MODEL 4 TRSDOS 6.Ø2.ØØ

UTILITIES PACKAGE

1Ø 9 8 7 6 5 4 3 2 1

TRSDOS 6.2 Utilities
Cat. No. 26-Ø315
Addendum

The COMP6 utility recognizes only the abbreviated form of the PRINT parameter, P.

Using COMP6 to Compare 2 Diskettes in a 2-Drive System

At the TRSDOS Ready prompt, type:

```
SYSTEM (SYSRES=1) <ENTER>
```

When the TRSDOS Ready prompt again appears on your screen, type the following commands::

```
SYSTEM (SYSRES=4) <ENTER>
RUN (X) COMP6 :Ø :1 <ENTER>
```

Now the screen shows:

```
Insert SOURCE disk <ENTER>
```

If the system disk contains the COMP6 utility, press <ENTER>. If the COMP6 utility is on a different diskette, insert that diskette and press <ENTER>.

Now the screen shows:

```
Insert SYSTEM diskette <ENTER>
```

Insert one of the diskettes to compare in Drive Ø. Insert the other diskette in Drive 1. Now press <ENTER>.

When the comparison is complete, the screen shows:

```
Disk compare completed
```

Remove the diskette from Drive Ø and insert the system diskette.

Using COMP6 to Compare Files on Data Diskettes

At the TRSDOS Ready prompt, type the following commands::

```
SYSTEM (SYSRES=1) <ENTER>
SYSTEM (SYSRES=2) <ENTER>
SYSTEM (SYSRES=3) <ENTER>
SYSTEM (SYSRES=4) <ENTER>
RUN (X) COMP6 <ENTER>
```

The screen shows:

```
Insert SOURCE disk <ENTER>
```

If the system disk contains the COMP6 utility, press <ENTER>. If COMP6 is on another diskette, insert that diskette and press <ENTER>.

The screen shows:

```
Insert SYSTEM diskette <ENTER>
```

Insert one diskette in Drive Ø. Insert the other diskette in Drive 1. Press <ENTER>.

The screen shows:

```
Filespec 1?
Filespec 2?
```

Type the names of the files you are comparing and the drive number that contains the diskette on which the file resides. When the comparison is complete, the screen shows:

Disk compare completed

Remove the diskette from Drive Ø and insert the system diskette.

Part No. 875-9724

C O N T E N T S

I N T R O D U C T I O N

The TRSDOS 6.2 UTILITIES package is a powerful group of programs that you use to write other programs. You use some of these utilities with BASIC programs only.

This package contains the following 6 utility programs:

QFB6, a program that quickly formats and backs up floppy diskettes.
COMP6, a program that lets you compare 2 floppy diskettes or 2 files. You can also use it to compare parts of diskettes or parts of files.

BSORT, a utility you call from a BASIC program that lets you sort primary, secondary, tag, and index arrays. Arrays can be multidimensional, and you can arrange them in ascending or descending order.

MOD324, a system for transferring Model III BASIC programs to a Model 4 BASIC form, with a minimum of editing or reprogramming.

UNKILL, a utility for recovering a file that you removed or purged, but did not yet allocate the space to another file.

TBA, a system that lets you write BASIC programs in a structured self-documenting manner, and lets you maintain programs easily.

Note: Use these utilities on Model 4 TRSDOS, Version 6.Ø2.ØØ. Do not use them on earlier versions of TRSDOS.

BLANK PAGE

QFB6--QUICK FORMAT AND BACKUP

The QFB6 (Quick Format and Backup) utility lets you create a mirror image backup of the source disk without first formatting the destination disk. To use QFB6, you must have 2 floppy drives and a source diskette formatted with the TRSDOS 6.2 FORMAT utility. The syntax is:

QFB6 [:]s [:]d [(parameters,...)]

:s indicates the source drive. The colon is optional.
:d indicates the destination drive. The colon is optional.

If you omit source and/or destination drive, QFB6 prompts you for them.

Parameters:

ALL= specifies whether QFB6 reads and copies all cylinders of the source disk to the destination disk, or only allocated cylinders. You can specify the ON/OFF switch; the default is OFF.

V1= specifies whether QFB6 verifies the destination disk on the first pass (after it writes each cylinder). You can specify the ON/OFF switch; the default is on.

V2= specifies whether QFB6 verifies the destination disk on the second pass (after it writes the complete disk). You can specify the ON/OFF switch; the default is OFF.

QUERY= prompts for unspecified parameters. You can specify the ON/OFF switch. The default is OFF.

Abbr: ON=Y, OFF=N, QUERY=Q, ALL=A

QFB6 formats and backs up in a single pass. If you omit drives, QFB6 asks you for them. In the command line, :s is the source drive; :d is the destination drive. If you omit parameters, QFB6 uses the defaults.

To format and backup, at the TRSDOS prompt, type:

QFB6 1 2 [ENTER]

Drive 1 is the source drive, and Drive 2 is the destination drive. The screen shows: Load diskettes and press [ENTER]. After you insert the diskettes and press [ENTER], the backup begins. The following takes place:

1. QFB6 logs in the source diskette to determine the type of format.
2. QFB6 formats Cylinder Ø of the destination diskette.
3. If Cylinder Ø of the source disk contains data, QFB6 reads it into memory and writes it to the destination diskette.
4. QFB6 verifies Cylinder Ø of the destination diskette (the V1 parameter default).
5. QFB6 repeats Steps 2-4 for all remaining cylinders.
6. The screen shows the following message after QFB6 verifies the last cylinder:

```
Duplication complete        1 disk  created

   Replace destination disk and press <ENTER> to repeat
          ..<R> to restart with new parameters
            ...or....<BREAK> to exit program.
```

7. To terminate the program, press **[BREAK]**. To make another backup, press **[ENTER]**. If you press <BREAK>, the screen shows:

    Load SYSTEM diskette and press <ENTER>

   Place a system diskette in Drive Ø, and press **[ENTER]** to return to TRSDOS. The prompt appears even if you run QFB6 from a hard drive, in which case, press **[ENTER]**.

   To use QFB6 again with different parameters, type R **[ENTER]**. QFB6 prompts you for the drives and the parameters.

The screen shows these same prompts (displayed below) if you type the command **QFB6 (Q=Y) [ENTER]**.

```
Source drive?
Destination drive?
Duplicate unallocated tracts? (Y/N)
Verify on same pass? (Y/N)
Verify on second pass? (Y/N)
```

Respond to these questions by typing either Y (Yes) or N (No) and [ENTER].

The first prompt relates to the ALL parameter. If you answer Yes, QFB6 reads all cylinders from the source diskette and writes them to the destination diskette, regardless of whether or not the cylinder contains information. If you answer No, QFB6 reads and writes only cylinders containing information.

The next prompt relates to the V1 parameter. If you answer Yes, QFB6 verifies each cylinder on the destination diskette immediately after each write. If you answer No, QFB6 does not immediately verify.

The final prompt relates to the V2 parameter. If you answer Yes, QFB6 verifies all cylinders on the destination diskette after it completes all writes to the diskette. If you answer NO, there is no second pass verification.

If an error occurs, the screen shows an appropriate error message and prompts you for an action.

During any QFB6 operation, you can press [BREAK] it to terminate the process.

Note: QFB6 assumes that you want a mirror image backup, and does not check for data on the destination diskette. It destroys any existing information on a destination diskette. Also, QFB6 does not clear the Mod Flags of files on the source disk.

COMP6--COMPARE PROGRAM

Using a character for character match, this utility compares 2 files or 2 diskettes to determine if the information is identical. Use COMP6 after a backup or a copy to determine the validity of the data. Thesyntax is:

COMP6 filespec1 [TO] filespec2 [(parameter,...)]
COMP6 :drive1 [TO] :drive2 [(parameter,...)]

Parameters:

REC= specifies record number at which to start comparing 2 filespecs (default is Ø).

NUM= indicates the number of records of a filespec, or the sectors of a disk, to compare.

ALL displays each nonmatching byte.

PRINT sends display to *PR and *DO.

CYL= specifies cylinder at which to start comparing 2 drives (default is Ø).

SEC= specifies sector at which to start comparing 2 diskettes or 2 disks (default is Ø).

Abbr: REC=R, NUM=N, ALL=A, PRINT=P, CYL=C, SEC=S

This command:

COMP6 MAY/DAT:3 :4 [ENTER]

generates this output:

```
MAY/DAT:3      contains  17 sectors, EOF offset = 7Ø
MAY/DAT:4      contains  17 sectors, EOF offset = 7Ø
```

With COMP6, a drivespec alone can serve as the second filespec. COMP6 allows this 1 exception to typing a complete filespec. Since the files are identical, the screen displays the number of sectors compared followed by the end-of-file offset.

If the files are different, the following occurs:

COMP6 FISCAL82/DAT:3 FISCAL82/DAT:4 (R=4) [ENTER]

```
   Posn= X'ØØØ5,ØØ  FISCAL82/DAT:3  = X'2Ø, FISCAL82/DAT:4   = X'ØØ
29 bytes did not match.

   Posn= X'ØØØ5,BØ  FISCAL82/DAT:3  = X'54, FISCAL82/DAT:4   = X'ØØ
32 bytes did not match.

   FISCAL82/DAT:3   contains   18 sectors, EOF offset =  1ØØ
   FISCAL82/DAT:4   contains   18 sectors, EOF offset =  1ØØ
```

Line 1 shows the record number of a discrepant sector, the number of the first discrepant byte (following Posn=), and the contents of that byte in each filespec. Line 2 displays the total number of continuous bytes that do not match. If you specify the ALL parameter, the screen displays each unmatching byte. By specifying the parameter R=4, the comparison begins at Record 4.

To compare 1 disk with another, use drive numbers instead of filespecs. Specify the starting cylinder and sector number either in hexadecimal (X'ØØ') format or as a decimal integer. Use the NUM= parameter to specify the number of continuous sectors to compare.

Unlike file-to-file comparisons, disk-to-disk comparisons display the currently accessed cylinder, sector, and byte. The source drive (the first drivespec) reads as much information as possible into memory. COMP6 then compares this information to the destination drive. If COMP6 detects discrepant bytes, the screen displays:

```
   Cyl X'ØD, Sec X'ØØ, Byte X'ØØ, Drive 2 = X'6D, Drive 3 = X'31
3Ø78 bytes did not match.
```

If you specify the ALL parameter, the screen displays the contents of each different byte. To send the output to the printer and the screen, specify the PRINT parameter.

BSORT--BASIC SORT UTILITY

BSORT is a high speed utility that sorts BASIC arrays. It sorts any type of array (integer, single- or double-precision, or string), including 1- and 2-dimensional arrays. To utilize BSORT, use the following syntax as a line number in your BASIC program.

Note: Integer refers to integer variables or constants.

SYSTEM"RUN BSORT [NUM],*IND%,[+][-]PSA[x],[parameter,...]"

SYSTEM"RUN BSORT $STRVAR$"

NUM — number of elements to sort, must be an integer.

*IND%(x) — single dimension integer array. Use to generate an index array containing element numbers of the sorted array. Do not use to reorder elements in the array you are sorting.

PSA — primary sort array name. An optional plus (+) or minus (-) precedes the array name to indicate ascending (+) or descending (-) order. If you omit a directional sign, BSORT assumes ascending order. A type declaration tag (!,#,$,%) follows the array name.

x — integer; indicates the first element number you want to sort (subscript).

Optional parameters:

SSA(x) secondary sort array. A plus (+) or minus (-) precedes the array name. A type declaration tag follows. Use a sort key that includes corresponding information from the primary and secondary arrays. Any reordering of the primary array causes a corresponding reordering of the secondary array. You can use more than 1 array, but if the secondary array is 2-dimensional, use a subscript.

TA tag array. Any reordering of the primary array causesa corresponding reordering in a tag array. A plus or minus cannot precede a tag array. Specify tag arrays after secondary array definitions. To specify more than 1 tag array, separate each with a comma.

(s,n) mid-string information that indicates the sort key begins at position s in the string, for n characters, where s and n are integer numbers. Valid only with string arrays and immediately following the array information. Do not use it with tag arrays.

$STRVAR$ non-array string variable that contains parameters for sorting. Use if there are more than 79 characters within the quotation marks.

BSORT performs many different sorting tasks within a BASIC program. Use established variables and arrays (BSORT cannot allocate memory for them). You must use a dimensioned array in a sort command. The following examples illustrate the types of sorts BSORT performs.

Sorting a Single-Dimension Array

Although you can sort any type of array (integer, single- or double-precision, or string), sorting the single dimension array is easiest. To sort a single dimension array, pass 2 parameters--the number of elements to sort, and the starting position in the primary sort array--to BSORT. For example, assume the following string array exists in memory:

A$(1)	A$(2)	A$(3)	A$(4)	A$(5)	A$(6)
SMITH	JONES	BROWN	WILLIAMS	JOHNSON	GREEN

To sort this array, enter the following command as a line number in your BASIC program:

SYSTEM"RUN BSORT 6,+A$(1)" [ENTER]

A$(1)	A$(2)	A$(3)	A$(4)	A$(5)	A$(6)
BROWN	GREEN	JOHNSON	JONES	SMITH	WILLIAMS

The command specifies sorting 6 elements in array A$ (the primary sort array), and starting to sort at Element 1.

This type of sort reorders elements in ascending order, so that the value of A$(1) is less than A$(2) is less than A$(3), and so on. The plus sign preceding the primary sort array tells BSORT to reorder the array in ascending order.

To sort a primary array in descending order, precede the primary sort array with a minus sign. Execute the sort command by typing:

SYSTEM"RUN BSORT 6,-A$(1)" [ENTER]

Now, the value of A$(1) is Williams, and the value of A$(6) is Brown.

In the previous examples, integer constants represent both the number of elements to sort (6) and the starting string array position (1). If the variable in the sort command has a type declaration tag, you can use DEF statements (for example, DEF INT).

You can sort any part of an array for any number of elements. In the previous examples, if A$ has 7 elements [A$(Ø) through A$(6)], you can sort Elements 2-5 in ascending order with these commands:

```
NM%=4:PO%=2
SYSTEM"RUN BSORT NM%,A$(PO%)"
```

If you sort beyond the dimensions of the array, BSORT returns an error. In the previous example, if PO% is 2, NM% must be less than 6.

Using Secondary Sort Arrays

You can use more than 1 array to determine the results of a sort operation. Specify secondary sort arrays after the primary sort array. BSORT reorders them in conjunction with the primary sort array, and they help determine direction.

EXAMPLE (Array A$):

A$(1)	A$(2)	A$(3)	A$(4)	A$(5)	A$(6)
SMITH	JONES	JONES	WILLIAMS	JOHNSON	JONES

F$(1)	F$(2)	F$(3)	F$(4)	F$(5)	F$(6)
SAMMY	BILLY	BETTY	RICHARD	CHARLES	BOBBY

The Array A$ represents a list of last names; the Array F$ contains the corresponding first names. If the last names are the same, the first name determines the ascending order. To create a list of these names in ascending order, use the command below:

SYSTEM"RUN BSORT 6,A$(1),+F$ [ENTER]

A$(1)	A$(2)	A$(3)	A$(4)	A$(5)	A$(6)
JOHNSON	JONES	JONES	JONES	SMITH	WILLIAMS

F$(1)	F$(2)	F$(3)	F$(4)	F$(5)	F$(6)
CHARLES	BETTY	BILLY	BOBBY	SAMMY	RICHARD

The Array F$ is a secondary sort array. It determines the sorted order if a direct match occurs in the primary array. When you specify a secondary sort array, you assume a direct correlation between elements in the primary array. If you reorder the primary array, you reorder the secondary array as well. In the above example, the last names carry the first names with them to their new position in the array. If any last names match exactly, the program sorts the first names.

Separate a single-dimension secondary array from the primary array with a comma. You do not need a subscript number. The element number in the primary array determines any reordering. In other words, Element 1 in the primary array corresponds to Element 1 in the secondary array. You can use BSORT only to sort the number of elements that is common to both the primary and secondary array.

For example, if a primary array has 5Ø elements (Ø-49) and a secondary array has 1Ø elements (Ø-9), you can sort both arrays up to and including Element 9. Attempting to sort beyond the highest allowable element number common to both the primary or secondary array generates error.

Unlike primary arrays, a direction sign [(+) or (-)] is mandatory when you specify a secondary array. The direction of the sort in a secondary array does not have to match that in the primary array. Using the Arrays A$ and F$, the following sort command:

SYSTEM"RUN BSORT 6,+A$(1),-F$" [ENTER]
produces these results:

A$(1)	A$(2)	A$(3)	A$(4)	A$(5)	A$(6)
JOHNSON	JONES	JONES	JONES	SMITH	WILLIAMS

F$(1)	F$(2)	F$(3)	F$(4)	F$(5)	F$(6)
CHARLES	BOBBY	BILLY	BETTY	SAMMY	RICHARD

Note: The direction of the secondary sort array (in descending order) does not affect the reordering of the primary array (ascending order). However, any exact matches in the primary array sorts the secondary array (first name array) in descending order.

Using Multiple Secondary Arrays

When using more than 1 name, separate secondary arrays with commas.

The order of the array names in the command line determines how the arrays reorder. In particular, the secondary array specified first

takes precedence. For example, examine the 3 arrays below:

A$(1)	A$(2)	A$(3)	A$(4)	A$(5)	A$(6)	A$(7)
SMITH	BROWN	JONES	JONES	JONES	JONES	JONES

F$(1)	F$(2)	F$(3)	F$(4)	F$(5)	F$(6)	F$(7)
SAMMY	ROBBY	JOHN	JAKE	JOHN	HERB	HERM

I%(1)	I%(2)	I%(3)	I%(4)	I%(5)	I%(6)	I%(7)
1ØØ1	1ØØ2	1ØØ3	1ØØ4	1ØØ5	1ØØ6	1ØØ7

Array A$ contains last names, Array F$ contains first names, and Array I% contains ID numbers. Consider the results of the command:

SYSTEM"RUN BSORT 7,A$(1),+F$,-I%" [ENTER]

A$(1)	A$(2)	A$(3)	A$(4)	A$(5)	A$(6)	A$(7)
BROWN	JONES	JONES	JONES	JONES	JONES	SMITH

F$(1)	F$(2)	F$(3)	F$(4)	F$(5)	F$(6)	F$(7)
ROBBY	HERB	HERM	JAKE	JOHN	JOHN	SAMMY

I%(1)	I%(2)	I%(3)	I%(4)	I%(5)	I%(6)	I%(7)
1ØØ2	1ØØ6	1ØØ7	1ØØ4	1ØØ5	1ØØ3	1ØØ1

First, BSORT sorts the last names in ascending order. If the last names match exactly (as with JONES), BSORT determines the correct ascending order by referring to the first names in the secondary array F$. If 2 people have identical first and last names, BSORT uses the ID number in the secondary array I% to sort the identical names in descending order.

If you transpose the arrays on the command line, you get different results. For example:

SYSTEM"RUN BSORT 7,A$(1),-I%,+F$" [ENTER]

A$(1)	A$(2)	A$(3)	A$(4)	A$(5)	A$(6)	A$(7)
BROWN	JONES	JONES	JONES	JONES	JONES	SMITH

F$(1)	F$(2)	F$(3)	F$(4)	F$(5)	F$(6)	F$(7)
ROBBY	HERM	HERB	JOHN	JAKE	JOHN	SAMMY

I%(1)	I%(2)	I%(3)	I%(4)	I%(5)	I%(6)	I%(7)
1ØØ2	1ØØ7	1ØØ6	1ØØ5	1ØØ4	1ØØ3	1ØØ1

As in the previous example, the last names sort in ascending order. However, since the I% array now follows the primary array, the sorting characteristics of the I% array (ID numbers in descending order) take precedence over the F$ array. If the names are identical, the I% array determines their order.

Using Tag Arrays

In addition to using secondary arrays, you can specify tag arrays on a sort command. If an array (other than the primary array) has no directional sign, it is a tag array. Tag arrays do not affect the results of a sort. Any reordering that occurs in the primary sort array also occurs in a tag array.

If you use both tag and secondary arrays, specify secondary arrays on the sort command line before tag arrays. If you include more than 1 tag array, separate them with commas.

If the following arrays are in memory:

A$(1)	A$(2)	A$(3)	A$(4)	A$(5)	A$(6)
JONES	JONES	JONES	WILLIAMS	JOHNSON	JONES

F$(1)	F$(2)	F$(3)	F$(4)	F$(5)	F$(6)
ROBIN	BILLY	BETTY	RICHARD	CHARLES	BOBBY

this command (using F$ as a tag array):

SYSTEM"RUN BSORT 6,A$(1),F$ [ENTER]

can produce these results:

A$(1)	A$(2)	A$(3)	A$(4)	A$(5)	A$(6)
JOHNSON	JONES	JONES	JONES	JONES	WILLIAMS

F$(1)	F$(2)	F$(3)	F$(4)	F$(5)	F$(6)
CHARLES	ROBIN	BILLY	BOBBY	BETTY	RICHARD

BSORT sorts the last names in correct ascending order, but the tag array F$ does not affect the order in which BSORT arranges the data. Whenever exact data matches occur (as is the case with the name JONES) reordering of these elements is random. To sort your information more precisely, use a secondary sort array. If you want only to shift information along with the array you sort, use a tag array.

MID$ Sorting

When sorting string arrays, you can specify a mid-portion of the string as the sort key with primary and/or secondary arrays. This allows you to sort a specified part of the string.

As an example, consider the following 2 string arrays:

L$(Ø)	L$(1)	L$(2)	L$(3)	L$(4)
D. BROWN	R. SMITH	T. JONES	R. SMITH	P. JONES

F$(Ø)	F$(1)	F$(2)	F$(3)	F$(4)
BR, DALE	SM, BOB	JO, TERRY	SM, RICH	JO, PETE

Array L$ contains a first name initial, followed by a period, a space and a last name. Array F$ contains the first 2 characters of the last name, followed by a comma, a space, and the first name. To sort the array by last name/first name, type:

SYSTEM"RUN BSORT 5,L$(Ø)(4,7),+F$(5,6) [ENTER]
to achieve these results:

L$(Ø)	L$(1)	L$(2)	L$(3)	L$(4)
D. BROWN	P.JONES	T. JONES	R. SMITH	R. SMITH

F$(Ø)	F$(1)	F$(2)	F$(3)	F$(4)
BR, DALE	JO, PETE	JO, TERRY	SM, RICH	SM, BOB

L$ is the primary array, and F$ is the secondary array. Both arrays are in ascending order.

The mid-string information, 2 integer numbers enclosed in parentheses, immediately follows the subscript for the primary array. For the secondary array, mid-string information immediately follows the type declaration tag for the secondary array name. The first number specifies the position at which to start sorting the

string--in this case, the fourth character (the first character of the last name). No comma separates MID$ sort information from the last piece of information associated with the array.

The second number indicates the number of characters to sort. In this case, 7 characters of each string (starting at Position 4 of the string) comprise the sort key for the primary sort array.

Similarly, mid-string information sorts the secondary array, F$, beginning at Position 5 (the first character of the first name) in each element of the array, and extending for 6 characters into each string.

BSORT does not check the validity of the mid-string values if they are less than 256. If the position starts at a point exceeding the entire length of the string, that particular element of the array receives a null value. If the MID$ position starts within the string, but uses more characters than remain in the string as sort criteria, BSORT uses only the remaining characters.

For example, if Array A$ contains these values:

```
A$(1)="HI   "
A$(2)="BYE  "
A$(3)="THIS IS THE END"
```

these commands produce the adjacent results:

1. **SYSTEM"RUN BSORT 3,A$(1)(1,3)"**	BYE HI THIS IS THE END
2. **SYSTEM"RUN BSORT 3,A$(1)(2,4)"**	THIS IS THE END HI BYE
3. **SYSTEM"RUN BSORT 3,A$(1)(3,2)"**	HI BYE THIS IS THE END

In Example 1, BSORT uses the first through third characters of each string to sort the array. In Example 2, BSORT uses the second through fifth characters of each string to sort the array. In Example 3, BSORT uses the third to fourth characters of each string to sort the array. Since the first element has only 2 characters, its sort value is null; therefore, it appears first in ascending order.

Generating an Index Array

In some cases, such as reading data into an array from a random access file, you may not want to physically reorder an array. You use BSORT to create an index array contains the element numbers of the sorted array. BSORT reorders the index array so its values represent the sorted order of the elements in the primary array. For example, assume that the following arrays are currently in memory:

P$(1)	P$(2)	P$(3)	P$(4)	P$(5)	P$(6)	P$(7)
WILLIAMS	SMITH	JONES	BROWN	GREEN	JOHNSON	RICH

I%(1)	I%(2)	I%(3)	I%(4)	I%(5)	I%(6)	I%(7)
Ø	Ø	Ø	Ø	Ø	Ø	Ø

This sort command:

```
SYSTEM"RUN BSORT 7,*I%(1),P$(1)" [ENTER]
```

creates the index array I%:

I%(1)	I%(2)	I%(3)	I%(4)	I%(5)	I%(6)	I%(7)
4	5	6	3	7	2	1

Although the sort command does not alter the primary sort array (P$), the values in the index array (I%) reflect the sorted order of P$. For example, I%(1) has a value of 4.

If you want the ascending, sorted order of Array P$, access Element 4 of Array P$ first. To print the contents of Array P$ in sorted order, use I% as an index:

```
FOR L%=1 TO 7                    FOR L%=1 TO 7
PRINT P$(I%(L%))       or        M%=I%(L%):PRINT P%(M%)
NEXT L%                          NEXT L%
```

To include an index array, specify an asterisk (*), followed by the index array name after the number of items to sort. The array must be a 1-dimensional, integer-type array with an explicit type declaration tag. The integer subscript number indicates the starting position of the indexed formation. The array must be as large as the number of items it sorts.

The subscript number used with the index array does not necessarily parallel the subscript number specified in the sorted array. For example, assume that the following integer array exists in memory:

I%(1)	I%(2)	I%(3)	I%(4)	I%(5)	I%(6)	I%(7)	I%(8)	I%(9)	I%(1Ø)
1ØØ	2ØØ	3ØØ	4ØØ	5ØØ	6ØØ	7ØØ	8ØØ	9ØØ	1ØØØ

Using this as an index array, the following sort command on Array P$:

SYSTEM"RUN BSORT 4,*I%(6),P$(2)" [ENTER]
produces these results:

I%(1)	I%(2)	I%(3)	I%(4)	I%(5)	I%(6)	I%(7)	I%(8)	I%(9)	I%(1Ø)
1ØØ	2ØØ	3ØØ	4ØØ	5ØØ	4	5	3	2	1ØØØ

This command sorts 4 elements (Elements 2-5) of the P$ array, and stores the index information in the I% array, starting at Element 6. It does not affect Elements 1-5 and 1Ø, because the index array only stores the sorted element numbers of the primary array (in this case,

Elements 2-5). You cannot store index information beyond the end of the index array.

If the I% array contains 11 elements (Ø-1Ø), this BSORT command causes error:

```
SYSTEM"RUN BSORT 4,*1%(8), P$(2) [ENTER]
```

Finally, you can perform indexed sorts using all of the previously defined sort parameters (for example, mid-string and secondary arrays). Once you specify the index array in the sort command, the syntax remains the same. Because BSORT does not reorder any arrays used in an index sort, tag arrays are meaningless in the sort command.

Sorting 2-Dimensional Arrays

The same concepts applicable to 1-dimensional arrays apply to 2-dimensional arrays. To retrieve and sort the key information, specify a row of the array, and the number of the column at which to start the sort. The number of columns equals the number of elements you want to sort. Therefore, reordering an array transposes an entire column of data.

For example, assume Array A$ contains:

		COLUMN 1	2	3	4	5
	1	DALE	DAN	DON	DICK	DOC
R	2	BROWN	JONES	SMITH	GREEN	PETERS
	3	25	34	19	53	42
O	4	BOSTON	BUTTE	BALT	PHIL	PITT
	5	Ø3Ø21	78654	23376	19769	16511
W	6	MA	MT	MD	PA	PA
	7	REP	REP	CLIENT	ADV	STOCK

To sort this array by last name in ascending order, this BSORT command:

SYSTEM"RUN BSORT 5,A$(2,1)" [ENTER]

produces these results:

		COLUMN 1	2	3	4	5
	1	DALE	DICK	DAN	DOC	DON
R	2	BROWN	GREEN	JONES	PETERS	SMITH
	3	25	53	34	42	19
O	4	BOSTON	PHIL	BUTTE	PITT	BALT
	5	Ø3Ø21	19769	78654	16511	23376
W	6	MA	PA	MT	PA	MD
	7	REP	ADV	REP	STOCK	CLIENT

The command indicates there are 5 items to sort; sorting begins at Column 1 in Row 2, and continues for 5 columns. When BSORT moves an element, it also moves the elements in the other rows of the same column.

To exchange positions of Columns 4 and 5 in the original A$ array, issue the command:

SYSTEM"RUN BSORT 2,A$(5,4)" [ENTER]

This sort uses information in Row 5 as the key, begins at Column 4, and includes 2 columns (Columns 4 and 5). Since 16511 is less than 19769, a reordering occurs.

With the A$ array in memory, you can create an index array (I%) sorting the information in Row 3 in descending order with the following command:

```
SYSTEM"RUN BSORT 5,*I(1),-A$(3,1)" [ENTER]
```

I%(1)	I%(2)	I%(3)	I%(4)	I%(5)
4	5	2	1	3

When you index a 2-dimensional array, the index array stores the column position of the sorted array, leaving the sorted array unchanged.

Using 2-Dimensional Secondary and Tag Arrays

Using 2-dimensional secondary and tag arrays resembles sorting 2-dimensional arrays. There must be at least as many elements in the secondary or tag array as there are columns in the primary array.

Tag arrays do not need subscripts. Reordering of columns in the tag array corresponds to reordering in the primary array. The entire column transposes, no matter how many rows in the array, with any reordering.

The same reordering rules apply to 2-dimensional secondary arrays. However, a secondary array requires a subscript indicating the number of the key information. Assume that the following arrays exist in memory:

A%(1)	A$(2)	A$(3)	A$(4)	A$(5)
BROWN	ADAMS	BROWN	ADAMS	BROWN

ARRAY B$

COLUMN

		1	2	3	4	5
R	1	PRES	VP	MGR	SALES	DIST
O	2	25	53	34	42	19
W	3	DALE	DOC	DAN	DICK	DON

A$ is the primary array, and Row 3 of B$ is the secondary sort array. The following BSORT command first sorts the data by last name, then by first name:

SYSTEM"RUN BSORT 5,A$(1),+B$(3) [ENTER]

A$(1)	A$(2)	A$(3)	A$(4)	A$(5)
ADAMS	ADAMS	BROWN	BROWN	BROWN

ARRAY B$

COLUMN

		1	2	3	4	5
R	1	SALES	VP	PRES	MGR	DIST
O	2	42	53	25	34	19
W	3	DICK	DOC	DALE	DAN	DON

In a BSORT command, you can use the same 2-dimensional secondary array more than once, provided you specify a different row in each case. In fact, if the primary array is 2-dimensional, you can specify a row other than the primary sort row as a secondary sort array. To sort a 2-dimensional array (Z$) primarily by last name (Row 2), and secondarily by first name (Row 1), use the following command:

```
SYSTEM"RUN BSORT 5,Z$(2,1),+Z$(1)" [ENTER]
```

You can also sort a 2-dimensional secondary array with mid-strings. As is the case with a 1-dimensional primary array, the mid-string information immediately follows the row subscript of the secondary array. For example, in the command:

```
SYSTEM"RUN BSORT 1Ø,XX%(2,5),+C$(3)(19,8)" [ENTER]
```

XX% is the primary array. BSORT sorts 1Ø columns of Row 2, starting at Column 5 (Columns 5-14). In C$, the secondary array, you sort Columns 5-14 in Row 3. The sort key begins at Position 19 of each element and extends for 8 characters (Columns 19-26).

Using a Variable to Pass the Sort Command

A SYSTEM command cannot exceed 79 characters in the quotation marks. BSORT allows you to include sort parameters as string variables. The string variables can contain up to 255 characters.

For example, the following sort command replaces a lengthy number of parameters with the variable PARM$:

```
PARM$ = "1Ø,*II%(1),-AA$(4,1)(15,2Ø),+AC#(3),-SD$(7)(13,8)"
 SYSTEM "RUN BSORT $PARM$"
```

Precede the string variable with a ($) in the system command.

MOD324--PROGRAM CONVERSION UTILITY

MOD324 converts MODEL III BASIC programs to a form that MODEL 4 BASIC can read. The MODEL III program must be on a diskette formatted by either MODEL 4 TRSDOS or MODEL III LDOS. To move a program from a MODEL III TRSDOS diskette to a MODEL 4 TRSDOS diskette, use CONV. Store the MODEL III program on a disk in compressed format. Do not save it in ASCII.

Note: Some program commands and sequences which function properly on the MODEL III do not work on the MODEL 4. MOD324 attempts to flag every possible error situation. However, it cannot guarantee that a program it converts will work, even if it does not indicate manual corrections.

The syntax for MOD324 is:

MOD324 filespec1 filespec2 [(parameter,...)]

filespec1 stores MODEL III program to convert in compressed format. If you omit filespec1, MOD324 prompts for it.

filespec2 contains the converted program. If you omit filespec2, MOD324 prompts for it. If the filename already exists, MOD324 overwrites it. If it does not exist, MOD324 creates it.

Parameters are:

MODIFY adjusts numeric constants in PRINT@ statements to the corresponding value on the MODEL 4 video. Does not adjust PRINT TAB statements.

CENTER[=n] n indicates the additional offset value to add to all PRINT@ positions changed by MODIFY. If you include CENTER, you must include MODIFY. If you omit n, MOD324 assumes 328 (4 lines, 8 columns).

PRINT	sends manual corrections to the printer. If you omit PRINT, MOD324 displays manual corrections on the display.
WIDTH[=n]	n specifies the number of characters per printed line when you include PRINT. n may be an integer in the range 9 to 255. If you omit WIDTH, MOD324 assumes 8Ø characters per printed line.

Parameters may be abbreviated to their first character.

General Information

MOD324 changes tokenized key words and symbols in the MODEL III program to their corresponding ASCII representation in the MODEL 4 program.

MOD324 removes values specifed in CLEAR statements because on the MODEL 4, the CLEAR statement functions differently. For example, MOD324 changes the MODEL III statement, CLEAR 5ØØØ to CLEAR in the MODEL 4 program.

MOD324 inserts a space in MODEL 4 text after key words that aren't followed by information in parentheses (FOR, TO, NEXT) and after variables or constants that precede a key word, but are not separated from the key word by a terminator. For example, in the statement: IF A%=1ØTHEN A%=5, MOD324 inserts a space between the Ø and the T.

The maximum line length in Model 4 BASIC is 254 characters. When MOD324 adds additional spaces in a statement, it can cause the line length to exceed this limit. If that occurs, MOD324 truncates the line and displays the truncated text. Then, you can create a new line to add to the MODEL 4 BASIC program that contains the truncated text.

Note: The truncation of a line can affect program logic.

Assume that the following line exists in a MODEL III program:

```
1Ø FORLL=1TO1Ø:FORLK=1TO2Ø:FORLP=1TO3Ø:LPRINTTAB(2Ø)"This
   is an example of a converted line being too
   long":LPRINTTAB (2Ø)"The value of LL
   is";LL:LPRINTTAB(2Ø)"The value of 1k
   is";LK:LPRINTTAB(2Ø)"The value of LP
   is";LP:NEXTLP:NEXTLK:NEXTLL:PRINTTAB(2Ø)"Done"
```

After the conversion MOD324 stores the line in the Model 4 program as:

```
1Ø FOR LL=1 TO 1Ø:FOR LK=1 TO 2Ø:FOR LP=1 TO 3Ø:LPRINT
   TAB(2Ø)"This is an example of a converted line being
   too long":LPRINT TAB(2Ø)"The value of LL is";LPRINT
   TAB(2Ø)"The value of 1k is";LK:LPRINT TAB(2Ø)"The
   value of LLP is";LP:NEXT LP:NEXT LK:NEX
```

MOD324 displays:

```
The following lines may need manual correction:

1Ø   TAB
1Ø   -Line truncated, should be extended as follows:
   T LL:PRINTTAB(2Ø)"Done"
```

In a MODEL III program, you may omit the word THEN in an IF-THEN statement. For example:

```
IF A=1 A=2
```

On the MODEL 4 this statement causes a syntax error. MOD324 flags any IF statement not followed by a THEN.

MOD324 does not change information that appears in the MODEL III program file as ASCII, that is enclosed in quotation marks, or that follows an apostrophe.

Some program statements that exist in MODEL III BASIC have no meaning on the MODEL 4. Other program statements which exist in both BASICs function differently in each.

MODEL III commands that MOD324 flags as possibly needing manual correction are:

CLOAD	POINT
CMD	POS
CSAVE	PRINT@
ERR	PRINT TAB
IF	PRINT #-1, PRINT #-2
INP	RESET
INPUT #-1, INPUT #-2	SET
NAME	SYSTEM
OUT	TIME$
PEEK	USR
POKE	

MOD324 flags every PRINT@ and PRINT TAB statement because the video sizes differ between the Model III (64x16) and Model 4 (8Øx24). The MODIFY parameter and the number you specify for CENTER control how MOD324 adjusts values in PRINT statements.

Program Usage

To convert a program, type **MOD324 [ENTER]** at the TRSDOS Ready prompt. MOD324 displays:

```
Input Filespec?
Output Filespec?
```

All entries must follow the rules associated with valid filespecs. You may press **[BREAK]** in response to either prompt to return to TRSDOS Ready. If your answer to these prompts is incorrect, the screen displays the appropriate error message.

The first prompt requests the name of the MODEL III program. Enter the filespec. If you omit adrivespec, MODD324 searches all active drives. If the file has an extension, include it. MOD324 does not assume /BAS.

The second prompt requests the name of the file to contain the converted program. If the filespec does not exist, MOD324 creates it. If the filespec exists, the converted program overwrites that file. To assure that the file writes to the proper place, include a drivespec with the output filespec. If you omit drivespec, MOD324 writes the output

file to the first drive containing the file, or to the first available drive if the file does not exist on any drive in the system.

You can enter both filespecs on the command line. For example, if you want to create the MODEL 4 program TEST/M4 on Drive 2 from the MODEL III program TEST/BAS on Drive 1, enter the following command:

MOD324 TEST/BAS:1 TEST/M4:2 [ENTER]

If you specify 1 filespec on the command line, MOD34 assumes it is the MODEL III file and prompts for the output filespec.

Assume that you have created the following MODEL III BASIC program and saved it in compresssed form under the filespec SAMPLE/BAS:

```
1Ø CLEAR 5ØØØ:DEFINTA-N:DEFSTRS,T
2Ø CLS:FORL = 1TO1Ø
3Ø PRINTTAB (5)"This is Line";L;"on the MOD III
   video";TAB (45)"Position 45"
4Ø NEXT L
```

To convert this program for use on the MODEL 4 with the filename SAMPLE/M4 on Drive 2, enter the following command:

MOD324 SAMPLE/BAS SAMPLE/M4:2 [ENTER]

MOD324 creates an ASCII file containing the converted program and displays possible manual program corrections. The following is a listing of the file SAMPLE/M4:

```
1Ø CLEAR:DEFINT A-N:DEFSTR S,T
2Ø CLS;FOR L=1 TO 1Ø
3Ø PRINT TAB (5)"This is Line";L;" on the MOD III
   Video";TAB(45)"Position 45"
4Ø NEXT L
```

In Lines 1Ø, 2Ø and 3Ø, MOD324 inserts spaces as needed. MOD324 does not insert a space in Line 4Ø because there was already a space between the T in NEXT and the variable L. MOD324 strips the value in the CLEAR statement.

After the conversion, MOD324 displays:

```
The following lines may need manual correction:

3Ø TAB,TAB

File output completed
```

As MOD324 creates the MODEL 4 file, it displays line numbers that contain possible problem key words. See the previous list. Commas separate multiple key words on the same line. In this example, the key words PRINT TAB appear twice in Line 3Ø. When TAB appears in a manual correction listing, it implies a PRINT TAB sequence. IF you use TAB with an LPRINT statement, MOD 324 does not flag it.

After MOD324 creates the MODEL 4 file, you can make manual corrections. In this example, you can run the program as it is. However, if MOD324 flags any key words (such as SET) because they do not exist in MODEL 4 BASIC, remove them. Modify lines that contain key words which cause unpredictable results (such as a POKE of video ram).

MODIFY and CENTER Parameters

Consider the following MODEL III program saved in compressed format as CENTER/BAS. It draws a box on the first 15 lines of the screen, and displays messages on the last line and in the middle of the box:

```
5 CLEAR 2ØØØ
1Ø CLS
2Ø PRINT@Ø,CHR$(151);STRING$(62,131);CHR$(171)
3Ø PRINT@64,CHR$(149):PRINT@127,CHR$(17Ø)
4Ø PRINT@128,CHR$(149):PRINT@191,CHR$(17Ø)
5Ø PRINT@192,CHR$(149):PRINT@255,CHR$(17Ø)
6Ø PRINT@256,CHR$(149):PRINT@319,CHR$(17Ø)
7Ø PRINT@32Ø,CHR$(149):PRINT@383,CHR$(17Ø)
8Ø PRINT@384,CHR$(149):PRINT@447,CHR$(17Ø)
9Ø PRINT@448,CHR$(149):PRINT@511,CHR$(17Ø)
1ØØ PRINT@512,CHR$(149):PRINT@575,CHR$(17Ø)
11Ø PRINT@576,CHR$(149):PRINT@639,CHR$(17Ø)
12Ø PRINT@64Ø,CHR$(149):PRINT@7Ø3,CHR$(17Ø)
```

```
13Ø PRINT@7Ø4,CHR$(149):PRINT@767,CHR$(17Ø)
13Ø PRINT@768,CHR$(149):PRINT@831,CHR$(17Ø)
15Ø PRINT@832,CHR$(149):PRINT@895,CHR$(17Ø)
17Ø PRINT@896,CHR$(181);STING$(62,176);CHR$(186);
175 PRINT@96Ø,"";TAB(15)"Press Any Key to end this
     Program";
18Ø PRINT@473,"Center of Box";
19Ø I$=INKEY$:IFI$<>""THENEND
2ØØ FORL+1TO3Ø:NEXTL
21Ø PRINT@473,"                ":
22Ø I$=INKEY$:IFI$<>""THENEND
23Ø FORL=1TO2Ø:NEXTL:GOTO18Ø
```

The following command converts CENTER/BAS to CENTER/M4:

MOD324 CENTER/BAS CENTER/M4:3 [ENTER]

MOD324 displays:

```
File CENTER/M4:3

The following lines may need manual correction:

2Ø   PRINT@(Ø)
3Ø   PRINT@(64),PRINT@(127)
4Ø   PRINT@(128),PRINT@(191)
5Ø   PRINT$(192),PRINT@(255)
6Ø   PRINT@(256),PRINT@(319)
.
.
.
14Ø  PRINT@(786),PRINT@(831)
15Ø  PRINT@(832),PRINT@(895)
17Ø  PRINT@(896)
175  PRINT@(96Ø),TAB
18Ø  PRINT@(473)
21Ø  PRINT@(473)
```

All PRINT@ commands use numeric constants to represent print positions. If you run CENTER/M4 without performing manual corrections the program does not draw a box on the screen.

To convert a MODEL III PRINT@ position to a MODEL 4 PRINT@ position:

1. Divide the Model III positon by 64.

2. Multiply the quotient by 8Ø and add the remainder to that product. The result is the MODEL 4 PRINT@ position.

To convert all PRINT@ positions, use the MODIFY parameter. The MODEL 4 program contains the adjusted PRINT@ values.

To convert the program CENTER/BAS and include the MODIFY parameter to convert screen positions, type:

MOD324 CENTER/BAS CENTER/M4:3 (M) [ENTER]

MOD324 displays:

```
File CENTER/M4:3

The following lines may need manual correction:

2Ø    PRINT@(Ø=>Ø)
3Ø    PRINT@(64=>8Ø),PRINT@(127=>143)
4Ø    PRINT@(128=>16Ø),PRINT@(191=>223)
5Ø    PRINT@(192=>24Ø),PRINT@(255=>3Ø3)
6Ø    PRINT@(256=>32Ø),PRINT@(319=>3Ø3)
7Ø    PRINT@(32Ø=>4ØØ),PRINT@(383=>463)
8Ø    PRINT@(384=>48Ø),PRINT@(447=>543)
9Ø    PRINT@(448=>56Ø),PRINT@(511=>623)
1ØØ   PRINT@(512->64Ø),PRINT@(575=>7Ø3)
11Ø   PRINT@(576=>72Ø),PRINT@(639=>783)
12Ø   PRINT@(64Ø=>8ØØ),PRINT@(7Ø3=>863)
13Ø   PRINT@(7Ø4=>88Ø),PRINT@(767=>943)
14Ø   PRINT@(768=>96Ø),PRINT@(831=>1Ø23)
15Ø   PRINT@(832=>1Ø4Ø),PRINT@(895=>11Ø3)
17Ø   PRINT@(896=>112Ø)
175   PRINT@(96Ø=>12ØØ),TAB
18Ø   PRINT@(473=>585)
21Ø   PRINT@(473=>585)
```

The adjustments made to Line 4Ø translate the original PRINT@ position of 191 into 223. Running the program CENTER/M4 draws a box in the upper left corner of the screen. You do not need to manually correct the PRINT@ positions. Because PRINT TAB commands (see Line 175) refer to column position only, MODIFY does not adjust them.

Because the MODEL 4 screen is larger than the MODEL III screen, you can overlay a MODEL III screen--up to 8 rows and 16 columns--onto a portion of the MODEL 4 screen. To further adjust PRINT@ positions use the CENTER parameter with the MODIFY parameter. The default value for the CENTER parameter is 328 (4 rows, 8 columns).

To convert CENTER/BAS and include the MODIFY and CENTER parameters to draw the box in the center of the MODEL 4 screen, type:

MOD324 CENTER/BAS CENTER1/M4:3 (M,C) [ENTER]

MOD324 displays:

```
File CENTER/M4:3

The following lines may need manual correction:

2Ø   PRINT@(Ø=>328)
3Ø   PRINT@(64=>4Ø8),PRINT@(127=471)
4Ø   PRINT@(128=>488),PRINT@(191=>551)
5Ø   PRINT@(192=>568),PRINT@(255=>631)
6Ø   PRINT@(256=>648),PRINT@(319=>711)
7Ø   PRINT@(32Ø=>728),PRINT@(383=>791)
8Ø   PRINT@(384=>8Ø8),PRINT@(447=>871)
9Ø   PRINT@(448=>888),PRINT@(511=>951)
1ØØ  PRINT@(512=>968),PRINT@(575=>1Ø31
11Ø  PRINT@(576=>1Ø48),PRINT@(639=>1111)
12Ø  PRINT@(64Ø=>1128),PRINT@(7Ø3=>1191)
13Ø  PRINT@(7Ø4=>12Ø8),PRINT@(767=>1271)
14Ø  PRINT@(768=>1288),PRINT@(831=>1351)
15Ø  PRINT@(832=>1368),PRINT@(895=>1431)
17Ø  PRINT@(896=>1448)
17Ø  PRINT@(96Ø=>1528),TAB(15=>23)
18Ø  PRINT@(473=>913)
21Ø  PRINT@(473=>913)
```

In Line 4Ø, the original PRINT@ position of 191 translates into 551. CENTER1/M4 draws a box in the center of the screen (with the upper left corner of the box positioned at Row 4, Column 8). The program does not require manual correction because CENTER adjusts PRINT TAB values.

The CENTER parameter affects column positioning by moving the entire screen. It adjusts PRINT TAB commands (see Line 175) by adding the value of the column offset to the numeric constants in PRINT TAB statements. If you use Ø as a column offset (CENTER=8Ø, 16Ø, 24Ø, etc.), CENTER does not have to adjust PRINT TABS. To determine the column offset, divide the value specified with CENTER by 8Ø. The remainder after that division is the column offset.

Although you can use any value with CENTER, some values produce undesirable results. Avoid offsets of more than 8 rows and/or 16 columns. This table lists some practical CENTER value ranges and the resulting row offset:

CENTER=Range	Row Offset
Ø-16	Ø
8Ø-96	1
16Ø-176	2
24Ø-256	3
32Ø-336	4
4ØØ-416	5
48Ø-496	6
56Ø-576	7
64Ø-656	8

When PRINT@ and PRINT TAB statements use numeric expressions as print position values, MOD324 does not adjust the values. This MODEL III program, saved in compressed format as CNTLOOP/BAS, draws a box on the video using a FOR-NEXT loop.

```
5 CLEAR 2ØØØ
1Ø CLS
2Ø PRINT@Ø,CHR$(151);STRING$(62,131);CHR$(171)
25 FORL =1TO13:A1=L*64:PRINT@A1,CHR$(149):PRINT@A1+63,
CHR$(17Ø):NEXTL
17Ø PRINT@896,CHR$(181);STRING$(62,176;CHR$(186);
172 MC$="Center of Box":MB$="Press Any key to end this
Program"
174 M1=LEN(MC$):M2=LEN(MB$):CM=7*64-(M1/2)
175 PRINT@96Ø,"";TAB((64-M2)/2);MB$;
18Ø PRINT@CM,MC$;
19Ø I$=INKEY$:IFI$<>""THENEND
2ØØ FORL=1TO3Ø:NEXTL
21Ø PRINT@CM,STRING$(M1,32);
22Ø I$=INKEY$:IFI$<>""THENEND
23Ø FORL =1TO2Ø:NEXTL:GOTO18Ø
```

To convert this program, type:

MOD324 CNTLOOP/BAS CNTLOOP/M4:3 (M,C) [ENTER]

MOD324 displays:

```
File CNTLOOP/M4:3

The following lines may need manual correction:

    2Ø    PRINT@(Ø=>328)
    25    PRINT@(EXP),PRINT@(EXP)
    17Ø   PRINT@(896=>1448)
    175   PRINT@(96Ø=>1528),TAB(EXP)
    18Ø   PRINT@(EXP)
    21Ø   PRINT@(EXP)
```

MOD324 adjusts the value in Line 2Ø. Because the print position in Line 25 is a numeric expression, it does not change that value in CNTLOOP/M4. It displays (EXP) for the numeric expression and MOD324 retains the values of the MODEL III program.

BLANK PAGE

UNKILL--RESTORE A PREVIOUSLY DELETED FILE

REMOVE and PURGE reset bits in the directory entries and reset the HIT and GAT. They do not erase a file. If you do not overwrite the directory entry of the file with another entry, and do not reallocate the space it occupies, UNKILL can restore a file deleted with REMOVE or PURGE. The syntax is:

UNKILL filename/ext:d

There are no parameters and no abbreviations are allowed. Type the command, followed by <ENTER> at the TRSDOS Ready prompt.

The filename, extension, and drivespec must match the previous file. If you omit drivespec, UNKILL assumes Drive Ø. It does not search other active drives. If the file contains a password, UNKILL restores it.

To resore the previously deleted file ACCOUNT/DAT on Drive 1, type:

UNKILL ACCOUNT/DAT:1 [ENTER]

If UNKILL is successful, the screen displays:

File successfully restored.

Error Messages

Illegal Filename

Check the spelling of the filename or the syntax and try the command again.

Directory Read Error

Part or all of the directory is unreadable. Try the command again. If it produces the same error, change drives and try the command again. If the same error occurs on more than 1 drive, you cannot recover the file because of a faulty diskette.

Directory Write Error

UNKILL cannot verify the attempted write. Follow the procedure for Read Error.

File Already Alive

The file is still active. If the deleted file has the same filename as an existing file, RENAME the existing file and try the command again.

No file by that name

No such filename is in the directory. If another file has already overwritten the directory entry, you cannot reactivate.

Cannot Unkill File

Another file is using the disk space of the deleted file. You cannot reactivate.

Can't Log in Disk

Check the drive. A number of things can be wrong, such as no disk in the drive or the drive door isn't closed.

THE BASIC ANSWER USER GUIDE

Introduction

The BASIC Answer, (TBA) is a processing utility designed to help you create meaningful and structured BASIC programs. Use TBA only with the Model 4 TRSDOS operating system, version 6.Ø2.ØØ or later. TBA uses all the commands and concepts available in interpretive BASIC, plus additional concepts which further define and structure program code. This is especially useful when you debug or modify rarely used programs.

This User Guide explains syntax and defines statements you need for the TBA processor. The accompanying tutorial manual contains detailed explanations; 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.

To use TBA, type **TBA [ENTER]** at the TRSDOS Ready prompt. No parameters are allowed on the command line.

Creating Source Code

To create a source code, you can use a word processor such as SCRIPSIT, SUPERSCRIPSIT; a text processors such as ALEDIT; or the BASIC Interpreter. Save the source file in ASCII format. TBA allows a maximum source line of 24Ø characters.

Upper and Lower Case Usage in Source Code

As in the BASIC Interpreter, TBA converts some lower case text to upper case, such as characters not within quotation marks or remark statements. You can override this by including the Differentiate Case (DC) parameter. When you include DC, TBA interprets identical letters, in identical words, but with unmatched cases as distinct. For example, the word LOOP, Loop, loop and LOop are 4 unique labels, variables or expressions.

Labels

When you create programs with BASIC, you include line numbers in the statements. With TBA you use labels to reference locations or branches in source code. When you use a label to identify a procedure, it is the first phrase on a line and may be followed by a space, colon, or carriage return. For example:

```
@DELAY.LOOP
     FOR LOOP% = 1 TO 2ØØØ: NEXT
RETURN
```

the first line identifies the procedure @DELAY.LOOP. The second line contains the BASIC statements in the procedure.

Other BASIC statements may follow identifying labels. Separate the label from the first BASIC statement with a colon. For example:

```
@DELAY.LOOP : FOR LOOP% = 1 TO 2ØØØ: NEXT
RETURN
```

When you specify a label in a statement, it indicates a branch to the procedure defined by the label. For example:

```
PRINT MESSAGE$(MESSAGE.SUB%) : GOSUB @DELAY.LOOP: CLS

IF IN$ = CHR$(13) THEN GOTO @PROCESS.INPUT

GOSUB @GET.KEYBOARD
```

Creating Labels

A label may contain up to 15 characters. The first character of a label is an @ symbol. The second character must be a letter. It may be followed by 12 other characters. The last 12 characters may be a letter, number, or period (.), or underline (_). You cannot use any other special characters or spaces.

Legal labels include:

@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
KYBD.INPUT          : leading <@> missing
```

Variables

TBA uses variables in the same way BASIC uses them. Variable names may contain 3 to 15 characters. The first character of a variable must be letter. The remaining characters can include letters, numbers, periods (.), or underlines (_). The last character must be a type declaration tags defining the kind of data that the variable may contain. Type declaration tags include **%** for an integer, ! for a single precision, # for a double precision, or $ for a string variable.

TBA prohibits the use of certain BASIC reserved words as variable names. These include 2 character BASIC keywords (FN, ON, AS, IF, TO); BASIC keywords ending with a declaration tag (TIMES, MKD$, PRINT#, etc.); and any word in which the first three letters are REM. You can embed reserved words in a variable.

Examples of legal variables :

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

Illegal variables include:

```
TO%                     : reserved word violation
FLAG END%               : no spaces allowed
end.of.sequence1        : too long, no tag
1st.record%             : incorrect lead character
input#                  : reserved word violation
```

You should surround variables with spaces, nonalphanumeric characters such as parentheses and commas, or math and relational operators to prevent misinterpretation.

Global Variable Definitions and Implementation

You can use global varaibles throughout the entire program. Global varaible definitions must be the only statement on a line. The line must begin with an equal sign (=) followed by the variable name. End the definition line with a carriage return. To define more than 1 global variable, separate them with commas. For example:

```
=LOOP%,LOOP1%,TOTAL#,START.TIMES$
```

defines the gloabal varaibles LOOP%, LOOP1%, TOTAL#, and START.TIME$.

Local Variable Definition and Implementation

A local variable has meaning only in the subroutine where you define it. Its value and purpose relates to that procedure alone. Local variables cannot pass information to and from the main body of a program.

Local variable definitions immediately follow the procedure label on the same line. Follow the procedure label with an equal sign = and the variable name. To define more than one local variable, separate the variable names with a comma. End the line with a carriage return. Each procedure allows 1 line of local variable definitions in the proper syntax:

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

defines the local varaibles LOOP%, INK$, and FIELD.LEN in the procedure @INKEY.INPUT.

Array Variables

Define array variables as you define other global or local variables. Array names must conform to the rules for variable names. Use a DIM statement to declare the array size if it contains more than 1Ø elements.

Use the following type of routine to avoid the Redimensioned Array error when dimensioning a local array that contains more than 1Ø elements:

```
WHILE ARRAY$(Ø) <> "DIMENSIONED"
     ERASE ARRAY$
     DIM ARRAY$(36)
     ARRAY$(Ø) = "DIMENSIONED"
WEND
```

The REM Statement

If a REM statement is the first text on a line, TBA passes the line to the object code. If REM appears later in a line, TBA does not recognize it properly and can generate an error. To pass a remark statement to the object code, use REM at the beginning of a line.

If a line begins with an apostrophe ('), TBA deletes the entire line. If there is an apostrophe in the middle of a line, TBA deletes the remaining text on that line. To use a remark in source code only, use an ['].

The RETURN Statement

TBA uses RETURN to signify the end of 1 subroutine and the beginning of the next. Therefore, use the RETURN statement once in a procedure. It must be the last statement in a procedure. You can precede the RETURN statement with a label, as in

```
@EXIT.INPUT : RETURN.
```

TBA Directives

You can embed dircetives in the source code to modify the output of TBA. The directive and its parameters must be the only statement on the line. Directives begin with the asterisk (*) character. The directives are *PRLINES, *LIST ON/OFF, *PAGE, *TITLE, *IF, *expression, and *END.

*PRLINES [n]

n specifies the number of lines per page to print. n can be a number in the range 2Ø to 254. If you omit n, PRLINES assumes 56. *PRLINES sends the printer a top-of-form character (X'0C') after n lines.

To use *PRLINES, activate the TRSDOS printer filter FORMS/FLT. Set the PAGE and LINES parameters to the physical size of the page, usually 66. Use the CHARS parameter to set the number of characters per line. If you omit CHARS, FORMS/FLT assumes 132.

*LIST [ON/OFF]
toggles the process listing on and off. If you omit ON and OFF, *LIST assumes ON.
*PAGE
sends a top-of-form character (X'ØC') to the printer, when the source code encounters *PAGE.

*TITLE string
prints a header at the top of each page. string can be up to 14 characters and you must enclose it in quotes. The header prints in the form:

BASIC Answer "string" September 21, 1982 12:Ø3 A.M. Page 1

*IF, *END, and *expression
process source code on a conditional basis. *IF indicates to test for the presence of expression. An expression is a token that you may include as a response to the Directives prompt or as a source code statement. *IF checks to see if you have included expression. If so, TBA includes the source statements between *IF and *END.

expression is a phrase that conforms to the same rules as variables, but does not have a declaration tag. To specify expression in the source code, precede expression with an *. It must be the only statement on a line. You may also include expression in response to the Directives? prompt. When you do so, omit the asterisk.

*END signifies the conclusion of the conditional block that begins after the previous *IF expression statement. It appears alone on a subsequent line.

You cannot nest *IF/*END in other *IF/*END directives.

Starting TBA

TBA does not allow you to include filespecs or parameters on the command line. It prompts you for them. This section explains how to respond to these prompts. If your response is invalid, TBA redisplays the prompt. To terminate TBA and return to TRSDOS press **[BREAK]** as a response to any of the prompts.

To enter TBA, type **TBA** at the TRSDOS Ready prompt.

The prompts are:

Source Filespec ?

Type the filespec of the the ASCII file you want to process. If you omit the extension, TBA assumes /TBA.

Object Filespec ?

Type any legal filespec or press **[ENTER]**. If you press **[ENTER]**, TBA assumes source filespec/BAS.

Processing Parameters?

Press **[ENTER]** to omit parameters or specify 1 or more parameters to control processing. If you include more than 1, separate them with commas. If you press **<ENTER>**, TBA displays processing information on the screen.

Processing Parameters:

LP prints processing information on the line printer.

TO displays only the object code.

NL indicates no listing. TBA does not print processing information on the screen or line printer.

NX omits a cross reference listing.

DC omit upper to lower case conversion in variables, labels, and expressions.

FC removes extra spaces from the object code.

NOTE: Omit the FC parameter if you intend to run the object files on the Model 4 because it eliminates spaces surrounding reserved words. Use FC to create output files for a BASIC which does not require such spaces, for example, Model III BASIC.

The last prompt is:

Directives?

TBA is prompting you for an expression that the source code uses in *IF directives. To omit expressions, press **[ENTER]**. To include expressions, type 1 or more expressions separated by commas. After you press **[ENTER]** to terminate the expressions, TBA redisplays the directives prompt. You can enter more directives or press **[ENTER]**. The directives prompt repeats until you press **[ENTER]** as the first character of the input line.

TUTORIAL GUIDE

Introduction

TBA procceses non-executable text, referred to as source code. The source code must be in a file in ASCII format. TBA processes the source code into object code, an executable program. You can run this object file as a BASIC program.

In your source code for TBA, you can use any of the BASIC commands. Since TBA processes the source code into object code, you need to follow some guidelines. This section details the differences between writing BASIC programs and writing source code.

Labels

Use labels instead of line numbers in the source code. A label identifies a reference location in TBA just as line number does in BASIC. The label identifies the code that follows it. When you use

that label in a branching statement, TBA transfers program control to the code following the label.

A label may contain up to 15 characters. The first character of a label is an @ symbol. The second character must be a letter. It may be followed by 12 other characters. The last 12 characters may be letters, numbers, a period (.), or an underline (_). Use the period and underline to break up words within the labels to improve readability. You cannot use any other special characters or spaces.

Valid labels include:

```
@field.buffer1      @strobe.kbd         @Process_data
@check.file         @CHECK.FILE         @back_to_TRSDOS
@Get.A.Record       @ROUTE.TAKENØØ1     @find_PRIME
```

Invalid labels include:

```
@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
```

TBA converts lower case characters to upper case characters except in REM statements or within quotation marks. To distinguish an upper case label from a lower case label, include the DC parameter. If you include DC, TBA references the labels @CHECK.FILE and @check.file as 2 distinct labels.

If you include the DC parameter, use the label exactly as you defined it originally--a precise character for character match. If you omit the DC parameter, use variable names that are different from each other in more than case of the characters. Decide whether to distinguish between upper and lower case before you write source code.

Since labels are analogous to line numbers, you can only define a label once within the source code. A label is the first text on a line, unless it acts as a branch reference. Then it follows the appropriate BASIC branching keyword. For example, GOTO @PLAN.3, GOSUB @INPUT.PROCESS; are branching references.

In most high-level languages, the term procedure denotes a block of code referenced by a label. In BASIC, these procedures are called

subroutines. Because TBA source code is not BASIC code, TBA documentation uses the term procedures rather than subroutines.

This program uses normal BASIC syntax. The procedure flashes a message on the screen until you press [ENTER]. This example represents actual source code TBA processes:

```
@start.program
' *** Define Variables for TBA processing ***
=loop%,in$
'
gosub @flash.message
@end.program
stop
'
@flash.message
  cls
  for loop% = 1 to 2Ø
       in$ = inkey$
          if in$ = chr$(13) then goto @end.flash.mssg
  next loop%
  print @(1Ø,1Ø),"Flashing Message - Press <ENTER> to continue"
  for loop% = 1 to 5Ø
       in$ =inkey$
       if in$ = chr$(13) then goto @end.flash.mssg
  next loop%
  goto @flash.message
@end.flash.mssg
  return
```

@start.program and @end.program are reference points in the main portion of the program. Because these labels are not the object of a branching instruction, they are little more than commentary entries in this program. To conditionally execute the various portions of the main program, expand this program with branching references to both labels.

@flash.message indicates the start of the flashing message procedure to you and TBA. The main program typically references a procedure in a branching instruction. In this program, the reference is GOSUB @flash.message.

The RETURN statement signifies the end of a procedure. It must be the last statement in a procedure. To exit the procedure from anywhere

else within the procedure, you must branch to a label immediately preceding the RETURN.

In this program, @end.flash.mssg informs you of the end of the procedure and serves as a label for branching to the end of the procedure. RETURN informs TBA of the end of the procedure and returns control to the program statement immediately following the call to the procedure.

The lines containing the statements GOTO @flash.message and GOTO @flash.end show how to reference a label as a point of transfer within a procedure.

To include another statement in a line after a label reference, separate them by a non-alphanumeric character, usually a space or a colon. This distinguishes the label from the next statement. For example, to reference the procedure on a specific condition, use:

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

If you omit the space between the label @flash.message and the ELSE, TBA interprets ELSE as part of the label name. This causes an error.

There are 3 advantages to using labels:

labels improve readability
labels are easier to remember than line numbers
labels make it easier to add additional code

In a BASIC program, to add additional code to the subroutine that begins at Line 1ØØØ, you must add the code at Line 995 and change all references to Line 1ØØØ to Line 995 or renumber those lines. To add additional code to a procedure, insert the additional code between the procedure label and the first statement in the procedure. A statement such as GOSUB @flash.message (instead of GOSUB 1ØØØ) informs you of the next sequence of events.

Choose labels that describe the function of the routine. For example, @flash.mssg is more informative that @routine2.

BASIC lets you merge programs and routines together, but you still need absolute line references. When you write source code, you can merge procedures for different programs with the main text of the program without absolute line number references. With TBA, you can

create a library of procedures to use over and over again with many different programs.

You use labels instead of line numbers when you write source because TBA inserts line numbers into the object code. It assigns lines numbers to all lines in the source code.

Using the @flash.mssg routine as an example, assume that after processing, TBA assigns line number 5ØØ to the first CLS line in the procedure. The object code refers to this source code label as GOSUB 5ØØ or GOTO 5ØØ. If you include absolute lines numbers in the source code, TBA does not change them. They may produce an "Undefined line number" error message.

Knowing the proper syntax for defining and referencing labels enables you to build and maintain structured, easy to follow, meaningful code that differs significantly from BASIC.

Procedures

The main body of a program flows logically using loops and routines that control specific operations on the data. For instance, it prints prompting messages or performs calculations based on the data the program uses. Usually, TBA limits branching to a loop operation, or a movement from a main menu to a specific function of the program and back again.

The procedure completes functions at different logical points throughout the program. For instance, a block of code that controls the input of all data to a program is 1 example of a procedure.

Since the main body of the program accesses a procedure, write procedures so that various functions in the main body of the program can use them. You can write procedures so that more than one program can use them. The more generic a procedure is, the more you can apply it to all programs.

A procedure accepts and retrieves data. Then, the main body of the program manipulates or makes decisions about it. A procedure commonly sets conditions which the main body of the program evaluates. The main body of the program makes decisions based on conditions the

procedure sets. A procedure never makes decisions based on conditions in the main body of the program.

Limit branching statements in a procedure to a branch within the procedure, not to a point in the main body of the program. To exit a procedure, use RETURN.

Use GOSUB to reference procedures from the main body of the program. Use GOTO to branch to another label in the main body of the program.

Variables

Model 4 BASIC allows 4Ø-character variable names and Model III BASIC allows 2-character variable names. Model 4 BASIC does not compress the variable names in memory or on disk. A program with long variable names executes slower and requires more space on disk.

Variable names in TBA can contain up to 14 characters. This helps to make programs more descriptive and meaningful. TBA processes variable names into unique 2-character variable names. They do not decrease efficiency in memory usage or speed, or decrease the transportability of the object code to other BASICs.

BASIC assumes single-precison numerics if you omit a type declaration tag. For example, A = B + C, assigns storage for single-precision numbers, even if integer storage is adequate. BASIC program statements, such as DEFINT A-C, allow the programmer to change the default.

In TBA source code, you must use a type declaration tag as the last character of the variable name. There is no default. This encourages storage efficiency.

TBA supports 2 different types of variables: global and local. Global variables are known throughout the program. Local variables are known only within the procedure in which you define them.

You can use the same variable name in more than 1 procedure if you define that variable as a local variable in each procedure. Using it in 1 procedure does not affect its value in another procedure. You can also define the same variable name as a global variable without affecting the local variable. TBA recognizes each of the global and local variables as a distinct variable.

Variable Names

The first character of a variable name is a letter. The last character must be a type declaraion tag (%, #, !, or $). You can use any combination of letters, numbers, a period (.), or underline (_) between the first letter and type declaration tag. The type declaration tags represent the following types of variables:

%	Integer
!	Single Precision
#	Double Precision
$	String

Variable names less than 2 characters long (excluding type declaration tags) require special handling.

If you do omit the DC parameter, maintain consistency in using upper and lower case when you define and reference variable names. The variable names LOOP1% and loop1% can represent 2 different variables.

Valid variable include:

Loop1%	First.name$
loop1%	last_name$
loop1$	total.dollars#
Account_total#	Spvariable!

Using a different type declaration tag with the same variable name creates unique and distinct variables. If you omit the type declaration tag in a variable name, TBA issues a "Definition Format" error.

When declaring a variable name, you cannot use:

2-character BASIC keywords (such as IF, OR, ON)
BASIC keywords that end with a type declaration tag (such as TIME$,MKD$ INPUT#)
a variable whose first 3 characters are the letters REM

You can embed these BASIC keywords in a variable name. For example, realtime$, spremember%, ifset# are valid variable names. You can use any BASIC keyword that is longer than 2 characters and does not end with a type declaration tag. For example, goto% and lset! are

valid variable names. If you use an invalid variable name, TBA issues an "Illegal Variable" error.

Defining Variables

Before you use any variables, define them in a definition statement.

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 you want to define more than 1 variable, separate the variables with commas. The global variable definition statement must be the only text on a line. This example illustrates how to define 6 global variables:

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

The variables loop1% and loop2% are integer; totaldollars# is double-precision; firstname$ and lastname$ are string; and spvariable! is single-precision.

You can define global variables anywhere in the source code. To give the program more structure, define global variables at the beginning of the source code where they serve as a quick reference list of all global variables currently in use.

To define a variable as local to a procedure, type the variable or list of variables after the label which defines the entry point into the procedure. Use an equal sign (=) to separate the end of the label name from the first character of the first local variable. To define more than 1 variable, separate them with commas.

Use local variables only in the routine in which you define them. The value you assign to a local variable does not affect the other procedures in the program.

This example expands on the previous example to include 3 local variables -- delayloop1%, delayloop2%, and kbdscan$:

```
@start.program
' *** Define Global Variables for TBA processing ***
' No globals used
'
gosub @flash.message
@end.program
```

```
    stop
    '
    @flash.message=delayloop1%,delayloop2%,kbdscan$
    ' *** Define Local Variables with @label.name=variable.list ***
      cls
      for delayloop1% = 1 to 2Ø
           kbdscan$ = inkey$
              if kbdscan$ = chr$(13) then goto @end.flash.mssg
      next delayloop1%
      print @(1Ø,1Ø),"Flashing Message - Press <ENTER> to continue"
      for delayloop2% = 1 to 5Ø
           kbdscan$ =inkey$
           if kbdscan$ = chr$(13) then goto @end.flash.mssg
      next delayloop2%
      goto @flash.message
    @end.flash.mssg
      return
```

Global Versus Local Variables

Because TBA processes source into object code, it transforms global variables throughout the program, and local variables throughout the procedure in which you define them.

The first phase of processing replaces all local variables with distinct 2-character variable names. The second phase replaces all the global variables with distinct 2-character variable names.

The following example expands the @flash.message routine. It consists of 2 procedures, which the main body of the program references. testvar% is a global variable, and each of the 2 procedures contains 2 local variables:

```
    '***
    'Main Body of program
    '***
    =testvar%
    '*** Define and initialize Global Variables ***
@beginning
    testvar%=Ø
    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$,loop%
    '*** Define and initialize Local variables
    kbdscan$="" : cls
    for loop%=1 to 2Ø
        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 @(1Ø,1Ø),"Flashing-mssg-1, <enter> for 2, <x> to end"
    for loop%=1 to 5Ø
        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$,loop%
    '*** Define and initialize Local variables ***
    kbdscan$="" : cls
    for loop%=1 to 2Ø
        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 @(1Ø,1Ø),"Flashing-mssg-2, <enter> for 1, <x> to end"
    for loop%=1 to 5Ø
        kbdscan$=inkey$
        if kbdscan$=chr$(13) then goto @end.flash2
        if kbdscan$="X" or kbdscan$="x" then testvar%=1:goto
        next loop%
        goto @flash.message2
```

```
@end.flash2@end.flash2:return
        'The preceding line meets restrictions of the use of "RETURN"
        '***
        '
        'End of program
        '
        '***
@ending.mssg
        cls
        print @(1Ø,1Ø),"This program has been run in its entirety."
        end
```

The main body of the program defines testvar% as the only global variable and sets it equal to Ø. The main body of the program calls the first procedure, which causes Message 1 to flash on the screen. Message 1 continues to flash on the screen until you press **[ENTER]** or **[X]**. If you press **[ENTER]**, the procedure returns without changing testvar%. If you press **[X]**, the procedure sets testvar% to 1 and returns to the main body of the program.

When the main body regains control of the program, it tests testvar%. If testvar% equals 1, the program branches to the ending message routine and stops execution. If testvar% equals Ø, the program calls the second procedure.

The second procedure displays another flashing message and waits for you to press **[ENTER]** or **[X]**. On return to the main body of the program, the program tests testvar%. If it contains a 1, control branches to the ending message. If it contains any other value, control branches to the beginning of the main body of the program.

Remember that when TBA processes source code, it removes lines that begin with an apostrophe and does not insert a space between the line number and the BASIC statement. We have included these for readability in this sample of object code:

```
1 '***
2 'Main body of program
3 '
7 TE%=Ø
8 GOSUB 19
9 IF TE%=1 THEN GOTO 62
1Ø GOSUB 41
11 IF TE%=1 THEN GOTO 62 ELSE GOTO 7
12 '***
13 '
```

```
14 'Procedure #1
15 '
16 '***
19 KB$="" : CLS
2Ø FOR LO%=1 TO 2Ø
21 KB$=INKEY$
22 IF KB$=CHR$(13) THEN GOTO 33
23 IF KB$="X" OR KB$="x" THEN TE%=1:GOTO 33
24 NEXT LO%
25 PRINT @(1Ø,1Ø),"Flashing-mssg-1, <enter> for 2, <x> to
end"
26 FOR LO%=1 TO 5Ø
27 KB$=INKEY$
28 IF KB$=CHR$(13) THEN GOTO 33
29 IF KB$="X" OR KB$="x" THEN TE%=1:GOTO 33
3Ø NEXT LO%
31 GOTO 19
33 RETURN
34 '***
35 '
36 'Procedure #2
37 '
38 '***
41 KC$="" : CLS
42 FOR LP%=1 TO 2Ø
43 KC$=INKEY$
44 IF KC$=CHR$(13) THEN GOTO 54
45 IF KC$="X" OR KC$="x" THEN TE%=1:GOTO 54
46 NEXT LP%
47 PRINT @(1Ø,1Ø),"Flashing-mssg-2, <enter> for 1, <x> to
end"
48 FOR LP%=1 TO 5Ø
49 KC$=INKEY$
5Ø IF KC$=CHR$(13) THEN GOTO 54
51 IF KC$="X" OR KC$="x" THEN TE%=1:GOTO 54
52 NEXT LP%
53 GOTO 41
54 RETURN
56 '***
57 '
58 'End of program
59 '
6Ø '***
62 CLS
```

```
63 PRINT @(1Ø,1Ø),"This program has been run in its
entirety."
64 END
```

TBA translates labels and variables in the source code into line numbers and 2-character variables in the oject code. This chart shows the label translations:

Procedure label	Line Number in Object
@beginning	7
@flash.message1	19
@end.flash1	33
@flash.message2	41
@end.flash2	54
@ending.mssg	62

TBA assigns line numbers based on the location of the carriage return, (X'ØD') in the file. All characters from the beginning of the file until the carriage return comprise Line 1. All characters from that point until the next carriage return comprise Line 2.

A procedure is the code that lies between the label that initiates the procedure and the RETURN statement. Within a procedure, TBA translates all occurrences of a defined local variable into a unique variable name.

When TBA processes local variables in a given procedure, it starts transforming variables at the beginning of the procedure and finishes at the RETURN statement. For this reason, each procedure definition contains only 1 RETURN statement; it appears as the last line of the procedure; and it is the only BASIC statement on that line.

If you define the same local variable in more than 1 procedure, TBA translates the local variable into a different variable name in each procedure. Once you define a variable as local to a procedure, you cannot destroy its value even if you assign a new value to a different variable with the same name in a different procedure. TBA translates the local variables kbdscan$ and loop% in the @flash.message1 routine into KB$ and LO% in the object. @flash.message2 defines the same 2 local variables. In @flash.message2, the variable kbdscan$ translates into KC$ and loop% changes into LP%.

Using local variables eliminates the need to review already existing variables every time you introduce a new variable. Local variables help you debug programs because the procedure determines the value of the local variable. This prevents assigning a variable to some obscure number before you enter the procedure.

When you reenter a procedure, you can reinitialize the local variables in the procedure. Otherwise, the procedure maintains the same values you assigned earlier to the local variables.

To perform a return from somewhere within a procedure, use a GOTO to branch to a label at the end of the procedure. The RETURN statement follows this label. You can have the label and the RETURN statement on the same line, as in @end.flash1:RETURN. As the @flash.message examples illustrate, you can choose the label name @end.procedure to represent the branch to the RETURN statement.

After TBA processes all local variables, it transforms the global variables into 2-character variable names. In this example, testvar% is the only global variable defined. TBA globally replaces testvar% with the variable TE%, no matter where it appears in the source code--in the main body of the program or within the procedures.

To maintain a variable throughout the entire body of the main program, define it as a global variable. In the flashing message program, testvar% is a global variable that passes between the main program and the procedures.

You can define the same variable name as both global and local. It represents a different variable in the procedure than it represents in the main program. Consider this source code which uses the variable test% as a local and global variable:

```
' *** Define and initialize test% as a Global variable ***
=test%
     test%=Ø
     gosub @sub1
     print"this is the current value of test% -->";test%
     end
@sub1=test%
' *** Define a Local test%, distinct from the Global test% ***
     for test%=1 to 1Ø
         print test%
     next test%
```

```
     return
```

Processing this source code produces this object code:

```
     3 TF%=Ø
     4 GOSUB 9
     5 PRINT"this is the current value of test% -->";TF%
     6 END
     9 FOR TE%=1 TO 1Ø
     1Ø PRINT TE%
     11 NEXT TE%
     12 RETURN
```

In the procedure @sub1, TBA transforms the local variable test% into the variable TE%, and the global variable test% into TF%. Different variable names represent the local and global nature of the variable.

To have variables common to more than 1 procedure, define them as global. The following example illustrates 1 procedure (@sub1) referencing another procedure (@sub2):

```
@start
gosub @sub1
stop
'
@sub1=var1%,loop%
     for loop%=1 to 1Ø
          var1%=loop%
'
'call of @sub2 within @sub1
'
          gosub @sub2
          print var1%
     next loop%
     return
'
'entry to @sub2
'
@sub2=var1%
' *** Local var1% is distinct from @sub1's Local var1% ***
     var1%=var1%+(5*1ØØ)
     return
```

Both @sub1 and @sub2 define the variable var1% as local. Processing this source code produces this object code:

```
2 GOSUB 6
3 STOP
6 FOR LO%=1 TO 1Ø
7 VA%=LO%
11 GOSUB 2Ø
12 PRINT VA%
13 NEXT LO%
14 RETURN
2Ø VB%=VB%+(5*1ØØ)
21 RETURN
```

TBA translates the variable var1% in the @sub1 routine into VA%, and var1% in the @sub2 routine into VB% in the object code. This prevents the variable from passing between the 2 procedures. To remedy this situation, define var1% as a global variable, not as a local variable, in both procedures.

Use a similar procedure to define, dimension, and use arrays. To define an array, place the array name in a variable definition statement. Follow the array name with a type declaration tag, as though you were defining a simple variable. Place the array name alone in the definition statement. Using parentheses or a subscript value causes an error during processing. This example demonstrates how to define, dimension, and use the array totals#. All other variables in the definition statement represent simple global variables:

```
=totals#,loop1%,loop2%,dimrow%,dimcolumn%
     dimrow%=5Ø
     dimcolumn%=2Ø
     dim totals#(dimrow%,dimcolumn%)
     for loop1%= 1 to dimrow%
          for loop2%=1 to dimcolumn%
               totals#(loop1%,loop2%)=Ø
          next loop2%
     next loop1%
```

Processing this source code produces this object code:

```
2 DI%=5Ø
3 DJ%=2Ø
4 DIM TP#(DI%,DJ%)
5 FOR LO%= 1 TO DI%
6 FOR LP%=1 TO DJ%
7 TP#(LO%,LP%)=Ø
8 NEXT LP%
9 NEXT LO%
```

References to the array totals# in the source code translate to TP# in the object code. Since you can use defined variables with any BASIC function, you can use them as subscripts. In this example, the variables dimrow% and dimcolum% are subscripts.

Always define variables. Processing variable names without type declaration tags causes a syntax error. TBA does not process a variable that is not defined. It issues an "Undefined Variable" error message.

Miscellaneous Differences and Information

In TBA, you can use the REM statement to comment your source and object code. If a REM statement is the first text on a line, TBA passes the line unchanged to the object code. If REM appears later in a line, TBA does not recognize it properly and can generate an error. To pass a remark statement to the object code, use REM at the beginning of a line.

If a line begins with an apostrophe ('), TBA deletes the entire line. If there is an apostrophe in the middle of a line, TBA deletes the remaining text on that line. To use a remark in source code only, use an (').

To improve readability, use tabs throughout the source code. TBA removes leading tabs and extra spaces in the object code. This enables you to include spaces in the source code without using up extra memory in the object code.

In addition to this feature, the FC (Full Compression) parameter removes spaces within a BASIC line. Since FC removes spaces around reserved words, Model 4 BASIC cannot use this parameter. Model III BASIC and LBASIC can use the TBA object code that includes the FC parameter. To avoid forming the reserved word ASC and producing a

syntax error, FC does not delete the space between AS and a variable beginning with C.

When you define a label, you can type additional BASIC statements on the same line as the label definition. Do not define any local variables with the label. Do not embed a label definition starting a branch within a line. Use it as the first statement on the line. To include BASIC statements on a labeled line, follow the label with a colon (:). This example demonstrates writing a source line that contains BASIC statements following label:

```
@DELAY.LOOP : FOR J = 1 TO 2ØØØ : NEXT J
```

TBA does not process the variable J because it is too short and lacks a type declatation tag. It passes this statement into object code unmodified, without harming the execution or readability of the line. This is not recommended practice.

WRITING SOURCE CODE

You can create ASCII source code with a word processor such as SCRIPSIT; a text editor such as ALEDIT; or Interpreter BACIC. Assign the the source files a common file extension when you save them to disk. Since you use both source and object files, differentiate between the 2 with a common extension. For example, use /TBA for The BASIC Answer. Then, you can execute the object file only.

Using a Word Processor/Text Editor to Write Source Code

Whatever editor or processor you choose has advantages and disadvantages. The text must be pure ASCII.

Do not insert characters or lines that serve a specific function in the word processor or text editor into the source code. Page formatting lines, block markers or any other control characters cannot appear in the source code. You can use these characters when you write source code, to move blocks of code around, or make source listings more readable. If you use these special characters, delete them from the source code before you process it.

Unless you indicate otherwise, certain word processors save some characters in the text as non-ASCII characters. For example, SCRIPSIT saves a carriage return (X'ØD') as a non-ASCII character (X'8D'). Characters with a value less than X'2Ø' (except a tab, carriage return, or linefeed character) cannot appear in the text. For more information on saving a file in ASCII, consult the documentation for the word processor or text editor you use.

A carriage return character ends all lines in the program text and is the last character in the text. Some word processors have extraneous spaces at the end of the last carriage return. Prior to saving a program text file, perform a **delete to the end of text** after the last carriage return in the program.

Using the BASIC Interpreter to Write Source Code

There are disadvantages to writing source code with the BASIC Interpreter. You cannot transport or relocate it within the program and BASIC converts lower to upper case outside of quotation marks or remarks. If you do write source code with BASIC, you must adhere to several guidelines.

BASIC interprets any keyboard entries you make in answer to the READY prompt, and acts on them immediately unless you precede them with a line number. BASIC uses the line numbers to store the program text in memory. No line number references may appear in source code. Although you cannot use internal line numbers in the source code, you have to use line numbers to enter the source code in BASIC.

This is an example of source code written in BASIC:

```
1Ø @START.PROGRAM
2Ø ' *** Define Variables for TBA processing ***
3Ø =LOOP%,IN$
4Ø   '
5Ø   GOSUB @FLASH.MESSAGE
6Ø   '
7Ø @END.PROGRAM
8Ø        STOP
9Ø '
1ØØ '
11Ø @FLASH.MESSAGE
12Ø  CLS
13Ø  FOR LOOP% = 1 TO 2Ø
14Ø         IN$=INKEY$
```

```
15Ø         IF IN$=CHR$(13) THEN GOTO @END.FLASH.MSSG
16Ø  NEXT LOOP%
17Ø      PRINT@(1Ø,1Ø),"Flashing Message - Press <ENTER> to continue"
18Ø  FOR LOOP% = 1 TO 5Ø
19Ø       IN$=INKEY$
2ØØ       IF IN$=CHR$(13) THEN GOTO @END.FLASH.MSSG
21Ø  NEXT LOOP%
22Ø  GOTO @FLASH.MESSAGE
23Ø @END.FLASH.MSSG
24Ø  RETURN
```

When TBA processes the source code, it removes line numbers at the beginning of a line without affecting source code.

After you write the source code, include the A parameter in the SAVE command. If you omit the A parameter, BASIC saves keywords in compressed form and the file contains non-ASCII characters. This produces unpredictable results when you process the file. To save a file in ASCII, use the following syntax:

SAVE"filename",A

Source code lines may contain a maximum of 24Ø characters, including the line number. TBA truncates lines that exceed this limit.

Although source code does not reference line numbers, the sequence of the line numbers is important. As you write program lines, BASIC inserts them into the program in sequence by line number. Therefore, choose line numbers that result in the proper sequencing of lines, as you do when writing BASIC programs.

Using DIRECTIVES in Source Code

Using a directive, you can alter the output of an object file, and can conditionally process source files. The directives are:

*PRLINES	(use in source code only)
*LIST ON/OFF	(use in source code only)
*PAGE	(use in source code only)
*TITLE	(use in source code only)
*IF expression	(use in source code only)
*END	(use in source code only)

*expression	(use in source code or at directive prompt)

You can include all of the directives in source code. But the only directive you can include at the directives prompt is expression.

To specify a directive in the source code, the asterisk (*) must be the first character on the line. The directive is the only statement on the line.

***PRLINES [=n]**

Specifies the number of lines to print per page.

n specifies the number of lines to print per page and may be in the range 2Ø to 254. If you omit n, TBA assumes 56.

When the printer prints the specified number of lines, *PRLINES sends an X'ØC' character to the printer, which produces a top-of-form position.

Do initial top-of-form alignment before you process. To do this, make sure the printer is on line and the TRSDOS printer filter FORMS/FLT is in memory. Type a top-of-form command:**TOF [ENTER]** at TRSDOS Ready. You typically use PAGE=66,LINES=66 as forms control parameters. After the paper moves, position it so that the initial print occurs where you want it to. A top of form command positions the paper at the same relative line on subsequent pages.

If the printer prints 8Ø characters per line, set the CHARS parameter with FORMS/FLT at 8Ø to paginate correctly. You can set other print filter parameters, but any INDENT disrupts the formatted output.

To activate the printer filter, at TRSDOS Ready, type:

```
SET *FF TO FORMS/FLT [ENTER]
FORMS (PAGE=66,LINES=66) [ENTER]
```

Optionally, also type:

```
FORMS (CHARS=8Ø) [ENTER]
```

***LIST [ON/OFF]**

Toggles the listing of the object code.

If you omit ON and OFF, TBA assumes ON.

Assume that you want to list only the main body of the program and the ending message. The following *LIST directives added to the source code enable you to do so. For clarity, the directives are in upper case.

```
    '***
    'Main Body of program
    '***
    =testvar%
    '*** Define and initialize Global Variables ***
@beginning
    testvar%=Ø
    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
    '
    '***
*LIST OFF
@flash.message1=kbdscan$,loop%
    '*** Define and initialize Local variables
    kbdscan$="" : cls
    for loop%=1 to 2Ø
         kbdscan$=inkey$
         if kbdscan$=chr$(13) then goto @end.flash1
         if kbdscan$="X" then kbdscan$="x"
         if kbdscan$="x" then testvar%=1:goto @end.flash1
    next loop%
    print @(1Ø,1Ø),"Flashing-mssg-1, <enter> for 2, <x> to end"
    for loop%=1 to 5Ø
         kbdscan$=inkey$
         if kbdscan$=chr$(13) then goto @end.flash1
         if kbdscan$="X" then kbdscan$="x"
         if kbdscan$="x" then testvar%=1:goto @end.flash1
    next loop%
    goto @flash.message1
@end.flash1
```

```
    return
    '***
    '
    'Procedure #2
    '
    '***
@flash.message2=kbdscan$,loop%
    '*** Define and initialize Local variables ***
    kbdscan$="" : cls
    for loop%=1 to 2Ø
         kbdscan$=inkey$
         if kbdscan$=chr$(13) then goto @end.flash2
         if kbdscan$="X" then kbdscan$="x"
         if kbdscan$="x" then testvar%=1:goto @end.flash2
    next loop%
    print @(1Ø,1Ø),"Flashing-mssg-2, <enter> for 1, <x> to end"
    for loop%=1 to 5Ø
         kbdscan$=inkey$
         if kbdscan$=chr$(13) then goto @end.flash2
         if kbdscan$="X" then kbdscan$="x"
         if kbdscan$="x" then testvar%=1:goto @end.flash2
    next loop%
    goto @flash.message2
@end.flash2:return
    'The preceding line DOES meet restrictions on the use of "RETURN"
    '***
    '
    'End of program
    '
    '***
*LIST ON
@ending.mssg
    cls
    print @(1Ø,1Ø),"This program has been run in its entirety."
    end

TBA lists:

     '***
     ====>>

     'Main Body of program
     ====>>

     '***
```

```
====>>

=testvar%
====>>

'*** Define and initialise Global Variables ***
====>>

@beginning
====>>

testvar%=Ø
====>> 7 TE%=Ø

gosub @flash.message1
====>> 8GOSUB 2Ø

if testvar%=1 then goto @ending.mssg
====>> 9IF TE%=1 THEN GOTO 68

gosub @flash.message2
====>> 1ØGOSUB 44

if testvar%=1 then goto @ending.mssg else goto
@beginning
====>> 11IF TE%=1 THEN GOTO 68 ELSE GOTO 7

'***
====>>

'
====>>

'Procedure #1
====>>

'
====>>

'***
====>>

*LIST OFF
====>>
```

```
@ending.mssg
====>>

cls
====>> 68 CLS

print @(1Ø,1Ø),"This program has been run in its
entirety."
====>> 69PRINT @(1Ø,1Ø),"This program has been run in its
entirety."

end
====>> 7ØEND
```

The listing continues with the cross reference table. In this example, the *LIST directive allows you to control the exact parts of the program that list during processing. *LIST does not affect the output to the object file. TBA writes the entire program to disk. *LIST affects only listings directed to the printer or screen.

You do not have to specify *LIST ON at the beginning of the source file. The file generates a listing until it encounters the first *LIST OFF command. Also, the *LIST OFF directive appears in the listing, but the *LIST ON does not.

***PAGE**

Sends a top of form character (X'ØC') to the printer when listing an object file.

*PAGE directive does not affect listings sent to the screen. To direct the list output to the printer, include the LP parameter and activate the printer filter FORMS/FLT.

Using the flashing message program as the source code, the *PAGE directive lists the main body of the program and each individual procedure on a separate page:

```
    '***
    'Main Body of program
    '***
    =testvar%
    '*** Define and initialise Global Variables ***
@beginning
```

```
    testvar%=Ø
    gosub @flash.message1
         if testvar%=1 then goto @ending.mssg
    gosub @flash.message2
         if testvar%=1 then goto @ending.mssg else goto @beginning
*PAGE
    '***
    '
    'Procedure #1
    '
    '***
@flash.message1=kbdscan$,loop%
    '*** Define and initialise Local variables
    kbdscan$="" : cls
    for loop%=1 to 2Ø
         kbdscan$=inkey$
         if kbdscan$=chr$(13) then goto @end.flash1
         if kbdscan$="X" then kbdscan$="x"
         if kbdscan$="x" then testvar%=1:goto @end.flash1
    next loop%
    print @(1Ø,1Ø),"Flashing-mssg-1, <enter> for 2, <x> to end"
    for loop%=1 to 5Ø
         kbdscan$=inkey$
         if kbdscan$=chr$(13) then goto @end.flash1
         if kbdscan$="X" then kbdscan$="x"
         if kbdscan$="x" then testvar%=1:goto @end.flash1
    next loop%
    goto @flash.message1
@end.flash1
    return
*PAGE
    '***
    '
    'Procedure #2
    '
    '***
@flash.message2=kbdscan$,loop%
    '*** Define and initialize Local variables ***
    kbdscan$="" : cls
    for loop%=1 to 2Ø
         kbdscan$=inkey$
         if kbdscan$=chr$(13) then goto @end.flash2
         if kbdscan$="X" then kbdscan$="x"
         if kbdscan$="x" then testvar%=1:goto @end.flash2
    next loop%
```

```
    print @(1Ø,1Ø),"Flashing-mssg-2, <enter> for 1, <x> to end"
    for loop%=1 to 5Ø
         kbdscan$=inkey$
         if kbdscan$=chr$(13) then goto @end.flash2
         if kbdscan$="X" then kbdscan$="x"
         if kbdscan$="x" then testvar%=1:goto @end.flash2
    next loop%
    goto @flash.message2
@end.flash2:return
    'The preceding line DOES meet restrictions on the use of "RETURN"
*PAGE
    '***
    '
    'End of program
    '
    '***
@ending.mssg
    cls
    print @(1Ø,1Ø),"This program has been run in its entirety."
    end
```

The normal printed output is usually 1 continuous block of printed text. Because we included the *PAGE directive, it takes 4 physical pages to contain the printed listing of this file. Page 1 contains the main body of the program. Page 2 contains the first procedure: @flash.message1. Page 3 contains the second procedure: @flash.message2. Page 4 contains the ending message routine.

***TITLE "string"**

Prints a title at the top of each page.

string may contain 14 characters and you must precede it with a quote. The closing quote is optional. If you omit string, TBA displays an error message.

*TITLE prints the contents of string at the top of each page. It can appear anywhere in the source code. TBA titles all pages of the listing. For example, if you include *TITLE at the end of the source program, TBA prints the title at the top of the first page. You cannot turn *TITLE on and off.

The directive:

```
*TITLE "Flash Message"
```

prints:

```
BASIC Answer Flash Message June 24, 1982 12:Ø3 A.M.  Page 1
```

TBA prints 2 blank lines after the title. Use LP and *TITLE to document the history of a program's development.

***IF expression / *END**

*IF expression, *END, and *expression allow conditional processing of source code.

They allow you to produce multiple object files that use almost the same code.

This BASIC line using the conditional IF/THEN statement illustrates how the *IF / *END directives operate:

```
IF A% THEN PRINT "Condition true" : PRINT "A not equal zero"
```

This line tests the condition: Is the variable A% non-zero? If the condition is true, BASIC executes the 2 PRINT statements following the THEN. If the condition is false, it ignores the 2 PRINT statements following THEN.

With *IF / *END directives, you apply the same type of conditional testing to processing source code. You can use conditional blocks that begin with the *IF directive and end with the *END directive. All code between the *IF and *END directives belongs to that conditional block.

If the program encounters an *IF directive during processing, it procedes with a conditional test. If the condition is true, the program processes all code in that conditional block. If the condition is false, TBA ignores all code up to the *END directive.

An expression of up to 14 alphanumeric characters follows the *IF and tests the conditions. expression must begin with a letter. The remaining characters can include letters, numbers, the period (.), and underline (_) characters.

This example includes *IF / *END directives:

```
=total.items%,index.array%,item.drive$,item.file$
'
*IF hard.drive
total.items%=4ØØØ : item.drive$ = ":3"
*END
'
'
*IF floppy.drive
total.items%=5ØØ : item.drive$ = ":1"
*END
'
'
dim index.array%(total.items%)
item.file$ = "ITEMFILE/DAT" + item.drive$
```

This example defines the global variables total.items% and the array index.array%. Two conditional blocks follow the definition statement. The first conditional block tests whether the condition hard.drive is true. If it is true, TBA processes all code between the *IF and the *END, and stores the processed code in the object file. If the condition is false, TBA ignores all code between the *IF and the *END and does not include it in the object file.

The second conditional block tests whether the condition floppy.drive is true. If it is true, TBA processes the code between the *IF and *END and writes it to the object file. If it is false, TBA ignores the code between the *IF and *END and does not write it to the object file.

This source code can be a part of a larger program that manages data stored on a drive. If you store the data on a hard drive and you set the proper condition, the object code allows you to initialize the variable total.item% to 4ØØØ. The program dimensions the array index.array% and sets appropriate drive numbers for a particular file. If you store the data on a floppy disk, the object code allows you to initialize the variable to 5ØØ and the program dimensions the array accordingly. The program allows 4ØØØ data items if it runs on a hard drive and 5ØØ if it runs on a floppy disk.

Omitting expressions can cause unpredictable results. If you omit hard.drive and floppy.drive this program omits both conditional blocks. TBA processes the other statements into object code. Because

total.item% is not initialized by the conditional blocks, TBA dimensions index.array% contiaining only 1 element, index.array%(Ø). Attempting to access any other element results in a "Subscript out of range" error message.

Now that you understand how conditional blocks function, you need to know how to set a condition as either true or false. To establish conditional values, use the *expression directive.

*expression

The *expression directive establishes a condition of being true or false. You can include it in the source code or at the Directives? prompt. Use this directive with the *IF / *END directives to dictate the outcome of an *IF conditional block.

Use the same expression as the 1 specified in the *IF directive. If you include *expression, the corresponding *IF conditional is true, and the program processes all code in the conditional block. If you omit *expression, the corresponding *IF conditional is false, and the program omits the code in the conditional block.

In the previous example, change the second line from a remark to a directive, as follows:

```
=total.items%,index.array%,item.drive$,item.file$
*floppy.drive
*IF hard.drive
total.items%=4ØØØ : item.drive$ = ":3"
*END
'
'
*IF floppy.drive
total.items%=5ØØ : item.drive$ = ":1"
*END
'
'
dim index.array%(total.items%)
item.file$ = "ITEMFILE/DAT" + item.drive$
```

Since you include *floppy.drive, any time the conditional *IF floppy.drive occurs, it is true, and the program processes the code in the conditional block.

Since you omit *hard.drive, any time the conditional *IF hard.drive occurs, it is false, and the program does not process any of the code in the conditional block. If you process this source code, it produces the object code for the floppy drive version of the program.

In general, to perform conditional processing of the source code with directives, follow these steps:

1. Establish the conditional block within the program, by using the *IF and *END directives.

2. To process the code in the conditional block, specify the *expression in the source code. The expression is the same as the one you use in the *IF block.

3. To omit the conditional block, omit the *expression that you use in the *IF directive.

You can include *expression during processing. To do this, respond to the directives prompt by entering the *expression. This enables you to define the conditional processing you want at processing time, instead of embedding the conditions within the source code.

For example, to set the *hard.drive conditional as true in the previous example, answer the Directives ? prompt by typing:

Directives ? **hard.drive [ENTER]**

To enter more than 1 expression, separate them with commas. The program evaluates an *IF conditional it finds in the source code that contains the expression *hard.drive as true. After you enter the expression, TBA redisplays the prompt. Press **[ENTER]** or include more expressions. TBA assumes you want to include additional expressions until you press **[ENTER]** as the first key in respose to the directives prompt.

As with variables and labels, TBA ignores upper and lower case when differentiating expressions. An expression *HARD.DRIVE exactly matches *hard.drive because the only difference is the case of the letters. If you include the DC parameter, expression must match the corresponding expression in the *IF conditional exactly, including the case of the letters.

This example shows how to include more than 1 expression in a single line. The spaces following the commas are optional:

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

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

Both of these lines define the 4 conditionals as true.

When you specify *IF, *END, and *expression directives within the source code, place each on separate lines.

The *expression directive appears in the source code before the corresponding *IF conditional. If the *IF conditional physically precedes the *expression, the program evaluates it as false. Place all *expressions you use in the source code in the beginning of the source text. This ensures that you set all conditionals, and it groups the conditionals together at the beginning of the text. With this method, you do not have to search for any unwanted conditionals that you set when you processed the file in the past.

If the program encounters an *END directive without a corresponding *IF, it treats it as if there were no conditional.

If you specify an *IF without a corresponding *END, the program interprets all code from the *IF statement to the end of the source file as a part of the conditional, and processes it accordingly.

You can have as many *IF / *END conditional blocks within the source code as you want. You cannot nest conditional blocks.

If TBA encounters 2 *IF directives without an *END between them, the screen displays the following error message:

```
Nested IF Encountered
```

TBA stops processing this file and returns to TRSDOS Ready.

USING TBA

After you write source code, TBA processes it into object code to run as a BASIC program. No matter how you create source code, obtain a numbered source listing before you process it. To do this, at the TRSDOS Ready prompt type:

LIST filespec/ext:d (N,P)

to print a numbered listing. TBA references the numbers on the left of the listing as line numbers and as the reference point for processing errors.

If you intend to print output, establish top-of-form on the printer before proceding.

Processing Source Code

To return to the TRSDOS Ready prompt, press **[BREAK]** as a response to a prompt.

To use TBA, at the TRSDOS Ready prompt type:

TBA [ENTER]

TBA displays a paragraph of copyright information and the prompt:

Source Filespec ?

Enter the filespec you want TBA to process. If the filespec has an extension, include it. If you omit the extension, TBA assumes /TBA.

TBA displays the next prompt:

Object Filespec ?

Type any legal filespec or press **[ENTER]**. If you press **[ENTER]**, TBA assumes source filespec/BAS.

NOTE: At the same time that it processes the source file, TBA writes object code to the diskette under the filespec you specify as the object file. TBA writes to the object file no matter how you answer the prompt. Choose an object filespec to prevent TBA from overwriting anything you want preserved.

The next prompt is:

Processing Parms ?

Press **[ENTER]** to omit parameters or include 1 or more of the parameters. If you omit parameters, TBA displays processing information on the screen. If you include more than 1 parameter, separate them with commas.

The parameters are:

LP prints object code and the cross reference table on the printer.

TO displays only the object code.

NL indicates no listing. TBA does not print processing information on the screen or line printer.

NX omits the cross reference table.

FC eliminates as many spaces as possible from the object file.

DC causes all variables, labels and directives to appear in the object code exactly as they do in the source without conversion from lower case to upper case.

You can use only 1 output device to display the object code. If you include the LP parameter, TBA does not display processing information on the screen. If you include NL and TO parameters, TBA displays the cross-reference table and omits the source listing. If you use NL and NX, no video or printed output occurs.

NOTE: The FC parameter creates an object code that is incompatible with Model 4 BASIC.

You can include any combination of parameters. If you include more than 1, separate them by commas. These examples demonstrate the results of some combinations.

To send output from the object file to the printer, create a disk file which contains the object program, and generate a cross-reference table of variables and labels, answer the prompt by typing:

LP [ENTER]

TBA sends a listing to the printer, **not** to the screen, and generates a cross-reference table.

NL,NX [ENTER]

TBA omits the listings and writes the object to disk.

LP,NL,NX [ENTER]

TBA omits the listing even though you include the LP parameter because NL takes precedence.

[ENTER]

displays the object code and a cross-reference table on the screen.

The FC output option causes TBA to remove as many spaces as possible from the object code. If you omit the FC parameter, TBA leaves all single spaces from the source file intact. It reduces all groups of spaces to a single space. For example, the consider the source line:

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

If you omit FC, TBA produces:

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

TBA leaves the spaces between each keyword, but removes the extra spaces between THEN and GOSUB and GOSUB and the label. If you include

FC, the same source line becomes :

IFCR#<CU#THENGOSUB128

FC removes all spaces except those in REM lines, between quotation marks, and after the keyword AS if the following variable begins with C. Once FC removes the spaces, the object code no longer runs under Model 4 BASIC, which requires spaces around reserved words.
The DC option prevents TBA from converting lower case characters to upper case characters in variables, labels, and directives. This distinguishes a variable called LOOP1% from loop1% or LOop1%. The label @INPUT is different from @input, @Input, or @InpuT.

Omit the DC option unless you write the source code to accommodate it.

The last prompt is:

Directives ?

Press **[ENTER]** to omit expressions, or include 1 or more expressions that the program uses in an *IF conditional. If you include more than 1 expression, separate them by commas.

If the directives exceed the width of the input line, type as many as possible and press **[ENTER]**. TBA redisplays the prompt for you to include more expressions. This prompt repeats until you press the **[ENTER]** key as the first key of the line.

After you answer the prompts, TBA loads the source file into memory and displays the message Pass 1. TBA processes the source file.

During the processing, TBA checks to see that you write valid source code, which is is not necessarily working BASIC code. If TBA does not detect an error, the processing phase continues and creates object code. As each pass begins, TBA displays Pass X on the screen.

If TBA detects a hardware error during processing, (for example, Disk I/O Error, Parity Error), it returns to the TRSDOS Ready prompt.

If TBA detects an error in the source code, it suspends processing and displays the appropriate error message. To end processing and return to the TRSDOS Ready prompt, press **[BREAK]**. To see if there are additional source code errors, press **[SPACEBAR]**.

After TBA processes the entire source code, it returns to the TRSDOS Ready prompt.

Error Messages

When TBA encounters errors in the source code, it displays these error messages:

Illegal Procedure Label

A label name does not conform to valid label names.

Multiply Defined Label

You defined the same label more than once in the source code.

Illegal Variable

A variable does not conform to the variable name rules.

Variable Definition Format Error

You omitted the = or commas in a defintion statement.

Local Procedure Used without a RETURN

You omitted a RETURN statement as the last statement of a procedure.

Undefined Procedure Label

A GOTO, GOSUB or RESUME references an undefined label.

Multiply Defined Global Variable

You defined a variable as global more than once.

Undefined Variable

TBA encountered an undefined variable.

Illegal Title Format

A *TITLE directive does not conform to the rules for a correct title.

Illegal Directive Format

You used a directive expression that does not conform to the rules for directives.

If TBA encounters these errors, it returns to TRSDOS Ready:

Insufficient Memory to Load Text

There is not enough memory available to process the source file.

Symbol Table Overflow

You used too many variables or too many references to those variables.

Source Line too Long

A line in the file exceeds the 24Ø character limit.

Variable usage Overflow

You used more than 93Ø variables of a certain type declaration tag.

These errors occur during the input prompts. TBA repeats the prompt.

Illegal Filespec

Either the source or object filespec is not a proper file specification.

Bad *expression format

You responded to the directives prompt with an illegal input format, which cancelled the current input line.

Bad Parameter(s)

You answered the Processing Options prompt incorrectly.

Identical Source and Object Filespecs

The object file has the same name as the source file.

If possible, these error messages display the number of the line in which the error exists.

Sample Screen and Video Output

Assume the source file TBA is processing is:

```
     '***
     'Main Body of program
     '***
     =testvar%
     '*** Define and initialize Global Variables ***
@beginning
     testvar%=Ø
     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$,loop%
     '*** Define and initialize Local variables
     kbdscan$="" : cls
     for loop%=1 to 2Ø
          kbdscan$=inkey$
          if kbdscan$=chr$(13) then goto @end.flash1
          if kbdscan$="X" then kbdscan$="x"
          if kbdscan$="x" then testvar%=1:goto @end.flash1
     next loop%
     print @(1Ø,1Ø),"Flashing-mssg-1, <enter> for 2, <x> to end"
     for loop%=1 to 5Ø
          kbdscan$=inkey$
          if kbdscan$=chr$(13) then goto @end.flash1
          if kbdscan$="X" then kbdscan$="x"
          if kbdscan$="x" then testvar%=1:goto @end.flash1
     next loop%
     goto @flash.message1
@end.flash1
     return
     '***
     '
     'Procedure #2
     '
     '***
@flash.message2=kbdscan$,loop%
     '*** Define and initialize Local variables ***
     kbdscan$="" : cls
     for loop%=1 to 2Ø
          kbdscan$=inkey$
          if kbdscan$=chr$(13) then goto @end.flash2
          if kbdscan$="X" then kbdscan$="x"
          if kbdscan$="x" then testvar%=1:goto @end.flash2
     next loop%
     print @(1Ø,1Ø),"Flashing-mssg-2, <enter> for 1, <x> to end"
     for loop%=1 to 5Ø
          kbdscan$=inkey$
          if kbdscan$=chr$(13) then goto @end.flash2
          if kbdscan$="X" then kbdscan$="x"
          if kbdscan$="x" then testvar%=1:goto @end.flash2
     next loop%
     goto @flash.message2
```

```
@end.flash2:return
     'The preceding line DOES meet restrictions on the use of "RETURN"
     '***
     '
     'End of program
     '
     '***
@ending.mssg
     cls
     print @(1Ø,1Ø),"This program has been run in its entirety."
     end
```

If you include the LP parameter, TBA generates the following output to the printer:

```
'***
====>>

'Main Body of program
====>>

'***
====>>

=testvar%
====>>

'*** Define and initialize Global Variables ***
====>>

@beginning
====>>

testvar%=Ø
====>> 7 TE%=Ø

gosub @flash.message1
====>> 8GOSUB 19

if testvar%=1 then goto @ending.mssg
====>> 9IF TE%=1 THEN GOTO 66

gosub @flash.message2
====>> 1ØGOSUB 43
```

```
if testvar%=1 then goto @ending.mssg else goto @beginning
====>> 11IF TE%=1 THEN GOTO 66 ELSE GOTO 7

'***
====>>

'
====>>

'Procedure #1
====>>

'
====>>

'***
====>>

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

'*** Define and initialize Local variables
====>>

kbdscan$="" : cls
====>> 19 KB$="" : CLS

for loop%=1 to 2Ø
====>> 2ØFOR LO%=1 TO 2Ø

kbdscan$=inkey$
====>> 21KB$=INKEY$

if kbdscan$=chr$(13) then goto @end.flash1
====>> 22IF KB$=CHR$(13) THEN GOTO 35

if kbdscan$="X" then kbdscan$="x"
====>> 23IF KB$="X" THEN KB$="x"

if kbdscan$="x" then testvar%=1:goto @end.flash1
====>> 24IF KB$="x" THEN TE%=1:GOTO 35

next loop%
====>> 25NEXT LO%
```

```
print @(1Ø,1Ø),"Flashing-mssg-1, <enter> for 2, <x> to end"
====>> 26PRINT @(1Ø,1Ø),"Flashing-mssg-1, <enter> for 2, <x> to end"

for loop%=1 to 5Ø
====>> 27FOR LO%=1 TO 5Ø

kbdscan$=inkey$
====>> 28KB$=INKEY$

if kbdscan$=chr$(13) then goto @end.flash1
====>> 29IF KB$=CHR$(13) THEN GOTO 35

if kbdscan$="X" then kbdscan$="x"
====>> 3ØIF KB$="X" THEN KB$="x"

if kbdscan$="x" then testvar%=1:goto @end.flash1
====>> 31IF KB$="x" THEN TE%=1:GOTO 35

next loop%
====>> 32NEXT LO%

goto @flash.message1
====>> 33GOTO 19

@end.flash1
====>>

return
====>> 35 RETURN

'***
====>>

'
====>>

'Procedure #2
====>>

'
====>>

'***
====>>
```

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

'*** Define and initialize Local variables ***
====>>

kbdscan$="" : cls
====>> 43 KC$="" : CLS

for loop%=1 to 2Ø
====>> 44FOR LP%=1 TO 2Ø

kbdscan$=inkey$
====>> 45KC$=INKEY$

if kbdscan$=chr$(13) then goto @end.flash2
====>> 46IF KC$=CHR$(13) THEN GOTO 58

if kbdscan$="X" then kbdscan$="x"
====>> 47IF KC$="X" THEN KC$="x"

if kbdscan$="x" then testvar%=1:goto @end.flash2
====>> 48IF KC$="x" THEN TE%=1:GOTO 58

next loop%
====>> 49NEXT LP%
print @(1Ø,1Ø),"Flashing-mssg-2, <enter> for 1, <x> to end"
====>> 5ØPRINT @(1Ø,1Ø),"Flashing-mssg-2, <enter> for 1, <x> to end"

for loop%=1 to 5Ø
====>> 51FOR LP%=1 TO 5Ø

kbdscan$=inkey$
====>> 52KC$=INKEY$

if kbdscan$=chr$(13) then goto @end.flash2
====>> 53IF KC$=CHR$(13) THEN GOTO 58

if kbdscan$="X" then kbdscan$="x"
====>> 54IF KC$="X" THEN KC$="x"

if kbdscan$="x" then testvar%=1:goto @end.flash2
====>> 55IF KC$="x" THEN TE%=1:GOTO 58
```

```
next loop%
====>> 56NEXT LP%

goto @flash.message2
====>> 57GOTO 43

@end.flash2:return
====>> 58 RETURN

'The preceding line DOES meet restrictions on the use of "RETURN"
====>>

'***
====>>

'
====>>

'End of program
====>>

'
====>>

'***
====>>

@ending.mssg
====>>

cls
====>> 66 CLS
print @(1Ø,1Ø),"This program has been run in its entirety."
====>> 67PRINT @(1Ø,1Ø),"This program has been run in its entirety."

end
====>> 68END
```

```
Procedure Label    Defn #  Line #  Referenced at Line #'s
---------------    ------  ------  ----------------------

@BEGINNING           6       7      11
@FLASH.MESSAGE1     17      19      8,33
@END.FLASH1         34      35      22,24,29,31
@FLASH.MESSAGE2     41      43      1Ø,57
@END.FLASH2         58      58      46,48,53,55
@ENDING.MSSG        65      66      9,11

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

KBDSCAN$ *           17     KB$     19,21,22,23,23,24,28,29,3Ø,3Ø,31
KBDSCAN$ *           41     KC$     43,45,46,47,47,48,52,53,54,54,55

LOOP% *              17     LO%     2Ø,25,27,32
LOOP% *              41     LP%     44,49,51,56
TESTVAR%             4      TE%     7,9,11,24,31,48,55
```

TBA scrolls screen listings. To pause the listing, press **[SHIFT][@]**. To resume the scroll process of the listing, press **[SPACEBAR]**.

TBA displays each source line, followed by ====>>, and the object line.

Although the listing displays the first 6 source lines, they produce no object code. TBA ignores lines starting with ('), and does not translate label and variable definition statements into executable object code. The translation process continues at the line following the remarks and label and variable definition statements, in this case, testvar%=Ø. It translates this line into: 7TE%=Ø

TBA numbers lines in the object code consecutively, starting with Line 1. It also numbers lines which do not exist in the object code. It deletes these numbers after processing.

Since there are 68 lines of source code, the object file consists of Lines 1-68. A cross-reference listing follows the last line of the source/object code translation. The first part of the table identifies the translation performed on labels. The left column of the table contains the label names. Column 2 contains their definition line numbers in the source code.

Column 3 identifies the line number in the object code that corresponds to the definition line number in the source code. For example, the processor translates the label @beginning (Line 6 in the source program) into Line 7 in the object program--the first executable statement following the label definition.

The last column lists all line numbers in the object code which reference the given label. In the listing of the processed code, the source line referencing the label @beginning corresponds to object Line 11. Object Line 11 containing the statement GOTO 7 represents the label @beginning.

The second part of the table, the variable cross-reference table (XLATE), shows the variable translations that occur. This table lists global and local variables defined in the source code. Column 1 in the variable translation table lists all variables as you define and use them in the source code. An asterisk (*) follows the names of local variables. The variables kbdscan$ and loop% appear twice in the table because they represent local variables used in 2 separate procedures which translate into 2 distinct variables. Column 2 identifies the source lines that define the variables.

Column 3 gives the object code variable name of the same variable name in the source code. For example, the global variable textvar% translates into TE%, the variable kbdscan$ translates into KB$ in the first procedure, and into KC$ in the second procedure.

The last column gives the line numbers in object code that reference these variables. For example, the variable TE% appears in Lines 7,9,11,24,31,48, and 55 of the object code.

TBA groups variables in a cross-reference table in a special order. It lists string variables first, followed by integer, single-precision and double-precision variables. Within these subgroups, global variables appear first within a subgroup, listed in alphabetical order according to the variable name used in the source code. Next, the local variables appear in alphabetical order. This type of grouping makes it easy to select specific variables in the table.

HOW TBA OPERATES

TBA processes source code into object code in a series of steps. In each of these steps, TBA performs a pass on the source code, which means that TBA processes the current interim code. Each particular pass alters the code, changing it from the source code into executable object code, and places the interim results in the object filespec. TBA writes an interim object code to the disk containing the specified object filespec, and uses this code during subsequent passes.

TBA performs 6 passes on the source code. Each pass performs a specific processing function that changes the source code into object code. During these passes, it creates and maintains tables which store information about labels, directives and variables. The following briefly describes the actions that occur during each pass.

Pass 1

TBA writes text information from the source file to the object file. It removes line numbers at the beginning of the line, assigns line numbers to all text lines, starting with line number 1, incrementing by 1. As TBA encounters labels, the processor checks to see if the label has been defined. If it detects a multiply defined label, the screen displays the proper error message.

Since a label definition in source code produces no object code, TBA uses the next source text line as the object line number associated with the definition of that label.

Pass 2

In Pass 2, TBA evaluates variables. It checks the list of labels that define procedures and changes local variables to 2-character variables. TBA maintains a table of 2-character variable names already in use.

TBA handles each procedure individually, starting at the procedure definition line and ending with the associated RETURN. You can use the same local variable in 2 different procedures because the processor translates them into 2 different variables.

It examines the first 2 characters in the local variable and checks to see if that variable name is in use. For example, if TBA encounters the local variable test.variable%, the processor attempts to change this variable to the 2-character variable TE%. If TE% already exists, it uses TF% if that is not in use.

Passes 3 & 4

In Passes 3 and 4, TBA translates global variables. TBA examines the entire source code and translates any match of a defined global variable into a 2-character variable name. Since TBA translates local variables first, you can use the same variable name to represent a local and global variable. For example, if you define the variable test.variable% as global and local, TBA translates it into TE% in the local procedure and TF% everywhere else in the program.

To use a global variable within a procedure, do **not** define it as local. TBA does not translate the global variable until after it processes the local variables. The global variable receives the same variable name throughout the program.

Pass 5

In Pass 5, TBA changes all label references to reference the line number associated with the label. It goes through the table containing the label definitions and associated line numbers, and translates all label references in the source code into the corresponding line number. TBA removes extraneous spaces from text lines.

Pass 6

In the final pass, TBA compresses the source code and performs output options. It either lists to the printer or video, and creates an object file on disk. If you specify FC, TBA compresses all spaces except those found within quotation marks and REM statements. To produce pure ASCII, TBA examines the high bit in each character. If it is set, TBA resets it.

GENERAL GUIDELINES AND PROGRAM MAINTENANCE

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, you need to carefully construct the error handling routines.

All ON ERROR GOTO statements reference a label, for example, ON ERROR GOTO @error.routine. The label defines the entry point into the error trapping routine. To return to a specific line from an error trapping routine, use the RESUME command, followed by the label which represents the point of return. This is an example of how to establish error trapping routines:

```
'branch to @error.detected if an error is encountered.
'
ON ERROR GOTO @error.detected
'
@error.done
'
'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
```

Using a label to define an error trapping routine functions the same as referencing a line in the program with GOTO. Where you place the label following the RESUME depends on your program's needs. This example assumes that you need to repeat the program code if you encounter a recoverable error. Since you terminate error trapping routines with a RESUME instead of a RETURN, using local variables in error trapping routines produces an error during processing. Remember, in TBA, RETURN ends a procedure and the definition of variables used in the procedure.

You can use the statement ON ERROR GOTO Ø in the source to turn off any active ON ERROR GOTO statement.

Enhancing Program Operation and Speed

When TBA creates object files, it writes them to disk in pure ASCII, ommitting compression codes to represent BASIC keywords. To load an object file faster, LOAD and SAVE the program to disk, thereby using compression codes for all BASIC keywords.

Although you define a variable before you use it, defining a local or global variable in the source code does not initialize it in the object code. BASIC allows you to initialize variables as you need them. Initialization slows the operation of the program because you must establish the variable in BASIC's variable table before you use it. To make global variables readily accessible in the variable table, initialize them before you need them.

Because local variables are only valid in the procedure in which you define them, initilialize only global variables in this manner.

You can initialize variables to contain a specific value as you enter a program. This example initialize all variables to Ø:

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

To prevent delays when TBA first encounters a global variable, initialize them at the beginning of a program. Doing this produces a noticeable delay when you execute the variable initialization statements. However, it is better to encounter such a delay before the program execution begins rather than during execution.

BASIC establishes a variable table when it first encounters a variable in a program. Every time BASIC accesses a variable, it scans the variable table. BASIC scans and locates variables placed at the beginning of the initialization statement quickly. Place frequently used global variables early in the variable initialization statement.

BASIC accesses integer (%) variables, stored in 2 bytes, faster than single- or double-precision variables, stored in 4 and 8 bytes respectively. It accesses single-precision variables faster than double-precision. When declaring your variable names, use the most efficient type for the operation. If possible, use integer type variables in a FOR / NEXT loop.

Initialize all variables representing string constants at the start of the program. Do not store any other string information, such as input from the keyboard or a disk file, in these variables. The variable's location in the program text in RAM references the string.

Well designed source code consists of a program's main body and references to procedures that perform various tasks. Although programming requirements decide the tasks that these procedures perform, how you write the program determines the speed of its operation.

Every time you execute a backward branch, BASIC scans the program text from the beginning line to the branching line. Position frequently referenced routines at the beginning of the program text. To do this, perform a branch to the main body of the program, thereby bypassing the procedures located at the beginning of the program text. Even initializing many global variables leaves substantial program code in front of these procedures. Since initializing global variables is a 1 time operation, do it at the end of the program text. This program layout represents how such a structure can increase the speed of executing a program:

1. Global variable definition statements
   (do not appear in object 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 which are accessed infrequently
   (such as error trapping)
6. Global variable initialization statements
7. Branch statement to the Main body of the program
   (Step 4)

Use of CHAIN MERGE, and COMMON

Although TBA allows a certain amount of code compression, sometimes a program's amount of code or data storage requirements force you to break it up into several elements, or into a suite of smaller programs running together. With its CHAIN MERGE and COMMON options, Model 4 BASIC allows a sophisticated method of inter-program communication.

You can run each program in the suite without using data or other material from another program only if the programs refer to one another by RUN filename. You write and process the individual programs without regard to other programs, and without taking any special action.

If 1 or more programs in the suite use CHAIN filespec to refer to another program in the suite, use the ALL parameter or COMMON statement to pass data between the calling and the called programs. You need to take special action to ensure the variables maintain their common names during TBA processing, which changes matching variable names. For instance, if 2 programs contain only 2 globals, say LOOP1%, LOOP2%, and no local variables already declared can translate into LO% or LP% (LOCAL.INTEGER% translates into LO%, of course), then TBA processes both programs with the 2 globals translated into LO% and LP%. When you consider chaining, prepare contingencies for variable name changes in advance.

To use CHAIN with COMMON, alphabetically separate variables you are passing between programs from all other variables. Using a prefix such as CO. reminds you that the variable CO.LOOP1% is common to both programs and separates it from all other variables. Of course, other

variables can have their own prefixes, but not with the initial letter C. If the programs contain many variables, choosing a prefix becomes more difficult. It is preferable, to give common variables initial letters. For example, AA. through CZ., and other variables the initial letters EA. through ZZ.

You want the TBA definition statements of the common variables to appear identical in both the called and the calling programs. To copy these statements in both programs, set aside a separate line or group of lines for this purpose.

For clarity, place the COMMON statement's variable list in the same order and the same group of lines as, but in a separate file from, the definition statements.

To use CHAIN with the ALL parameter, define all variables as globals, and implement identical definition statements in the called and calling programs. This prevents you from using local variables in defined procedures, and from reusing a series of small modules to create larger programs. Instead, organize the programs to use the COMMON statement to eliminate variables not really required in called programs, and to exclude them from the COMMON block.

CHAIN MERGE requires you to implement one of these approaches and handle line numbers in the following manner.

When you use CHAIN with the line number options, you control line numbers because you create the calling and the called programs with the BASIC editor. TBA invalidates this. The BASIC line:

```
1Ø3Ø CHAIN MERGE "OVRLAY2", 1ØØ, , DELETE 12Ø-14Ø
```

fails to produce the desired results. With certain consistent source program structures, TBA successfully processes code containing similar CHAIN MERGE syntax. Use this structure when creating files that contain COMMON or CHAIN MERGE:

1. Global variable definition statements, for COMMON
   (do not appear in object code)
   COMMON statement (all COMMON global variables).
2. @ - label
   Global variable definition statements, for variables used before first overlay call, but not needed thereafter.

Global variable initialization statements.
Portion of main progam that you execute only on entry to program.
Branch statement to the Main body of the program (Step 4)
REM statements acting as place holders for OVERLAY section which is here.

3. Resident Procedures accessed frequently
4. Resident Main Body of the program
   When you need overlays, CHAIN them in to overwrite
   @ - label (Step 2)
5. Resident Procedures accessed infrequently.
   (such as error trapping)

NOTE: The similar outline for Enhancing Operation and Speed includes Steps 6 and 7. In this outline, they are included as the first 2 elements of Step 2.

The overlay area in this outline is Step 2. It initially contains code that you use only on entering the program, and that you can overwrite when you call the first overlay. To make this outline work, create a variable name scheme that allows all programs in the suite to share an identical global variable list and COMMON statement (Step 1). Estimate the number of lines of code required for the largest overlay section. This way, TBA reserves the number of lines that the overlays occupy between Steps 2 and 3.

To accomplish a CHAIN MERGE, follow this process or see Program 4. The numbers refer to the step numbers in the outline:

PREPARATION

Identify routines you want as overlays.

Estimate length of overlay in lines, not amount of memory.

Identify variables used throughout the program.

Group these into common globals, globals used only on entry, local variables for resident procedures, and local variables used in overlays.

Decide on variable name scheme, which prevents TBA from giving the common variables different names in different programs. For example, use the prefixes of CO. GL. LO. OV.

Ensure that existing procedures fit into the variable name scheme.

Create a common global variable definition list, and COMMON statement as a file to reuse.

RESIDENT PROGRAM

Begin source for resident program with the block that contains global variable definitions and COMMON statement. (Step #1).

First line of Step #2 is a label followed by the code you execute on entry to the program, for example, @START.OVRLAY. The active code finishes with a branch to the code contained in Step #4, for example, GOTO @MAIN. Conclude Step #2 with the number of REM statements that make the total number of lines in Step #2 equal to the number of lines estimated for the largest overlay.
Step #4 contains the main body of the program. When this code requires any of the overlays, it branches forward to a group of statements at the conclusion of Step #4, which handle the entry to, and exit from, the overlays. For example:

```
' END of MAINLINE section. START of OVRLAY call section.
' Execute OVRLAYs by setting direction switch CO.WHICH.OV%
' to 1 or 2 then GOTO @CALL.OVRLAY from MAINLINE
' on return from OVRLAYs, GO (back) TO CALLERx
' The merged program will always restart @EXEC.OVRLAY
' which will GOSUB @START (1st line in the OVRLAY)
' MAINLINE needs COMMON block (globals) to talk to OVRLAYs
' OVRLAYs forced to equal lengths so no need to DELETE lines.
'
@CALL.OVRLAY
  ON CO.WHICH.OV% GOTO @OVRLAY1.CALL, @OVRLAY2.CALL
  GOTO @NO.OVRLAY
@RETURN.OVRLAY
  ON CO.WHICH.OV% GOTO @CALLER1, @CALLER2
@OVRLAY1.CALL
  IF CO.RESIDENT% = 1 THEN GOTO @EXEC.OVRLAY
  CO.RESIDENT% = 1
  CHAIN MERGE "OVRLAY1", @EXEC.OVRLAY
```

```
@OVRLAY2.CALL
  IF CO.RESIDENT% = 2 THEN GOTO @EXEC.OVRLAY
  CO.RESIDENT% = 2
  CHAIN MERGE "OVRLAY2", @EXEC.OVRLAY
@EXEC.OVRLAY
       GOSUB @START.OVRLAY
       GOTO  @RETURN.OVRLAY
'
' END of OVRLAY call section
```

If a second group of programs uses overlays, make a small, reusable file for this group of lines.

Copy seldom used procedure (Step #5) into the source, or code and save it to reuse later.

Collect frequently used procedures (Step #3) into a file to copy into the resident source. The resident source is now complete, and saved for TBA processing. Process it to create the cross-reference table.

OVERLAY SECTIONS

Numbered references here refer back to 5 numbered sections in the resident program above.

For each overlay:

Start the overlay with the common block File 1, created for the resident program. Change the contents of remark statements and directives but do not insert or delete any lines.

First line of Step #2 is a label (for example, @START.OVRLAY) followed by the actual code of the overlay.

Make the overlay as self-contained as possible, and code it to allow 1 entry point (@START.OVRLAY) and 1 exit (RETURN) back to the end of the main body of the program Step #4.

If the overlay requires its own local procedures, prefix the variable names to prevent TBA from giving the common globals different names in the overlay than they have in the resident program.

If you encounter error conditions in the overlay, execute the ON ERROR GOTO in the main body of the program (Step #4). Make no direct references to the infrequently used subroutines in Step #5.

If the overlay requires the use of any of the frequently used subroutines in Step #3, then pad the Step #2 section with REM statements up to the number of lines. You can merge the file which contains all the frequently used procedures Step #3 into this overlay. Then, TBA can translate the procedure labels to line numbers accurately. Both the resident and the overlay programs contain duplicate copies of the Step #3 section.

The overlay source file is now ready for TBA processing.

Process the file, and double check the cross-reference table against the resident program's cross-reference to ensure an exact match between appropriate variable names, and between procedure labels that translate to the same line number. For example, @START.OVRLAY, and Step #3 section if required.

When TBA processes a source file, it produces object code in sequential lines ordered from 1 to the maximum number of lines in the source. Some lines such as labels, definitions, and remarks starting with ('), do not translate into an object code line, making it unlikely for an incoming overlay to completely overwrite the previous overlay.

To make each overlay completely overwrite the previous one, CHAIN MERGE the incoming overlay into memory. To do this, create a file in BASIC with sequential line numbers, from 1 to the requisite length. For example, the length of the common block Step #1, plus the length of largest overlay Step #2. Each line contains only REM. Load this file into BASIC. Then, for each overlay, MERGE the object file from TBA on top of the newly created file. This eliminates line number gaps in the resulting merged file.

You do not need the DELETE parameter because every line that the CHAIN MERGE statement deletes is either in the previous overlay, or in the Step #2 section of the resident program.

The resident program, and the overlays it requires, is now ready to test. If it requires modifications now or at a later date, you must take care to ensure that changes made to one program segment do not affect the way other segments run. If they do, reprocess each program that contains the changed segment.

The outline used here is different from the outline in "Enhancing Operation and Speed." The length of the overlay section (Step #2) moves the position of frequently used subroutines (Step #3) further down in memory. If the overlay portion is very large, or there are numerous calls to Step #3, the program slows down. To increase speed, slightly modify the outline and procedures above.

Resident program (Step #2) now contains only a GOTO instruction to a new step (#3A) created between Steps 3 and 4. The previous contents of Step 2 become the overlay section in #3A.

Overlay programs require similar treatment. The contents of Step 2 move into Step 3A. Step 3 changes from optional to required. Step 2 contains no active code. Instead, it has the same number of REM statements as the number of lines in Step 2 of the resident program. Thus, Step 3 in both resident and overlay programs starts at the same line number after processing, and the program needs no other modifications.

Note: These suggestions can help you develop consistent program structures. Use the method that makes your code most transportable to future programs you write, and do not stint on documentation which clarifies the source code for those who maintain the programs in the future.

Maintaining Programs

When maintaining programs created by TBA, change the program in the source code, **not** the object code. Because you write the source code in descriptive dialogue, it is easier to follow than the object code, where labels change to line numbers, variable names use only 2 characters, and extraneous spaces generally do not exist.

More important, by restricting changes to the source code, then processing it into object code, you ensure that there is one current version of a program. If you correct a slight error in the program by modifying the object code, then decide to add an additional feature to the program by editing the source code, the source code does not reflect the change made in the object code.

Example Programs

This section contains source BASIC programs and the object code. To learn how the processor functions and what results you obtain through the processing operation, experiment with these programs before you write your own source code.

You can enter these programs into almost any editor, but when you first start to work with TBA, it is easier to use an editor with which you are familiar.

Type this program into an editor or BASIC and save it as FACTOR/TBA:

```
*TITLE"EXERCISE 1"
'global definition statements are below
=FACTOR$,LOOP1!,LOOP2!,HALF!,HORIZ.LINE$,START!,END!,IN.PUT$
=PRIME.FLAG%,PRIME.ARRAY!,PRIME.FACTOR$,COUNTER%,START2!
=LOOP.COUNT%,PRIMES.FOUND%
     'dimension and clear statements;
     'both array variables are defined above.
DIM PRIME.ARRAY!(1Ø),PRIME.FACTOR$(1Ø)
     'program execution start in case subroutines are placed
     'there at a later date.
'
@START
'
CLS
INPUT "ORIGIN OF SCAN"; IN.PUT$
     'all IF statements are indented to set them apart
     IF VAL(IN.PUT$)<2 THEN @START
START!=INT(VAL(IN.PUT$))
INPUT "   END OF SCAN"; IN.PUT$
END!=(VAL(IN.PUT$))
     IF END!<START! THEN SWAP START!,END!
HORIZ.LINE$=STRING$(79,61)
CLS
'
```

```
@START.LOOP1
'
FOR LOOP1! = START! TO END!
     'statements contained within a FOR/NEXT loop are tabbed
     'over for clarity
     HALF!=LOOP1!
     FACTOR$=""
     PRINT @ Ø, "factoring "USING"###,###";LOOP1!;
     PRINT @ 4Ø,"primes found on this scan";
     PRINT USING"##,###";PRIMES.FOUND%;
     PRINT @ 8Ø,"prime factors : ";CHR$(3Ø);
     PRINT HORIZ.LINE$;: START2!=2
     '
     @START.1.LOOP2
     '
     FOR LOOP2! = 2 TO HALF!
       IF HALF!/LOOP2!=INT(HALF!/LOOP2!) THEN GOSUB @GOT.ONE
     NEXT LOOP2!
     '
     @END.1.LOOP2
     '
     IF VAL(FACTOR$) = LOOP1! THEN GOSUB @PRIME ELSE GOSUB @NOT.PRIME
     PRIME.ARRAY!(COUNTER%)=LOOP1!
     PRIME.FACTOR$(COUNTER%)=FACTOR$
     LOOP.COUNT%=COUNTER%
     '
     @START.2.LOOP2
     '
     FOR LOOP2!=Ø TO 1Ø
       PRINT @(LOOP2!+4,Ø),"";
       PRINT PRIME.ARRAY!(LOOP.COUNT%), PRIME.FACTOR$(LOOP.COUNT%);
       PRINT CHR$(3Ø);
       LOOP.COUNT%=LOOP.COUNT%-1
         IF LOOP.COUNT%=-1 THEN LOOP.COUNT%=1Ø
     NEXT LOOP2!
     '
     @END.2.LOOP2
     '
     COUNTER%=COUNTER%+1
       IF COUNTER%=11 THEN  COUNTER% =Ø
NEXT LOOP1!
@END.LOOP1
END
'
```

If you direct the processed output to a printer, prepare the printer for output and ensure proper pagination by sending a top-of-form to the printer. To do so, enter the TOF command at the TRSDOS Ready prompt. If the command does not work, set the *FF device to FORMS/FLT.

TOF causes the printer to advance the paper to its top-of-form position. Position the paper so that the first line of print appears at the start line of the paper. If you have to do this manually, take the printer off-line before you advance the paper.

If you are using an 8Ø column printer, use the CHARS parameter to allow for proper paging. TBA prints 132 characters per line. Using an 8Ø column printer can cause wrap-around and affect pagination. When the printer is ready, direct the processed code to it.

To process this file, type:

TBA [ENTER]

The first prompt asks for the source file. Type:

FACTOR [ENTER]

TBA searches for the file. You can also enter the entire filespec.

The second prompt asks for the object filespec. Press **[ENTER]** to cause TBA to use FACTOR/BAS as a filename for the processing operation and the finished file.

The third prompt requests information dealing with processing parameters. If you want to send the object listing to the screen, press **[ENTER]**. If you prefer to send the listing to the printer, type:

LP [ENTER]

The fourth prompt requests information about any directives on which you want the program to act. Since the source file does not contain any processing expressions, answer this prompt by pressing **[ENTER]**.

TBA begins to process the file. At the beginning of each pass, a message appears on the screen. Shortly after the Pass 5 message appears, TBA prints the file to the screen or printer and writes the final object file to the diskette.

To view the resulting BASIC program, enter these commands:

```
BASIC [ENTER]
LOAD "FACTOR/BAS"[ENTER]
SAVE "FACTOR/BAS" [ENTER]
```

Since TBA creates an ASCII file, it does not store any of the compression codes BASIC uses in the file. You can load a BASIC program stored in ASCII. It takes more time to load a program stored in ASCII than a program stored in compressed form. To store the program on the diskette in compressed form, and perform subsequent loads of the program faster, TBA instructs you to resave the program.

To compare the source and object codes, LIST or LLIST the program. The object code does not use 1 of the variables defined in source code, but you can locate this variable by inspecting the cross-reference table.

If you complete the processing operation successfully, TBA creates this object code:

```
8 DIM PR!(1Ø),PR$(1Ø)
14 CLS
15 INPUT "ORIGIN OF SCAN"; IN$
17 IF VAL(IN$)<2 THEN 14
18 ST!=INT(VAL(IN$))
19 INPUT "   END OF SCAN"; IN$
2Ø EN!=(VAL(IN$))
21 IF EN!<ST! THEN SWAP ST!,EN!
22 HO$=STRING$(79,61)
23 CLS
27 FOR LO! = ST! TO EN!
3Ø HA!=LO!
31 FA$=""
32 PRINT @ Ø, "factoring "USING"###,###";LO!;
33 PRINT @ 4Ø,"primes found on this scan";
34 PRINT USING"##,###";PS%;
35 PRINT @ 8Ø,"prime factors : ";CHR$(3Ø);
```

```
36 PRINT HO$;: SU!=2
4Ø FOR LP! = 2 TO HA!
41 IF HA!/LP!=INT(HA!/LP!) THEN GOSUB 73
42 NEXT LP!
46 IF VAL(FA$) = LO! THEN GOSUB 81 ELSE GOSUB 87
47 PR!(CO%)=LO!
48 PR$(CO%)=FA$
49 LO%=CO%
53 FOR LP!=Ø TO 1Ø
54 PRINT @(LP!+4,Ø),"";
55 PRINT PR!(LO%), PR$(LO%);
56 PRINT CHR$(3Ø);
57 LO%=LO%-1
58 IF LO%=-1 THEN LO%=1Ø
59 NEXT LP!
63 CO%=CO%+1
64 IF CO%=11 THEN CO% =Ø
65 NEXT LO!
67 END
73 FA$=FA$+STR$(LP!)+" x"
74 PRINT @ 96,FA$;
75 HA!=HA!/LP!
76 LP! = HA! + 1
77 RETURN
81 FA$="* Prime  Number *"
82 PS%=PS%+1
83 RETURN
87 FA$=LEFT$(FA$,LEN(FA$)-1)
88 RETURN
```

Merging Procedures

You do not see the full benefit of source code independent of the line numbers until you complete a program which relies heavily on reusing TBA procedures already created and debugged. Therefore, the merge process itself is important for programming with TBA.

The following 3 program listings represent a main program module, and 2 other programs. The other 2 programs can merge into the main module. These 2 procedures represent modules used in other programs.

After you enter the programs in to the editor, save them to disk under the name specified. The names are important for the merging steps.

MAIN PROGRAM - MAIN/TBA

```
' exercise 2 main body program : filename MAIN/TBA
*TITLE "Exercise 2"
     =INP$,AT%,FIELD%,CENTER$,WIDTH%,DEVICE%, STRING.ARRAY$,LOOP%,LOOP1%
          DIM STRING.ARRAY$(5Ø)
          GOTO @MAIN
'INPUT  PROCEDURE WILL GO HERE
'CENTER PROCEDURE WILL GO HERE
@MAIN
          CLS:PRINT "Enter a sentence of less than 5Ø words"
          PRINT STRING$(79,61)
          PRINT @16Ø,"Enter @ to stop"
          FOR LOOP% = 1 TO 5Ø

PRINT @24Ø,CHR$(3Ø);"Last Entry : ";

PRINT STRING.ARRAY(LOOP%-1)

PRINT @32Ø,"Current Entry ="
               AT%=336:FIELD%=1Ø:GOSUB @INPUT
                 IF INP$ = "@" THEN @DISPLAY.VIDEO
               STRING.ARRAY$(LOOP%)=INP$
          NEXT LOOP%
@DISPLAY.VIDEO
          WIDTH%=8Ø:DEVICE%=Ø
          CLS
          FOR LOOP1%= 1 TO LOOP%-1
                CENTER$=STRING.ARRAY$(LOOP1%)
                GOSUB @CENTER.DISPLAY
                  IF LOOP1% MOD 23 = Ø THEN GOSUB @WAIT.FOR.ENTER
                NEXT LOOP1%
          END
@WAIT.FOR.ENTER
          PRINT "Press <ENTER> to continue";
          AT%=23*8Ø+39: FIELD%=1: INP$="*"
          WHILE INP$<>""
               GOSUB @INPUT
          WEND
          CLS
          RETURN
```

PROCEDURE CENTER/TBA

```
        '
        'merge into any TBA Source file & define CENTER$ as a Global
        'define WIDTH% and DEVICE% as Global
        'ENTRY CONDITIONS:
             'CENTER$ = desired string
             'DEVICE% = Ø for video
             'OR
             'DEVICE% = 1 for printer
             'WIDTH% = total columns
        'USE:
             'GOSUB @CENTER.DISPLAY
        'EXIT CONDITIONS:
             'DEVICE = -1 for normal exit
        '
@CENTER.DISPLAY=FROM.LEFT%,WIDTH1%,LENGTH%
        'width will default to 8Ø
             IF WIDTH% <= 1 THEN WIDTH% = 8Ø
             WIDTH1% = INT(WIDTH%/2)
             LENGTH% = LEN(CENTER$)
        'take center of string if too wide for device
             WHILE  LENGTH%  >  WIDTH%
                  CENTER$ = MID$(CENTER$,(LENGTH%-WIDTH%)/2,WIDTH%)
                  LENGTH% = LEN(CENTER$)
             WEND
             FROM.LEFT% = WIDTH1%-INT(LENGTH%/2)
        'Default is screen
             IF DEVICE% < Ø OR DEVICE% > 1 THEN DEVICE% = Ø
        'screen
             WHILE DEVICE% = Ø
                  PRINT TAB(FROM.LEFT%)CENTER$
                  DEVICE% = -1
             WEND
             WHILE DEVICE% = 1
                  LPRINT TAB(FROM.LEFT%)CENTER$
                  DEVICE% = -1
             WEND
    @EXIT.CENTER
             RETURN
```

PROCEDURE INPUT/TBA

```
'Globals used : FIELD%,AT%,INP$
'ENTRY CONDITIONS :
'                  FIELD% = number of characters wanted (plus 1 ?)
'                  AT%    = character position for input
'USE               :GOSUB @INPUT
'EXIT CONDITIONS   :
'                  INP$    = string returned from keyboard.
'===
'INPUT SUBROUTINE
'===
'
@INPUT=INK$,FLASH.LOC%,FIELD.LEN%,AT.LEN%
@BEGIN.INPUT
        FIELD.LEN%=FIELD%:AT.LEN%=AT%
        INK$="":INP$=""
        PRINT @ AT%,STRING$(FIELD%,138);
@RE.INPUT
        FLASH.LOC% = LEN(INP$)
        PRINT @ AT.LEN% + FLASH.LOC% ,"";
        INK$ = INKEY$
        WHILE INK$ = ""
            INK$ = INKEY$
        WEND
@PROC.INPUT
        IF INK$=CHR$(13)THEN @END.INPUT
        IF INK$=CHR$(8) THEN 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
        IF FIELD.LEN%=Ø THEN PRINT @ AT%,INP$;:GOTO @END.INPUT
@DISP.FLD
        PRINT @ AT%, INP$+STRING$(FIELD.LEN%,138);
        GOTO @RE.INPUT
@END.INPUT
        IF FIELD.LEN%<>Ø THEN PRINT @ AT%,INP$;SPACE$(FIELD.LEN%);
        RETURN
'
'END OF INPUT SUBROUTINE
'
```

Merge instructions for ALEDIT

Enter these 3 programs into ALEDIT, and write them to disk under the 3 file names:

MAIN/TBA
INPUT/TBA
CENTER/TBA

To load all 3 files into ALEDIT, type the following:

```
ALEDIT [ENTER]                  'from TRSDOS Ready
LINPUT/TBA [ENTER]              'Load INPUT procedure
LCENTER/TBA$C [ENTER]           'Chain CENTER procedure
LMAIN/TBA$C [ENTER]             'Chain MAIN body
```

These commands place the 3 files in memory in correct order: the 2 procedures first, followed by the main body. However, the first few lines of MAIN/TBA belong at the top of the program, not buried in the middle of the text. To move them to the top, type:

```
#81                             'goto line 81 which should be
                                'exercise 2 main body , etc.
1                               'Begin Mark this line
[down-arrow] 6 times            'find end of section to be moved
                                'CENTER PROCEDURE WILL GO HERE
2                               'End Mark this line
T                               'goto first line of text
                                'globals used : FIELD%, etc.
M                               'Move Marked text in above here
```

The text is now in the correct order for TBA processing.

Using the filename MERGE/TBA these commands complete processing:

```
WMERGE/TBA [ENTER]              'Write to disk
Q [ENTER]                       'Return to TRSDOS Ready
LIST MERGE/TBA (N,P)[ENTER]     'Hard-copy with line numbers
TOF [ENTER]                     'Printer top of forms
TBA [ENTER]                     'Invoke TBA processor
MERGE/TBA [ENTER]               'Source filespec
MERGE/BAS [ENTER]               'Object filespec
LP [ENTER]                      'Process to printer
[ENTER]                         'no Directives required
```

If TBA reports errors during processing, do not correct them in the file MERGE/TBA. Correct them in INPUT/TBA, CENTER/TBA, or MAIN/TBA. TBA tells you the number of the incorrect line in MERGE/TBA, and your listing helps you identify which of the 3 programs contains that line.

Correct the 3 individual source codes, then start the merge process again.

The processing cycle is now complete, and you can run the program MERGE/BAS from BASIC.

If BASIC reports errors, or the program fails to run as expected, return again to the 3 individual source codes, and correct any problems you encounter.

Notes on the use of ALEDIT

ALEDIT is part of the Assembly Language Development System, ALDS, (Catalog number 26-2Ø12). Certain other language packages for the Model 4 use the same editor as ALDS.

Disadvantages of using ALEDIT include:

1) A partial full screen editor.
2) Global find or replace only acts on the first occurence in a line.
3) Maximum line length is 8Ø characters.
4) Tabs are defined as length 8

You may need lines of more than 8Ø characters because your program includes BASIC lines such as:

```
IF condition THEN action1 : action2 : action ... : etc.
```

You can produce the same effect in lines shorter than 8Ø characters, using a more structured form of code, by either:

```
IF condition THEN GOSUB @procedure.label
```

or:

```
WHILE condition.flag
      action1
      action2
      action ...
      (change condition.flag)
WEND
```

Using these methods also relieves the problem of reducing the available line length by 8 characters every time you use a tab. Alternatively, the TRSDOS KSM filter allows [CLEAR][A], [CLEAR][B], and [CLEAR][C], which you can set up equal to 3, 6, and 9 spaces respectively.

Advantages of ALEDIT include:

1) Offers over 41K of text space.
2) Allows moving text in memory, and some repetitive actions.
3) Does not require line numbers, allows you to address text by line numbers.
4) Requires no special method of saving the file.
5) Functions with KSM filter. Some editors do not respond to KSM.

Many editors do not allow this amount of text. During the debugging or merging stages of program development, it is convenient to directly reach a particular line of source. You do not need special instructions to save a file, and the KSM filter speeds up the keyboard work of entering source code.

Merge Instructions for BASIC

Enter the 3 programs above into BASIC. Use the AUTO feature of the BASIC editor to provide line numbers. Which line numbers you assign to the source code lines is unimportant. If you omit a line, insert it. When you complete the source of each program, type:

```
RENUM [ENTER]
```

to renumber the program in memory to start at Line 1Ø and increase in increments of 1Øs. Save each file to disk using the syntax:

```
SAVE "filename/ext:d",A [ENTER]
```

and the filenames:

```
MAIN/TBA
INPUT/TBA
CENTER/TBA
```

After you complete all 3 files and save them to disk, these instructions renumber each file, giving them line numbers which position the routines when you merge them. At the BASIC Ready prompt, type:

```
LOAD "INPUT/TBA" [ENTER]          'load INPUT into memory
RENUM 1ØØ,1Ø,1 [ENTER]            'renumber to 1ØØ increment 1
                                  'old 1st line (1Ø) becomes 1ØØ
LIST [ENTER]                      'confirm above.
SAVE "INPUT/TBA",A [ENTER]        'to disk.
LOAD "CENTER/TBA" [ENTER]         'load CENTER into memory
RENUM 2ØØ,1Ø,1 [ENTER]            'renumber to 2ØØ increment 1
                                  'old 1st line (1Ø) becomes 2ØØ
LIST [ENTER]                      'to confirm
SAVE "CENTER/TBA",A [ENTER]       'to disk
LOAD "MAIN/TBA" [ENTER]           'load MAIN into memory
LIST 8Ø [ENTER]                   'should show @MAIN
RENUM 3ØØ,8Ø,1 [ENTER]            'renumber to 3ØØ increment 1
                                  'old 8Ø becomes 3ØØ
LIST [ENTER]                      'to confirm
MERGE "INPUT/TBA" [ENTER]         'merge INPUT into MAIN
MERGE "CENTER/TBA" [ENTER]        'merge CENTER into both
```

```
LIST [ENTER]                        '1Ø-7Ø = 1st part of MAIN
                                    '1ØØ on = INPUT
                                    '2ØØ on = CENTER
                                    '3ØØ on = 2nd part of MAIN
RENUM 1,1Ø,1 [ENTER]                'renumber to 1 increment 1
                                    'old 1st line (1Ø) becomes 1
LIST [ENTER]                        'to confirm
LLIST [ENTER]                       'hard copy with line numbers
SAVE "MERGE/TBA",A [ENTER]          'complete to disk
SYSTEM [ENTER]                      'to TRSDOS Ready
```

Once you complete the file MERGE/TBA, type:

```
TOF [ENTER]                         'Printer top of forms
TBA [ENTER]                         'Invoke TBA processor
MERGE/TBA [ENTER]                   'Source filespec
MERGE/BAS [ENTER]                   'Object filespec
LP [ENTER]                          'Process to printer
[ENTER]                             'no Directives required
```

If TBA reports errors during processing, do not correct them in the file MERGE/TBA. Correct them in INPUT/TBA, CENTER/TBA, or MAIN/TBA. TBA tells you the line number in MERGE/TBA which contains the error, and the listing helps you identify which of the 3 programs contains that line.

Correct the 3 individual source codes, then start the merge process again. Keep in mind that line numbers in the 3 source codes are **not** identical to those in MERGE/TBA.

The processing cycle is now complete, and you can run the program MERGE/BAS from BASIC.

If BASIC reports errors, or the program fails to run as you expected, return to the 3 individual source codes and correct any problems you encounter.

Notes on Using the BASIC editor

Advantages to Using BASIC include:

1) Availablity
2) Familiarity
3) Less likely to create a line too long for TBA to process.
4) Runs with KSM filter and JCL processor.

Disadvantages include:

1) Line based editor, with no global find or replace functions.
2) Need to use RENUM, SAVE, MERGE, and DELETE several times each time you move a line.
3) Must make each SAVE with the [,A] option to merge and process with TBA

Example program #3

The third example program is longer and more complicated. It follows the outline structure recommended for a stand-alone TBA source program, described in "Enhancing Operation and Speed." Reread that section of the manual before continuing.

This program allows you to print a variable number of labels with the same content, such as you require for a floppy backup of a hard disk. Two directives in the program allow for different types of labels. The program also allows you to maintain a library of labels for reuse.

Because the program includes 1 of the procedures from Example 2, you do not need to type it in again. It also includes several other programs that you treat as procedures. Merge them into the program using the same sort of method described in "Merge Procedures Using ALEDIT, or BASIC."

The following is the file as it appears after you complete the merge process, but before TBA processes it:

```
*TITLE "LABELS/TBA Ex 3"
=ARRAY.SIZE%,LABEL.ARRAY$,FIELD%,AT%,INP$
=GLOBAL.LOOP%,GLOBAL.LOOP1%,TEMP%,TEMPORARY$,LABEL.LENGTH%
=LINE.LENGTH%,NAME.OF.FILE$,CLEAR.LINE$,CLEAR.SCREEN$,NO.OF.LABELS%
=AT.ENTER%,INK$,STAT$
        GOTO @START
'
'===
'INPUT SUBROUTINE
'===
'
@INPUT=INK$,FLASH.LOC%,FIELD.LEN%,AT.LEN%
@BEGIN.INPUT
        FIELD.LEN%=FIELD%:AT.LEN%=AT%
        INK$="":INP$=""
        PRINT @ AT%,STRING$(FIELD%,138);
@RE.INPUT
        FLASH.LOC% = LEN(INP$)
        PRINT @ AT.LEN% + FLASH.LOC% ,"";
        INK$ = INKEY$
        WHILE INK$ = ""
            INK$ = INKEY$
        WEND
@PROC.INPUT
          IF INK$=CHR$(13)THEN @END.INPUT
          IF INK$=CHR$(8) THEN 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
          IF FIELD.LEN%=Ø THEN PRINT @ AT%,INP$;:GOTO @END.INPUT
@DISP.FLD
        PRINT @ AT%, INP$+STRING$(FIELD.LEN%,138);
        GOTO @RE.INPUT
@END.INPUT
          IF FIELD.LEN%<>Ø THEN PRINT @ AT%,INP$;SPACE$(FIELD.LEN%);
        RETURN
'
'END OF INPUT SUBROUTINE
'
```

```
'===
'PRESS ENTER TO CONTINUE SUBROUTINE
'===
'
@PRESS.ENTER
        PRINT @ AT.ENTER%,"PRESS <ENTER> TO CONTINUE"
@INPUT.ENTER
        AT%= AT.ENTER%+3Ø: FIELD%=1:GOSUB @INPUT
          IF INP$<>"" THEN @INPUT.ENTER
        RETURN
'
'END OF PRESS ENTER SUBROUTINE
'
'===
'DISPLAY LABEL SUBROUTINE
'===
'
@DISPL.LABEL
        PRINT @ (5,Ø),CLEAR.SCREEN$
        FOR GLOBAL.LOOP%=1 TO ARRAY.SIZE%
            PRINT @ (GLOBAL.LOOP%+5,Ø),"LINE #";
            PRINT GLOBAL.LOOP%;" ";
            PRINT  LABEL.ARRAY$(GLOBAL.LOOP%)
        NEXT GLOBAL.LOOP%
        RETURN
'
'===
'STOP PRINTING ROUTINE
'===
'
@STOP.PRINTING
        PRINT @(23,Ø),CLEAR.SCREEN$;
        PRINT "DO YOU WISH TO STOP PRINTING   (Y/N)";
@INPUT.STOP
        AT%=(8Ø*23+38)
        FIELD%=2 : INP$ = ""
        WHILE INP$<> "Y" AND INP$<> "N"
           GOSUB @INPUT
        WEND
        PRINT @ (23,Ø),CLEAR.SCREEN$;
        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$(3Ø):CLEAR.SCREEN$=CHR$(31)

'
'===
'IF THE DIRECTIVE "CHARS" WAS PASSED, TAKE THE INPUT FOR THE
'NUMBER OF CHARACTERS PER LINE
'===
'
*IF CHARS
@CHARS.INPUT
        CLS:PRINT @ (8,12), "";
        PRINT "ENTER NUMBER OF CHARACTERS PER LABEL"
        AT%=(8Ø*8+49):FIELD%=2:GOSUB @INPUT
          IF INP$="@" THEN GOTO @END.PROGRAM
          IF INP$="" THEN @CHARS.INPUT
        FOR GLOBAL.LOOP%=1 TO LEN(INP$)
            TEMPORARY$=MID$(INP$,GLOBAL.LOOP%,1)
              IF TEMPORARY$<"Ø" OR TEMPORARY$>"9" THEN @CHARS.INPUT
         NEXT GLOBAL.LOOP%
         LABEL.LENGTH%=VAL(INP$)
         GOTO @INPUT.FILE
*END
'
'
        LABEL.LENGTH%=35
'
'
```

```
@INPUT.FILE
        NAME.OF.FILE$=""
        FOR GLOBAL.LOOP%=1 TO ARRAY.SIZE%
            LABEL.ARRAY$(GLOBAL.LOOP%)=" "
        NEXT GLOBAL.LOOP%
        CLS:PRINT TAB(16)"LABEL PRINTING PROGRAM"
        PRINT @(5,1Ø),"DO YOU WISH TO USE AN EXISTING FILE (Y,N,@)
        AT%=(5*8Ø+55):FIELD%=2:GOSUB @INPUT
          IF INP$="@" THEN @END.PROGRAM
          IF INP$="N" THEN @BUILD.LABEL
          IF INP$<>"Y" THEN @INPUT.FILE
@ENTER.FILE
        PRINT @(5,1Ø),CLEAR.LINE$;"ENTER THE NAME OF THE FILE"
        AT%=(5*8Ø+42):FIELD%=15:GOSUB @INPUT
          IF INP$="@" THEN @INPUT.FILE
        ON ERROR GOTO @NO.SUCH.FILE
        NAME.OF.FILE$=INP$
        OPEN"I",1,NAME.OF.FILE$
        ON ERROR GOTO Ø
        TEMP%=1
        STAT$="OK"
          IF EOF(1) THEN STAT$="MT"
        WHILE STAT$="OK"
            LINE INPUT#1, LABEL.ARRAY$(TEMP%)
              IF EOF(1) THEN STAT$="EOF"
            LINE.LENGTH%=LEN(LABEL.ARRAY$(TEMP%))
              IF LINE.LENGTH%>LABEL.LENGTH% THEN STAT$="BAD"
            TEMP%=TEMP%+1
              IF TEMP%>ARRAY.SIZE% AND STAT$<>"EOF" THEN STAT$="BAD"
        WEND
        CLOSE
          IF STAT$="BAD" THEN GOTO @WRONG.PROG
          IF STAT$<>"EOF" THEN GOTO @INPUT.FILE
        GOSUB @DISPL.LABEL
        GOTO @EDIT.LABEL
@BUILD.LABEL
        PRINT @ (1,Ø),CLEAR.SCREEN$
        FOR GLOBAL.LOOP%=1 TO ARRAY.SIZE%
            PRINT @ (GLOBAL.LOOP%+5,Ø),"LINE #";GLOBAL.LOOP%
            AT%=((GLOBAL.LOOP%+5)*8Ø+1Ø)
            FIELD%=LABEL.LENGTH%:GOSUB @INPUT
              IF INP$<>"@" THEN GOTO @BUILD.LOOP
              IF GLOBAL.LOOP%=1 THEN GOTO @INPUT.FILE
           GOTO @BUILD.LABEL
```

```
@BUILD.LOOP
              IF INP$="" THEN INP$=" "
            LABEL.ARRAY$(GLOBAL.LOOP%)=INP$
         NEXT GLOBAL.LOOP%
@EDIT.LABEL
         PRINT @ (23,Ø),"ENTER COMMAND ";
         PRINT "<Y> IF CORRECT, LINE # TO  CORRECT, ";
         PRINT "<@> TO STOP";
         AT%=(23*8Ø+7Ø):FIELD%=2:GOSUB @INPUT
           IF INP$="@" THEN @INPUT.FILE
           IF INP$="Y" THEN @PRINT.LABEL
         TEMP%=VAL(INP$)
           IF TEMP%<1 OR TEMP%>ARRAY.SIZE% THEN @EDIT.LABEL
         AT%=(TEMP%+5)*8Ø
         PRINT @ AT%, "LINE #"; TEMP%;
         AT% = AT% +1Ø : FIELD%=LABEL.LENGTH%
         GOSUB @INPUT
         LABEL.ARRAY$(TEMP%)=INP$
         GOTO @EDIT.LABEL
@PRINT.LABEL
         PRINT @ (23,Ø),CLEAR.SCREEN$;
         PRINT"NUMBER OF LABELS TO PRINT? (<ENTER>=1, <@> TO  STOP)";
         AT%=(23*8Ø+7Ø):FIELD%=3:GOSUB @INPUT
           IF INP$="@" THEN @EDIT.LABEL
           IF INP$=""THEN NO.OF.LABELS%=1 ELSE NO.OF.LABELS%=VAL(INP$)
           IF NO.OF.LABELS%<1 THEN @PRINT.LABEL
         FOR GLOBAL.LOOP%=1 TO NO.OF.LABELS%
             INK$=INKEY$:IF INK$="@" THEN GOSUB @STOP.PRINTING:
               IF INP$="Y" THEN @SAVE.FILE
             FOR GLOBAL.LOOP1%=1 TO ARRAY.SIZE%
                 LPRINT LABEL.ARRAY$(GLOBAL.LOOP1%)
             NEXT GLOBAL.LOOP1%
         NEXT GLOBAL.LOOP%
         PRINT @ (23,Ø),CLEAR.SCREEN$;
         PRINT "DO YOU WISH TO PRINT MORE? (Y/N)";
@PRINT.MORE
         AT%=(23*8Ø+7Ø):FIELD%=2:GOSUB @INPUT
           IF INP$="Y" THEN @PRINT.LABEL
           IF INP$<>"N" THEN @PRINT.MORE
@SAVE.FILE
         PRINT @ (23,Ø),CLEAR.SCREEN$;
         PRINT "DO YOU WISH TO SAVE THIS FILE (Y/N)";
         AT%=(23*8Ø+7Ø):FIELD%=2:GOSUB @INPUT
           IF INP$="N" THEN @INPUT.FILE
           IF INP$<>"Y" THEN @SAVE.FILE
```

```
@FILENAME
        PRINT @ (23,Ø),CLEAR.SCREEN$;
        PRINT "ENTER FILENAME (<ENTER>=SAME NAME)";
        AT%=(23*8Ø+36):FIELD%=15:GOSUB @INPUT
          IF INP$<>"" THEN NAME.OF.FILE$=INP$:GOTO @WRITE.FILE
          IF NAME.OF.FILE$="" THEN @FILENAME
@WRITE.FILE
        ON ERROR GOTO @CANNOT.WRITE
        OPEN "O",1,NAME.OF.FILE$
        FOR GLOBAL.LOOP%=1 TO ARRAY.SIZE%
            PRINT#1, LABEL.ARRAY$(GLOBAL.LOOP%)
        NEXT GLOBAL.LOOP%
        CLOSE
        ON ERROR GOTO Ø
        GOTO @INPUT.FILE
@NO.SUCH.FILE
        PRINT @ (23,1Ø),CLEAR.LINE$;
        PRINT "CANNOT USE FILE --->";NAME.OF.FILE$;
        PRINT @ (7,1Ø),""
        PRINT "THE ERROR THAT OCCURRED IS ERROR #";ERR
        AT.ENTER%=(9*8Ø+1Ø)
        GOSUB @PRESS.ENTER
        PRINT @ (5,1Ø),CLEAR.SCREEN$
        PRINT NAME.OF.FILE$=""
        RESUME @ENTER.FILE
@WRONG.PROG
        PRINT @ (5,1Ø),CLEAR.SCREEN$;
        PRINT "CANNOT USE THIS VERSION OF THE PROGRAM ";
        PRINT @ (6,1Ø),"TO PROCESS THE FILE ---> ";
        PRINT NAME.OF.FILE$
        AT.ENTER%=(9*8Ø+1Ø):GOSUB @PRESS.ENTER
        NAME.OF.FILE$="":CLOSE:GOTO @INPUT.FILE
@CANNOT.WRITE
        PRINT @ (23,Ø),CLEAR.SCREEN$;
        PRINT "CANNOT SAVE FILE -- ERROR=";ERR;
        PRINT " <R>ETRY OR <@>TO STOP";
```

```
@INPUT.WRITE
        AT%=(23*8Ø+7Ø):FIELD%=2:GOSUB @INPUT
          IF INP$="R" THEN NAME.OF.FILE$="":RESUME @SAVE.FILE
          IF INP$ <> "@" THEN @INPUT.WRITE
        RESUME @END.PROGRAM
@END.PROGRAM
        ON ERROR GOTO Ø:CLS
        PRINT @ (6,3Ø),"LABEL PROGRAM ENDED":
        CLOSE
        END
```

When TBA completes processing without error, you can obtain the object file (exact output depends on the directives you use for processing):

```
6 GOTO 95
14 FI%=FJ%:AT%=AU%
15 IN$="":IO$=""
16 PRINT @ AU%,STRING$(FJ%,138);
18 FL% = LEN(IO$)
19 PRINT @ AT% + FL% ,"";
2Ø IN$ = INKEY$
21 WHILE IN$ = ""
22 IN$ = INKEY$
23 WEND
25 IF IN$=CHR$(13)THEN 37
26 IF IN$=CHR$(8) THEN FI%=FI%+1
27 IF FI%>FJ% THEN 14
28 IF IN$=CHR$(8) THEN IO$=LEFT$(IO$,LEN(IO$)-1)
29 IF ASC(IN$)<32 THEN IN$="": GOTO 34
3Ø IO$=IO$+IN$
31 FI%=FI%-1
32 IF FI%=Ø THEN PRINT @ AU%,IO$;:GOTO 37
34 PRINT @ AU%, IO$+STRING$(FI%,138);
35 GOTO 18
37 IF FI%<>Ø THEN PRINT @ AU%,IO$;SPACE$(FI%);
38 RETURN
47 PRINT @ AV%,"PRESS <ENTER> TO CONTINUE"
49 AU%= AV%+3Ø: FJ%=1:GOSUB 14
5Ø IF IO$<>"" THEN 49
51 RETURN
```

```
6Ø PRINT @ (5,Ø),CM$
61 FOR GL%=1 TO AR%
62 PRINT @ (GL%+5,Ø),"LINE #";
63 PRINT GL%;" ";
64 PRINT LA$(GL%)
65 NEXT GL%
66 RETURN
73 PRINT @(23,Ø),CM$;
74 PRINT "DO YOU WISH TO STOP PRINTING   (Y/N)";
76 AU%=(8Ø*23+38)
77 FJ%=2 : IO$ = ""
78 WHILE IO$<> "Y" AND IO$<> "N"
79 GOSUB 14
8Ø WEND
81 PRINT @ (23,Ø),CM$;
82 RETURN
95 AR%=6
98 DIM LA$(AR%)
99 CL$=CHR$(3Ø):CM$=CHR$(31)
1Ø9 CLS:PRINT @ (8,12), "";
11Ø PRINT "ENTER NUMBER OF CHARACTERS PER LABEL"
111 AU%=(8Ø*8+49):FJ%=2:GOSUB 14
112 IF IO$="@" THEN GOTO 257
113 IF IO$="" THEN 1Ø9
114 FOR GL%=1 TO LEN(IO$)
115 TE$=MID$(IO$,GL%,1)
116 IF TE$<"Ø" OR TE$>"9" THEN 1Ø9
117 NEXT GL%
118 LA%=VAL(IO$)
119 GOTO 127
123 LA%=35
127 NA$=""
128 FOR GL%=1 TO AR%
129 LA$(GL%)=" "
13Ø NEXT GL%
131 CLS:PRINT TAB(16)"LABEL PRINTING PROGRAM"
132 PRINT @(5,1Ø),"DO YOU WISH TO USE AN EXISTING FILE (Y,N,@)
133 AU%=(5*8Ø+55):FJ%=2:GOSUB 14
134 IF IO$="@" THEN 257
135 IF IO$="N" THEN 162
```

```
136 IF IO$<>"Y" THEN 127
138 PRINT @(5,1Ø),CL$;"ENTER THE NAME OF THE FILE"
139 AU%=(5*8Ø+42):FJ%=15:GOSUB 14
14Ø IF IO$="@" THEN 127
141 ON ERROR GOTO 231
142 NA$=IO$
143 OPEN"I",1,NA$
144 ON ERROR GOTO Ø
145 TE%=1
146 ST$="OK"
147 IF EOF(1) THEN ST$="MT"
148 WHILE ST$="OK"
149 LINE INPUT#1, LA$(TE%)
15Ø IF EOF(1) THEN ST$="EOF"
151 LI%=LEN(LA$(TE%))
152 IF LI%>LA% THEN ST$="BAD"
153 TE%=TE%+1
154 IF TE%>AR% AND ST$<>"EOF" THEN ST$="BAD"
155 WEND
156 CLOSE
157 IF ST$="BAD" THEN GOTO 241
158 IF ST$<>"EOF" THEN GOTO 127
159 GOSUB 6Ø
16Ø GOTO 175
162 PRINT @ (1,Ø),CM$
163 FOR GL%=1 TO AR%
164 PRINT @ (GL%+5,Ø),"LINE #";GL%
165 AU%=((GL%+5)*8Ø+1Ø)
166 FJ%=LA%:GOSUB 14
167 IF IO$<>"@" THEN GOTO 171
168 IF GL%=1 THEN GOTO 127
169 GOTO 162
171 IF IO$="" THEN IO$=" "
172 LA$(GL%)=IO$
173 NEXT GL%
175 PRINT @ (23,Ø),"ENTER COMMAND ";
176 PRINT "<Y> IF CORRECT, LINE # TO  CORRECT, ";
177 PRINT "<@> TO STOP";
178 AU%=(23*8Ø+7Ø):FJ%=2:GOSUB 14
179 IF IO$="@" THEN 127
18Ø IF IO$="Y" THEN 19Ø
```

```
181 TE%=VAL(IO$)
182 IF TE%<1 OR TE%>AR% THEN 175
183 AU%=(TE%+5)*8Ø
184 PRINT @ AU%, "LINE #"; TE%;
185 AU% = AU% +1Ø : FJ%=LA%
186 GOSUB 14
187 LA$(TE%)=IO$
188 GOTO 175
19Ø PRINT @ (23,Ø),CM$;
191 PRINT"NUMBER OF LABELS TO PRINT? (<ENTER>=1, <@> TO  STOP)";
192 AU%=(23*8Ø+7Ø):FJ%=3:GOSUB 14
193 IF IO$="@" THEN 175
194 IF IO$=""THEN NO%=1 ELSE NO%=VAL(IO$)
195 IF NO%<1 THEN 19Ø
196 FOR GL%=1 TO NO%
197 IP$=INKEY$:IF IP$="@" THEN GOSUB 73:
198 IF IO$="Y" THEN 21Ø
199 FOR GM%=1 TO AR%
2ØØ LPRINT LA$(GM%)
2Ø1 NEXT GM%
2Ø2 NEXT GL%
2Ø3 PRINT @ (23,Ø),CM$;
2Ø4 PRINT "DO YOU WISH TO PRINT MORE? (Y/N)";
2Ø6 AU%=(23*8Ø+7Ø):FJ%=2:GOSUB 14
2Ø7 IF IO$="Y" THEN 19Ø
2Ø8 IF IO$<>"N" THEN 2Ø6
21Ø PRINT @ (23,Ø),CM$;
211 PRINT "DO YOU WISH TO SAVE THIS FILE (Y/N)";
212 AU%=(23*8Ø+7Ø):FJ%=2:GOSUB 14
213 IF IO$="N" THEN 127
214 IF IO$<>"Y" THEN 21Ø
216 PRINT @ (23,Ø),CM$;
217 PRINT "ENTER FILENAME (<ENTER>=SAME NAME)";
218 AU%=(23*8Ø+36):FJ%=15:GOSUB 14
219 IF IO$<>"" THEN NA$=IO$:GOTO 222
22Ø IF NA$="" THEN 216
222 ON ERROR GOTO 248
223 OPEN "O",1,NA$
224 FOR GL%=1 TO AR%
225 PRINT#1, LA$(GL%)
```

```
226 NEXT GL%
227 CLOSE
228 ON ERROR GOTO Ø
229 GOTO 127
231 PRINT @ (23,1Ø),CL$;
232 PRINT "CANNOT USE FILE --->";NA$;
233 PRINT @ (7,1Ø),""
234 PRINT "THE ERROR THAT OCCURRED IS ERROR #";ERR
235 AV%=(9*8Ø+1Ø)
236 GOSUB 47
237 PRINT @ (5,1Ø),CM$
238 PRINT NA$=""
239 RESUME 138
241 PRINT @ (5,1Ø),CM$;
242 PRINT "CANNOT USE THIS VERSION OF THE PROGRAM ";
243 PRINT @ (6,1Ø),"TO PROCESS THE FILE ---> ";
244 PRINT NA$
245 AV%=(9*8Ø+1Ø):GOSUB 47
246 NA$="":CLOSE:GOTO 127
248 PRINT @ (23,Ø),CM$;
249 PRINT "CANNOT SAVE FILE -- ERROR=";ERR;
25Ø PRINT " <R>ETRY OR <@>TO STOP";
252 AU%=(23*8Ø+7Ø):FJ%=2:GOSUB 14
253 IF IO$="R" THEN NA$="":RESUME 21Ø
254 IF IO$ <> "@" THEN 252
255 RESUME 257
257 ON ERROR GOTO Ø:CLS
258 PRINT @ (6,3Ø),"LABEL PROGRAM ENDED":
259 CLOSE
26Ø END
```

Although you often have to repeat the same instructions, especially during the final debugging stages, do not get discouraged. TBA works well with TRSDOS JCL, especially in debugging a program. TBA inputs sent with a JCL file can process the source, then call up BASIC to run the object program.

Before you start TBA processing, you can load, merge and move text around in your editor from a JCL file. This may not work with all editors. Once you edit, DO filename does the repetitive work for you. If your editor cannot take its input from a JCL file, you can still perform the merge processes from JCL by using the APPEND command. If you find maintaining assorted JCL files repetitive, then consider writing a TBA program which writes the JCL for you.

Finally, the TRSDOS KSM filter can expand 1 key stroke into many; therefore, [CLEAR][A], for example, does all the above for you, and more.

Exercise 4

The last exercise demonstrates how you execute a series of TBA-related processes from a JCL file. The source, along with other files in this exercise, demonstrates 1 method of creating a program by using overlays that the CHAIN MERGE statement in BASIC calls. The program follows the outline structure discussed in "The Use of CHAIN MERGE, and COMMON statements".

RES/TTL

```
*TITLE "Ex 4 Resident"
```

OV1/TTL

```
*TITLE "Ex 4 Overlay 1"
```

OV2/TTL

```
*TITLE "Ex 4 Overlay 2"
```

OV3/TTL

```
*TITLE "Ex 4 Overlay 3"
```

COMMON/BLK

```
=co.resident%,co.which%
common co.resident%, co.which%
' end of #1, start of #2
```

OVØ/TBA

```
@start.ovrlay
cls
print @(12,12), "Resident=";co.resident%
goto @main
'
' end of resident #2, start of resident #3
```

PRCDRS/TBA

```
@dummy.prcdr=lo.string$
lo.string$="Local string (set by resident)"
print @(11,12),lo.string$;
return
' end of #3, start of #4
```

RESMAIN/TBA

```
@main
gosub @dummy.prcdr
print @(13,12), "";
input "overlay 1, 2, or 3 ( Ø=exit) ";co.which%
if co.which%=Ø then goto @end.program
on co.which% goto @ovrlay1,@ovrlay2,@ovrlay3
@main.end
goto @main
'
@ovrlay1
if co.resident%=1 then goto @ovrlay.call
co.resident%=1
chain merge "ov1/bas", @ovrlay.call
'
@ovrlay2
if co.resident%=2 then goto @ovrlay.call
co.resident%=2
chain merge "ov2/bas", @ovrlay.call
'
@ovrlay3
if co.resident%=3 then goto @ovrlay.call
co.resident%=3
chain merge "ov3/bas", @ovrlay.call
'
@ovrlay.call
gosub @start.ovrlay
goto @main.end
' end of #4, start of #5
@end.program
end
```

OV1/TBA

```
@start.ovrlay
rem
print @(12,12), "(OV1) Resident=";co.resident%
return
rem
rem
```

OV2/TBA

```
@start.ovrlay
print @(12,12), "(OV2) Resident=";co.resident%
rem
rem
rem
return
```

PRCDRS2/TBA

Note: this file is only included for the purpose of clarity.
It contains a modified string string literal for demonstration purposes.
It will normally be the procedure group PRCDRS/TBA (above).

```
@dummy.prcdr=lo.string$
lo.string$="Local string (set by [ OV2 ] )"
print @(11,12),lo.string$;
return
' end of #3, start of #4
```

OV3/TBA

```
@start.ovrlay
for x=1 to 99:gosub @ovrlay.prcdr:next x
print @ (12,12), "(OV3) Resident=";co.resident%;:return
@ovrlay.prcdr=ov.string$
ov.string$="overlay string set by [OV3]":print@(11,12),ov.string$;chr$(31);
return
```

REMSIZ1Ø/TBA

```
1 REM
2 REM
3 REM
4 REM
5 REM
```

```
6 REM
7 REM
8 REM
9 REM
1Ø REM
```

With the above files on disk, the following JCL merges, processes, and then runs the resulting program.

Note: The following syntax is appropriate to source files that ALEDIT creates. You can modify it for source files you create on another editor.

TBA4/JCL

Note: To execute, use DO TBA4 <enter>

```
COPY RES/TTL:1 TO RES/MRG:1
APPEND COMMON/BLK:1 TO RES/MRG:1 (STRIP)
APPEND OVØ/TBA:1 TO RES/MRG:1 (STRIP)
APPEND PRCDRS/TBA:1 TO RES/MRG:1 (STRIP)
APPEND RESMAIN/TBA:1 TO RES/MRG:1 (STRIP)
TBA
RES/MRG:1
RES/BAS:1
<enter>
<enter>
COPY OV1/TTL:1 TO OV1/MRG:1
APPEND COMMON/BLK:1 TO OV1/MRG:1 (STRIP)
APPEND OV1/TBA:1 TO OV1/MRG (STRIP)
TBA
OV1/MRG:1
OV1/BAS:1
<enter>
<enter>
BASIC
LOAD "REMSIZ1Ø/TBA:1"
MERGE "OV1/BAS:1"
```

```
SAVE "OV1/BAS:1",A
SYSTEM
COPY OV2/TTL:1 OV2/MRG:1
APPEND COMMON/BLK:1 TO OV2/MRG:1 (STRIP)
APPEND OV2/TBA:1 TO OV2/MRG:1 (STRIP)
APPEND PRCDRS2/TBA:1 TO OV2/MRG (STRIP)
TBA
OV2/MRG:1
OV2/BAS:1
<enter>
<enter>
BASIC
LOAD "REMSIZ1Ø/TBA:1"
MERGE "OV2/BAS:1"
SAVE "OV2/BAS:1",A
SYSTEM
COPY OV3/TTL:1 TO OV3/MRG:1
APPEND COMMON/BLK:1 TO OV3/MRG:1 (STRIP)
APPEND OV3/TBA:1 TO OV3/MRG:1 (STRIP)
TBA
OV3/MRG:1
OV3/BAS:1
<enter>
<enter>
BASIC
LOAD "REMSIZ1Ø/TBA:1"
MERGE "OV3/BAS:1"
SAVE "OV3/BAS:1",A
RUN "RES/BAS:1"
//STOP
```

INDEX