @INPUT.LINE
00168.
00169.       IF EOF(L) THEN CLOSE:GOSUB @DISPL.LABEL:ONERRORGOTOO:GOTO @EDIT.LABEL
00170.      TEMPORARY%=TEMPORARY%+1:IF TEMPORARY%>ARRAY.SIZE% THEN @WRONG.PROG
00171.      LINEINPUT#1,LABEL.ARRAY$(TEMPORARY%)
00172.       IF LEN(LABEL.ARRAY$(TEMPORARY%))>LABEL.LENGTH% THEN @WRONG.PROG
00173.      GOTO @INPUT.LINE
00174.
00175.
00176. @BUILD.LABEL
00177.
00178.      PRINT@64,CLEAR.SCREEN$
00179.      FOR GLOBAL.LOOP%=1 TO ARRAY.SIZE%
00180.           PRINT@320+(GLOBAL.LOOP%*64),"LINE #";GLOBAL.LOOP%
00181.           AT%=330+(GLOBAL.LOOP%*64):FIELD%=LABEL.LENGTH%:GOSUB @INPUT
```

```
           IF INP$="@" THEN IF GLOBAL.LOOP%=1 THEN @INPUT.FILE ELSE @BUILD.LABEL
           IF INP$="" THEN INP$=" "
          LABEL.ARRAY$(GLOBAL.LOOP%)=INP$
     NEXT GLOBAL.LOOP%


@EDIT.LABEL

     PRINT@960,"ENTER COMMAND (Y IF CORRECT, LINE # TO CORRECT, @ TO ABORT";
     AT%=1020:FIELD%=2:GOSUB @INPUT
      IF INP$="@" THEN @INPUT.FILE
      IF INP$="Y" THEN @PRINT.LABEL
     TEMPORARY%=VAL(INP$):IF TEMPORARY%<1 OR TEMPORARY%>ARRAY.SIZE% THEN @EDIT.LABEL
     INP$=LABEL.ARRAY$(TEMPORARY%)
     AT%=310+(TEMPORARY%*64):FIELD%=LABEL.LENGTH%:EDIT.FLAG%=1:GOSUB @INPUT
     LABEL.ARRAY$(TEMPORARY%)=INP$
     GOTO @EDIT.LABEL


@PRINT.LABEL

      PRINT@960,CLEAR.SCREEN$;
      PRINT"NUMBER OF LABELS TO PRINT? (<ENTER>=1, <@> TO ABORT)";
      AT%=1015:FIELD%=3:GOSUB @INPUT
       IF INP$="@" THEN @EDIT.LABEL
       IF INP$=""THEN NO.OF.LABELS%=1 ELSE NO.OF.LABELS%=VAL(INP$):IF NO.OF.LABELS%<l         THEN @PRINT.LABEL
      FOR GLOBAL.LOOP%=1 TO NO.OF.LABELS%
           INK$=INKEY$:IF INK$="@" THEN GOSUB @ABORT.PRINTING:IF INP$="Y" THEN @SAVE.FILE
           FOR GLOBAL.LOOP1%=1 TO ARRAY.SIZE%
                LPRINT LABEL.ARRAY$(GLOBAL.LOOP1%)
           NEXT GLOBAL.LOOP1%
      NEXT GLOBAL.LOOP%
      PRINT@960,CLEAR.SCREEN$;"DO YOU WISH TO PRINT MORE? (Y/N)";

@PRINT.MORE

      AT%=994:FIELD%=2:GOSUB @INPUT
      IF INP$="Y" THEN @PRINT.LABEL ELSE IF INP$<>"N" THEN @PRINT.MORE

@SAVE.FILE

      PRINT@960,CLEAR.SCREEN$;"DO YOU WISH TO SAVE THIS FILE (Y/N)";
      AT%=997:FIELD%=2:GOSUB @INPUT
       IF INP$="N" THEN @INPUT.FILE ELSE IF INP$<>"Y" THEN @SAVE.FILE

@FILENAME

      PRINT@960,CLEAR.SCREEN$;"ENTER FILENAME (<ENTER>=SAME NAME)";
      AT%=996:FIELD%=15:GOSUB @INPUT
       IF INP$<>"" THEN NAME.OF.FILE$=INP$:GOTO @WRITE.FILE
       IF NAME.OF.FILE$="" THEN @FILENAME

@WRITE.FILE

      ONERRORGOTO @CANNOT.WRITE
      OPEN"O",1,NAME.OF.FILE$
      FOR GLOBAL.LOOP%=1 TO ARRAY.SIZE%
```

```
          PRINT#L, LABEL.ARRAY$(GLOBAL.LOOP%)
     NEXT GLOBAL.LOOP%
     CLOSE
     ONERRORGOTO0
     GOTO @INPUT.FILE


@NO.SUCH.FILE

     PRINT@330,CLEAR.LINE$;"CANNOT USE FILE --->";NAME.OF.FILE$
     PRINT@458,"THE ERROR THAT OCCURRED IS ERROR #";ERR
     AT.ENTER%=586
     GOSUB @PRESS.ENTER
     PRINT@330,CLEAR.SCREEN$
     NAME.OF.FILE$="":RESUME @ENTER.FILE


@WRONG.PROG

     PRINT@330,CLEAR.SCREEN$;"CANNOT USE THIS VERSION OF THE PROGRAM":PRINT@394,"TO PROCESS THE FILE ---> ";NAME.OF.FILE$
     AT.ENTER%=586:GOSUB @PRESS.ENTER
     NAME.OF.FILE$="":CLOSE:GOTO @INPUT.FILE


@CANNOT.WRITE

     PRINT@960,CLEAR.SCREEN$;"CANNOT SAVE FILE -- ERROR=";ERR;". <R>ETRY OR @ TO ABORT";
@INPUT.WRITE
     AT%=1017:FIELD%=2:GOSUB @INPUT
     IF INP$="R" THEN NAME.OF.FILE$="":RESUME @SAVE.FILE ELSE IF INP$<>"@" THEN @INPUT.WRITE
     RESUME @END.PROGRAM
@END.PROGRAM
      ONERRORGOTO0:CLS:PRINT@405,"LABEL PROGRAM ABORTED":CLOSE:END

```

```
1 CLEAR2000
6 GOT0199
11 IFED%=0ORIO$=""THEN20
12 FORLO%=LEN(IO$)TO1STEP-1
13 IFMID$(IO$,LO%,1)=" "THENNEXTLO%:GOTO20
14 IFLO%=FJ%THENLO%=LO%-1
15 FI%=FJ%-LO%
16 IO$=LEFT$(IO$,LO%)
17 AT%=AU%+LEN(IO$)
18 GOT050
20 FI%=FJ%:AT%=AU%
21 IN$="":IO$=""
22 PRINT@AU%,STRING$(FJ%,138);
24 FORLO%=1TO10
25 IN$=INKEY$
26 IFIN$<>""THEN40
27 NEXTLO%
28 FL%=LEN(IN$)
29 LO%=0
31 IFLO%=0THENPOKE15360+AT%+FL%,32:LO%=1:GOTO34
32 IFLO%=1THENPOKE15360+AT%+FL%,138:LO%=0
34 FORLP%=1TO10
35 IN$=INKEY$
36 IFIN$<>""THEN40
37 NEXTLP%
38 GOT031
40 IFIN$=CHR$(13)THEN53
41 IFIN$=CHR$(8)THENAT%=AT%-1:FI%=FI%+1
42 IFFI%>FJ%THEN20
43 IFIN$=CHR$(8)THENIO$=LEFT$(10$,LEN(IO$)-1):
44 IFASC(IN$)<32THENIN$="":GOTO50
45 IO$=IO$+IN$
46 FI%=FI%-1
47 AT%=AT%+1
48 IFFI%=0THENPRINT@AU%,IO$;:GOTO53
50 PRINT@AU%,IO$+STRING$(FI%,138);
51 GOT024
53 ED%=0
54 IFFI%<>0THENPRINT@AU%,IO$;STRING$(FI%,32);
55 RETURN
65 PRINT@AV%,"PRESS <ENTER> TO CONTINUE"
67 AU%=AV%,FJ%=1:GOSUB11
68 IFIO$<>""THEN67
69 RETURN
81 PRINT@320,CM$
82 FORGL%=1TOAR%
83 PRINT@320+(64*GL%),"LINE #";GL%;" ";LA$(GL%)
84 NEXTGL%
85 RETURN
96 PRINT@960,CM$;"DO YOU WISH TO ABORT PRINTING (Y/N)";
98 AU%=998:FJ%=2:GOSUB11
99 IFIO$="Y"ORIO$="N"THENPRINT@960,CM$;ELSE98
100 RETURN
109 AR%=8
110 GOT0118
115 AR%=6
118 DIMLA$(AR%)
119 CL$=CHR$(30):CM$=CHR$(31)
129 CLS:PRINT@524,"ENTER NUMBER OF CHARACTERS PER LABEL"
130 AU%=561:FJ%=2:GOSUB11
```

Sample output of EXAMPLE/BAS

```
131 IFIO$="@"THENGOT0271
132 IFIO$=""THEN129
133 FORGL%=1TOLEN(IO$)
134 TE$=MID$( IO$,GL%,l)
135 IFTE$<"0"ORTE$>"9"THEN129
136 NEXTGL%
137 LA%=VAL(IO$)
138 GOT0147
142 LA%=35
147 NA$=""
148 FORGL%=1TOAR%
149 LA$(GL%)=" "
150 NEXTGL%
151 CLS:PRINTTAB(16)"LABEL PRINTING PROGRAM"
152 PRINT@330,"DO YOU WISH TO USE AN EXISTING FILE (Y,N,@)"
153 AU%=376:FJ%=2:GOSUB11
154 IFIO$="@"THEN271
155 IFIO$="N"THEN178ELSEIFIO$<>"Y"THEN147
158 PRINT@330,CL$;"ENTER THE NAME OF THE FILE"
159 AU%=358:FJ%=15:GOSUB11
160 IFIO$="@"THEN147
161 ONERRORGOT0248
162 NA$=IO$
163 OPEN"I",1,NA$
164 ONERRORGOTO0
165 TE%=0
169 IFEOF(l)THENCLOSE:GOSUB81:ONERRORGOTO0:GOTO190
170 TE%=TE%+1:IFTE%>AR%THEN512
171 LINEINPUT#1,LA$(TE%)
172 IFLEN(LA$(TE%))>LA%THEN512
173 GOT0169
178 PRINT@64,CM$
179 FORGL%=1TOAR%
180 PRINT@320+(GL%*64),"LINE #";GL%
181 AU%=330+(GL%*64):FJ%=LA%:GOSUB11
182 IFIO$="@"THENIFGL%=1THEN147ELSE178
183 IFIO$=""THENIO$=" "
184 LA$(GL%)= IO$
185 NEXTGL%
190 PRINT@960,"ENTER COMMAND (Y IF CORRECT, LINE # TO CORRECT, @ TO ABORT";
191 AU%=1020:FJ%=2:GOSUB11
192 IFIO$="@"THEN147
193 IFIO$="Y"THEN203
194 TE%=VAL(IO$):IFTE%<10RTE%>AR%THEN190
195 IO$=LA$(TE%)
196 AU%=310+(TE%*64):FJ%=LA%:ED%=1:GOSUB11
197 LA$(TE%)=IO$
198 GOT0190
203 PRINT@960,CM$;
204 PRINT"NUMBER OF LABELS TO PRINT? (<ENTER>=1, <@> TO ABORT)";
205 AU%=1015:FJ%=3:GOSUB11
206 IFIO$="@"THEN190
207 IFIO$=""THENNO%=1ELSENO%=VAL(IO$):IFNO%<1THEN203
208 FORGL%=1TONO%
209 IP$=INKEY$:IFIP$="@"THENGOSUB96:IFIO$="Y"THEN223
210 FORGM%=1TOAR%
211 LPRINTLA$(GM%)
212 NEXTGM%
213 NEXTGL%
214 PRINT@960,CM$;"DO YOU WISH TO PRINT MORE? (Y/N)";
218 AU%=994:FJ%=2:GOSUB11
```

```
219 IFIO$="Y"THEN203ELSEIFIO$<>"N"THEN218
223 PRINT@960,CM$;"DO YOU WISH TO SAVE THIS FILE (Y/N)";
224 AU%=997:FJ%=2:GOSUB11
225 IFIO$="N"THEN147ELSEIFIO$<>"Y"THEN223
229 PRINT@960,CM$;"ENTER FILENAME (<ENTER>=SAME NAME)";
230 AU%=996:FJ%=15:GOSUB11
231 IFIO$<>""THENNA$=IO$:GOTO236
232 IFNA$=""THEN229
236 ONERRORGOTO265
237 OPEN"O",1,NA$
238 FORGL%=1TOAR%
239 PRINT#1,LA$(GL%)
240 NEXTGL%
241 CLOSE
242 ONERRORGOTO0
243 GOTO147
248 PRINT@330,CL$;"CANNOT USE FILE --->";NA$
249 PRINT@458,"THE ERROR THAT OCCURRED IS ERROR #";ERR
250 AV%=586
251 GOSUB65
252 PRINT@330,CM$
253 NA$="":RESUME158
258 PRINT@330,CM$;"CANNOT USE THIS VERSION OF THE PROGRAM":PRINT@394,"TO PROCESS THE
FILE ---> ";NA$
259 AV%=586:GOSUB65
260 NA$="":CLOSE:GOTO147
265 PRINT@960,CM$;"CANNOT SAVE FILE -- ERROR=";ERR;". <R>ETRY OR @ TO ABORT";
267 AU%=1017:FJ%=2:GOSUB11
268 IFIO$="R"THENNA$="":RESUME223ELSEIFIO$<>"@"THEN267
269 RESUME271
271 ONERRORGOTO0:CLS:PRINT@405,"LABEL PROGRAM ABORTED":CLOSE:END
```

[Original Warranty Card]

TBA(TM) - The BASIC Answer

(C)1982 by Logical Systems, Inc.

Serial # TBA-20029

TRS-80 Model I ________ III _________

LDOS Serial # __________________________

Name ____

Address ____

City ___________________________ State ___________ Zip _______________________

Country ________________________ Phone ____

Purchased from __________________________ Date purchased ______________________

This product is sold on an "as is" basic, with no expressed or implied warranties. The purchaser agrees that he will not duplicate or disseminate this product in any way whatsoever, except to make copies for his personal use.

Signature ____________________________________ Date ___________________________